Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

DIPLOMOVÁ PRÁCE

Využití techniky "Model checking" pro vývoj bezpečnostně kritických aplikací

Plzeň 2017 Bc. Tomáš Ausberger

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne ..... Podpis autora .....

Poděkování

Děkuji panu Ing. Pavlu Baldovi, Ph.D. za vedení diplomové práce. Panu profesoru Ing. Miloši Schlegelovi, CSc. za zajímavé téma výzkumu a podklady k modelu regulátoru výkonu výzkumného jaderného reaktoru a Ing. Ondřeji Severovi za podklady k modelu trhací stanice.

Abstrakt

V této práci je nastíněna problematika bezpečnostně kritických systémů. Dále jsou zde popsány principy metody Model checking a možnosti jejího využití při návrhu modelu bezpečnostně kritických systémů. V poslední části této práce je tato metoda použita na dvou zadaných modelech a je vyhodnocen přínos jejího využití.

Klíčová slova: Model checking, verifikace, bezpečnostně kritické systémy, formální metody

Abstract

This thesis outlines the issue of safety-critical systems. Here are also described the principles of the Model Checking method and the possibilities of its use in the design of a model of safety critical systems. In the last part of this work, this method is used on two models and the contribution of its use is evaluated.

Keywords: Model Checking, verification, safety critical systems, formal methods

Obsah

1. Úvod

Tato práce popisuje principy a uplatnění metody Model checking a možnosti jejího využití v rámci verifikace modelu.

1.1 Motivace

Při řešení problematiky řízení systémů můžeme narazit na systémy, u kterých je nutné věnovat zvýšenou pozornost jejich bezpečnosti. Jsou to systémy, jejichž selhání by způsobilo ztráty na životech, zdraví nebo na majetku. Tyto systémy se označují jako bezpečnostně kritické systémy.

Kvůli nárokům na bezpečnost těchto systémů je vývoj řízení pro tyto systémy podřízen pravidlům, která mají za účel organizovat a kontrolovat proces vývoje a zamezit tak vzniku chyby. Vývoj je rozdělen do několika etap, a v rámci těchto etap se ještě dělí na jednotlivé úkoly. Aby se zamezilo vzniku chyb, musí být výsledky jednotlivých úkolů kontrolovány. Jednou z možností kontroly je použití metody Model checking, která se používá ke kontrole modelu řešení ve fázi jeho návrhu.

Výhoda použití metody Model checking spočívá v tom, že se tato metoda používá již ve fázi návrhu řešení. Při správném použití tak odhalí chyby, které vznikly během návrhu modelu, a zamezí tak nákladnému vývoji chybného řešení. Metoda tímto způsobem může ušetřit velkou část nákladů. Chyba, kterou odhalí, by totiž jinak byla do řešení zavedena již na začátku procesu vývoje a zároveň by mohla být odhalena až na jeho konci nebo vůbec.

Další výhoda této metody spočívá v jistotě, kterou může poskytnout o bezpečnosti modelu. Její síla spočívá v ověření stanovených požadavků na systém ve všech možných stavech, kterých může model nabývat. Tím je zajištěno, že model systému při žádném (jakkoliv nestandardním) použití neporuší požadavky, které jsou na něj kladeny [1]. Takovouto jistotu je například pomocí testování prakticky nemožné získat. Pokud navíc metoda zjistí porušení některého z požadavků, poskytne také informace o tom, jakou posloupností kroků se systém do tohoto stavu dostal, což umožňuje rychleji identifikovat a opravit zdroj chyby.

Pro účely, pro které se používá metoda Model checking, byli dříve využíváni matematici, kteří podobných výsledků dosahovali pomocí vytváření důkazů za pomoci analytických formálních metod. Tyto metody však nebyly používány tak často, jak jim jejich tvůrci předpovídali. Důvodem byla cena a časová náročnost vytvoření matematického důkazu pomocí těchto metod. Metoda Model checking dosahuje stejných výsledků za použití počítačů v podstatně kratším čase. Umožňuje tak využít tento typ kontroly mnohem častěji, než tomu bylo dosud zvykem.

1.2 Postup řešení

Cílem této práce je nastínění problematiky vývoje řízení bezpečnostně kritických systémů (viz kapitolu 2), popsání principů metody Model checking (viz

kapitolu 3), výběr vhodného nástroje pro kontrolu modelu (viz kapitolu 4), použití metody Model checking na zvolených příkladech (viz kapitolu 5), vyhodnocení úspěšnosti této metody a navržení postupů pro její praktické použití.

Nejprve bylo zapotřebí nastudovat problematiku bezpečnostně kritických systémů. Tyto systémy mají různou klasifikaci a jsou na ně kladeny různé podmínky. V určitých místech této práce jsou pro názornost použity požadavky na systémy pro řízení jaderného bloku, které patří k velmi přísným. Dále je v této práci také věnována pozornost možnostem zajištění požadované bezpečnosti těchto systémů.

Druhou teoretickou částí této práce, která musela být nastudována, jsou principy metody Model checking a možnosti jejího použití. Kromě principů metody byly studovány i jazyky, pomocí kterých je možné specifikovat požadavky na daný systém. V praxi se při přepisu modelu používá vždy jazyk vybraného nástroje pro kontrolu modelu. Jazyky určené pro zápis požadavků jsou oproti tomu standardizované. Proto je nutné pochopit, jak je pomocí nich možné specifikovat požadavky na systém, nebo do nich případně přepsat požadavky, které byly zadány ve formě lidské řeči.

Praktickou částí této práce je výběr vhodného nástroje a použití metody Model checking na zvolených příkladech. Tyto příklady se od sebe liší svou složitostí a mírou požadované bezpečnosti. U obou byl jejich model přepsán do jazyka zvoleného nástroje a byly pro ně získány požadavky na daný systém ve formě lidské řeči, které byly poté převedeny do formálního jazyka a ověřeny pomocí zvoleného nástroje.

Na závěr byly shrnuty možnosti využití metody Model checking v praxi a byla navržena řešení některých problémů, které brání častějšímu využívání této metody.

2. Bezpečnostně kritické systémy

Bezpečnostně kritické systémy jsou takové systémy, jejichž provozem mohou být ohroženy lidské životy, zdraví nebo majetek. U těchto systémů se proto věnuje zvýšená pozornost zamezení vzniku takových chyb, které by mohly vést k nebezpečným situacím. Při vývoji řízení takového systému se používají různé analytické nástroje (např. systémová analýza, softwarová analýza, analýza rizik), které umožňují zajistit vyšší bezpečnost systému.

Pojem bezpečnost (z anglického pojmu safety) systému je v této práci chápán jako schopnost systému fungovat bez selhání. Pojem zabezpečení (z anglického pojmu security) systému je v této práci chápán jako schopnost systému chránit se před cílenými útoky.

Mezi bezpečnostně kritické systémy patří například jaderné elektrárny, telekomunikace, lékařské přístroje (umělé plíce, inzulínové pumpy, operační robot), armádní výzbroj (navádění raket) a všechny možnosti přepravy – letadla (autopiloti, navigace, řízení motoru), vlaky (dopravní značení, detekce lidí při zavírání dveří, automatické zastavení vlaku) i auta (airbagy, brzdy, pásy, asistenti řízení). S postupnou automatizací procesů se stále častěji objevují úlohy, kde je nutné k systému přistupovat jako k bezpečnostně kritickému [2].

Existuje mnoho způsobů, kterými je možné bránit zanesení chyb do systému. Jedním z nejčastěji používaných způsobů je testování systému. Testováním lze nalézt některé chyby systému, které jsou objevovány při zkoušení předem definovaných scénářů. Tímto způsobem však nelze prokázat naprostou nepřítomnost chyb v systému. Proto je nutné věnovat pozornost celému procesu vývoje a pokusit se zamezit vzniku chyb už před samotným testováním. Toho je možné docílit například dodržováním vhodně zvolených pravidel, kterým proces vývoje podléhá.

Ačkoliv se jedná o řízení systémů, které jsou potenciálně nebezpečné (např. jaderný reaktor), je nutné mít na paměti, že všechny chyby nejsou kritické. Různé chyby můžou mít různý vliv na chod systému. Chyby, které mohou způsobit selhání systému, jsou důležitější, než chyby, které systém dokáže přejít bez úhony. Často se například namísto zajištění absolutní bezpečnosti jednotlivé části používá redundance, která má za úkol zajistit, aby nenastala chyba ve všech oddělených větvích ve stejný čas a tedy aby vždy alespoň část systému fungovala správně. Vývoj bývá většinou omezen časem a financemi, proto musí být stanovena "rozumná míra", s kterou se k bezpečnosti systému přistupuje. Taková míra může být stanovena například pomocí analýzy rizik nebo kvalifikovaným odhadem odborníka.

2.1 Klasifikace systémů

Bezpečnostně kritické systémy se dají klasifikovat podle způsobu, jakým reagují na výskyt chyby a na selhání systému. Zatímco u některých systémů jsou chyby ošetřovány, jiné počítají s jejich výskytem nebo se po jejich výskytu vrací do předem určeného stavu. Bezpečnostně kritické systémy lze tímto způsobem rozdělit do několika skupin [2]:

- Provozovatelné při selhání (Fail-Operational)

- Bezpečné při selhání (Fail-Safety)
- Zabezpečené při selhání (Fail-Security)
- Pasivní při selhání (Fail-Passive)
- Odolné vůči chybám (Fault-Tolerant)

Systémy, které jsou při selhání provozovatelné (Fail-Operational), pracují i poté, co selže kritická část řízení. Na tomto principu pracují například výtahy a pasivně bezpečné jaderné reaktory. Tento způsob nemusí být považován za bezpečný, systém je totiž v provozu i po ztrátě kontroly.

Systémy, které jsou při selhání bezpečné (Fail-Safety), se při selhání kritické části řízení vrátí do stavu, který je považován za bezpečný. Systém se v tomto případě musí umět dostat z jakéhokoliv stavu do stanoveného bezpečného stavu (obecně může být bezpečných stavů více), v němž počká na odstranění závady nebo na další instrukce.

Systémy, které jsou při selhání zabezpečené (Fail-Security), při selhání kritické části řízení maximalizují své zabezpečení proti útokům a v tomto stavu čekají na další instrukce.

Systémy, které jsou při selhání pasivní (Fail-Passive), fungují po selhání kritické části řízení dále a upozorní na tuto skutečnost člověka nebo jiný systém, který poté převezme řízení systému sám. Příkladem je například autopilot v letadle, který umožní pilotovi převzít řízení a přistát.

Systémy, které jsou odolné vůči chybám (Fault-Tolerant), odolávají svému selhání reagováním na nastalé chyby. Pro tento účel se využívá například redundance, která umožňuje nahrazení porušené části systému jinou částí se stejnou funkcí.

U každého typu systému je poté kladen jiný důraz na různé části vývoje, jsou upřednostňovány jiné metody pro zajištění bezpečnosti a jsou pro ně specifikovány jiné požadavky na chování.

2.2 Standardní životní cyklus vývoje

Aby se zamezilo zavádění chyb do systému během procesu vývoje, podléhá tento proces určitým pravidlům. Ačkoliv se detailní proces vývoje může pro různé systémy lišit například jiným výběrem nástrojů pro zajištění bezpečnosti, existují pravidla, která jsou všem systémům společná. Typickým příkladem je rozdělení procesu vývoje do několika fází [3]:

1. Specifikace požadavků
2. Návrh řešení
3. Implementace
4. Testování
5. Odstranění závad (ladění)
6. Nasazení

7. Správa

Na začátku vývoje musí být identifikován řešený problém a rizika, kterým je nutné se v rámci vývoje vyvarovat. Chybně stanovené požadavky, špatně identifikovaný problém nebo špatně vyhodnocená rizika se mohou projevit na délce a ceně celého vývoje, ale také nemusí být odhaleny vůbec a mohou se objevit až za běhu systému, kde mohou způsobit nebezpečné situace. Proto je této fázi věnována velká pozornost. Často se provádí analýza rizik nebo odborný odhad, ve kterém se uplatňují zkušenosti z práce s podobnými systémy. Jelikož se určité typy úloh často opakují, jsou vytvářeny normy, které standardizují některé požadavky na systém už v závislosti na oblasti, do které problém spadá. Proces vývoje potom odpovídá zvolené normě, která stanovuje určité body, které je nutné pro zajištění požadované bezpečnosti splnit.

Ve druhé fázi vývoje se navrhuje model řešení daného problému, který splňuje podmínky vyplývající z požadavků na systém. Obvykle se iterativně mění navržené řešení tak, aby splňovalo všechny zadané požadavky na systém. Výsledné řešení je vhodné v každé iteraci simulovat, aby se zjistilo, zda se chová podle očekávání. Silnějším nástrojem pro ověření správnosti modelu je však formální analýza. Zatímco pomocí simulace lze simulovat pouze část podmínek, formální analýza poskytuje matematický důkaz o tom, zda model splňuje nebo porušuje zadané požadavky na systém. Platný důkaz je zárukou jistoty o správnosti chování ověřeného modelu. Nalezením chyb ve fázi návrhu se šetří velké množství času a peněz, které by jinak byly investovány do implementace, testování a opravy nefunkčního nebo neúplného řešení.

Při implementaci navrženého řešení se postupně implementují jednotlivé části návrhu. Během této fáze musí být věnována pozornost tomu, aby implementace výsledného řešení probíhala podle zadaného návrhu, předepsaných pravidel a v požadované kvalitě. Zároveň její výsledek musí splňovat zadané požadavky. Chyby, které jsou do systému zavedeny v této fázi musí být zachyceny při testování nebo pomocí statické analýzy. Nalezené chyby by měly být opraveny a implementovány v nové verzi řešení.

Ve fázi testování se kontroluje, zda implementované řešení odpovídá návrhu a zadaným požadavkům. Tato fáze často probíhá paralelně s implementací, přičemž každá implementovaná část je ihned testována a předána zpět k přepracování, pokud jsou v ní nalezeny chyby. Testování zároveň probíhá i jako závěrečná kontrola hotového řešení před nasazením. Pokud se této fázi věnuje dostatek úsilí, je testováním možné odhalit velkou část chyb, které byly do systému zavedeny během implementace.

Ladění probíhá vždy po odhalení chyby v implementovaném řešení. Během ladění jsou prováděny pokusy se scénářem, během kterého se chyba objevila. Většinou je cílem najít zdroj chyby a opravit ho. Někdy ovšem může být výhodnější zdroj chyby neopravovat, ale místo toho ho na některé úrovni obejít.

Během nasazování hotového řešení jsou prováděny závěrečné testy pro přijetí zákazníkem. Ověřuje se, zda celý produkt odpovídá zadaným požadavkům a zda skutečně požadovaným způsobem řeší zadaný problém. Na začátku této fáze se nasazuje do provozu připravená konfigurace systému a odstraňují se poslední nalezené závady.

Poslední fází vývoje je správa hotového řešení, kdy se hotové řešení udržuje v chodu a v nutných případech jsou na něm dodělávány opravy závažných chyb.

Z provozu řešení jsou získávány zkušenosti, postřehy a náměty pro další vývoj.

Vedle procesu vývoje mohou existovat další podpůrné procesy, jejichž účelem je zajistit bezpečnost vývoje. Náplní takového procesu může být například kontrola správného provedení všech činností podle nastavených pravidel, tvorba dokumentace, školení personálu a organizace zúčastněných lidí vykonávajících různé činnosti. Všechny tyto procesy by měly být definovány před započetím vlastního procesu vývoje.

2.3 Metody pro zajištění bezpečnosti

Existuje celá řada metod pro zajištění bezpečnosti. Jednotlivé metody se liší zabezpečovanou fází vývoje, způsobem, kterým pomáhají zvyšovat bezpečnost výsledného systému, možnostmi a náročností svého použití během procesu vývoje. Některé metody se používají v každém procesu vývoje, jiné jsou vázané na určitou oblast, typ problému nebo velikost aplikace. Výběr nástrojů a metod používaných pro zajištění bezpečnosti vývoje by měl proběhnout před započetím samotného procesu vývoje. Dodržování těchto postupů pak musí být vyžadováno a kontrolováno během celého procesu vývoje a o provedených akcích musí být vedeny požadované záznamy. Celý proces kontroly musí být zabezpečen stejně jako samotný proces vývoje.

Pro sjednocení požadavků na systémy, které patří do určité oblasti, jsou vytvářeny mezinárodní normy, které umožňují standardizovat pro danou oblast (například jadernou energetiku [4] nebo automobilový průmysl [5]) požadovanou míru bezpečnosti. Díky využití norem je pro různé firmy a instituce, které se zabývají vývojem systémů patřících do dané oblasti, snazší spolupracovat. Procesy vývoje těchto firem a institucí totiž často podléhají podobným pravidlům, která ve výsledku vždy splňují vybranou normu. Využití normy zároveň ulehčuje prokázání určitého stupně zajištěné bezpečnosti, její splnění lze pomocí auditu prokázat i zpětně. V jednotlivých normách jsou definovány požadavky na kvalitu procesu vývoje, pomocí kterých je určeno například, jak má být organizována a ověřována práce, jakým způsobem musí být vytvářeny a uskladňovány záznamy a které části systému jsou kritické.

Mezi metody, které lze použít pro zvýšení bezpečnosti procesu vývoje, patří například specifikace pravidel a postupů, které jsou poté využívány během implementace. U návrhu software se může jednat například o přijetí některého standardu pro psaní (například MISRA C [7] [6]), které umožňují sjednotit zápis zdrojového kódu. V takovém kódu je snazší nalézt chybu, kterou do něj někdo zavedl nesprávnou úpravou. Podobná pravidla, která mají za úkol zvýšit přehlednost a snížit pravděpodobnost zavedení chyby do systému během implementace, existují pro každou oblast.

Další možnou variantou, kterou lze použít pro zvýšení bezpečnosti, je specifikace škály testů a testovacích postupů, které mají za úkol odhalit chyby zavedené do systému během implementace návrhu řešení. Tyto testy mohou pokrývat různě velké části systému a mohou být prováděny automaticky nebo ručně. Často je možné se setkat s tendencí kompenzovat nedostatečné zajištění bezpečnosti procesu vývoje důkladným testováním, to však samo o sobě nemusí stačit. Testováním není možné provést vyčerpávající kontrolou systému a tudíž pomocí něj nelze prokázat nepřítomnost chyby v systému.

Pro zlepšení bezpečnosti je doporučeno použít ve fázích specifikace požadavků a návrhu řešení analytické nástroje a inženýrský přístup.

2.3.1 Analýza požadavků

Klíčovou fází procesu vývoje je analýza a specifikace požadavků na systém. Při špatné specifikaci požadavků může proběhnout celý proces vývoje, jehož výsledkem bude řešení, které vyhovuje zadaným požadavkům, a tedy neobsahuje chyby, ale přesto nevyhovuje svému účelu. Během této fáze by měly být požadavky shromažďovány a vhodně interpretovány (například pomocí dokumentů, grafů, příkladů, prototypů). Výsledné požadavky se mohou týkat architektury, struktury systému, funkcionality, výkonu, návrhu a samozřejmě všeho, co od systému vyžaduje zákazník [8].

Při specifikaci požadavků je třeba myslet na to, že výsledný systém musí být bezpečný, proto mezi zadanými požadavky musí být i takové, které vyplývají z potřeby bezpečného systému. Pro tyto účely se používá analýza rizik, která pomáhá odhalit slabá a silná místa systému. Získané znalosti lze dále použít pro úpravu požadavků na systém tak, aby byla co nejvíce chráněna kritická místa systému.

Při analýze rizik je potřeba zjistit jaká rizika u daného systému hrozí, jak častý je jejich výskyt a jaké by byly případné dopady nastalé rizikové situace. Seznam možných rizik je většinou vytvářen odborníky, kteří mají zkušenosti s bezpečnostními riziky v dané oblasti. Četnost nebo pravděpodobnost výskytu dané chyby je přiřazena pomocí některé stochastické metody, historických dat nebo kvantifikovaným odhadem odborníka. Využívá se i pseudo-kvantifikovaný odhad, kde odborník přiřazuje jako četnost výskytu jednu z následujících možností [9]:

1. Nepravděpodobné
2. Vzácné
3. Občasné
4. Možné
5. Časté

Podobně se podle možných následků nastalé situace klasifikuje i závažnost zkoumaného rizika:

1. Bez vlivu na bezpečnost
2. Velmi malý efekt – bez úrazů a škod
3. Malý efekt – může způsobit menší škody či zranění
4. Kritické – způsobí ztrátu primárních funkcí a narušení bezpečnosti, krok od katastrofy (maximálně 1 mrtvý)
5. Katastrofické – způsobí neovladatelnost systému, systém přestává být bezpečným, jsou ohroženy životy lidí

Podobným způsobem může být hodnocena detekovatelnost nastalé situace, latence (doba mezi objevením rizikové situace a jejím efektem) nebo její výraznost (jak snadno ji může operátor objevit).

Největší pozornost se při specifikaci požadavků věnuje rizikům s častým výskytem a s největším dopadem. Nejzákeřnější jsou však situace, které mají velký dopad a malou četnost výskytů. Tyto situace nebývají tak dobře ošetřené a můžou tak způsobit kritické selhání systému.

Specifikace požadavků na systém před vlastním návrhem umožňuje u jednotlivých návrhů řešení kontrolovat splnění těchto požadavků. Pro tuto kontrolu je možné využít například formální metody. Postupným iterováním je možné vytvořit návrh, který splňuje všechny požadavky na systém. Případně je možné tímto způsobem identifikovat nesplnitelné nebo obtížně splnitelné požadavky, jejichž požadování je potřeba znovu zvážit.

2.3.2 Tolerance k vadám

Protože lze jen obtížně odhalit všechny rizikové situace daného systému, vytvářejí se často systémy, které jsou robustní (například řízení jaderného reaktoru). Tyto systémy používají pro zajištění bezpečnosti i způsoby, které umožňují vyměnit při poruše havarovanou část za jinou část, která je provozuschopná a která poté vykonává funkci odstavené části. Tyto způsoby jsou uvedeny v následujícím seznamu [10]:

- Replikace
- Redundance
- Různorodost

Replikace je metoda, která využívá více identických instancí systému nebo subsystému. Výsledky jsou zpracovány paralelně každou instancí zvlášť a o celkovém výsledku rozhoduje hlasování, které je určeno dosaženými výsledky jednotlivých instancí.

Redundance je metoda, která využívá více identických instancí systému nebo subsystému, z nichž je ovšem vždy aktivní pouze jediná. Při selhání aktivní instance je vybrána nová fungující instance, která nahradí nefungující aktivní instanci.

Různorodost (Diversity) je metoda, která využívá více různých instancí daného systému nebo subsystému. Tím je zajištěno, že každá instance je vytvořena jiným postupem a obsahuje jiné chyby, které mnohem pravděpodobněji nenastanou ve stejný čas. Zvyšuje se tím pravděpodobnost toho, že vždy zůstane alespoň jedna funkční instance.

Výhodou tolerance systému k vadám je možnost systému reagovat na nepředvídatelné události, na stárnutí a opotřebení jednotlivých částí i na neobjevené vady zanesené do systému během vlastního procesu vývoje nebo při použití komponent od subdodavatele. Poškozené části lze často znovu opravit nebo vyměnit za jiné, aniž by musel být přerušen provoz systému.

2.3.3 Verifikace a validace

Verifikace a validace jsou dva důležité pojmy, které jsou často používané v souvislosti s procesem vývoje. Tyto pojmy mají několik různých definic, přesto však bývají často zaměňovány a bývá komolen jejich význam. V této práci jsou použity tyto pojmy tak, jak jsou definovány v českém překladu mezinárodní normy IEC 60880 [4]:

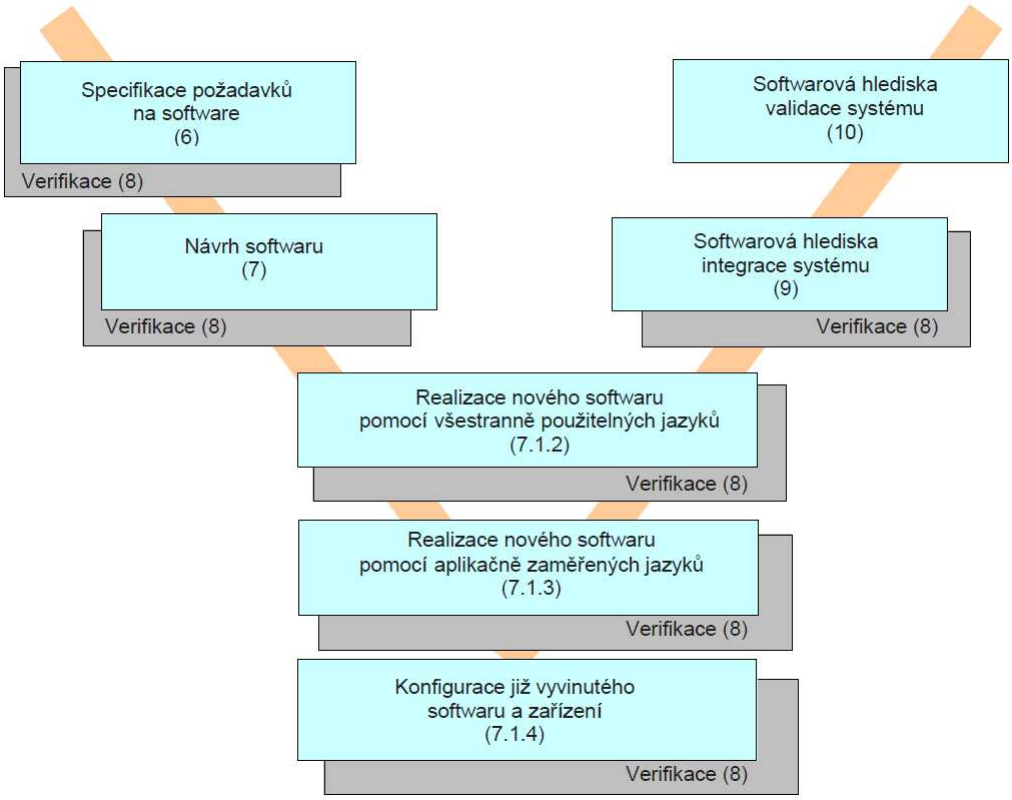
- Verifikace – potvrzení zkouškou a doložením objektivních důkazů, že výsledky činnosti splňují cíle a požadavky definované pro tuto činnost.
- Validace – potvrzení zkouškou a poskytnutím dalších důkazů, že systém jako celek splňuje příslušné požadavky specifikací (funkčnost, doba odezvy, odolnost proti vadám, robustnost).

Verifikace je tedy ověření správného provedení jakékoliv činnosti. Na obrázku 2.1 je tato část znázorněna šedými obdélníky. Je vidět že verifikována musí být každá činnost s výjimkou validace. Zároveň je ale verifikací i samotné testování, jak je znázorněno na obrázku 2.2. Je to proto, že činnosti ve fázi testování mají za úkol ověřit, zda bylo implementací (případně návrhem) dosaženo požadovaného chování.

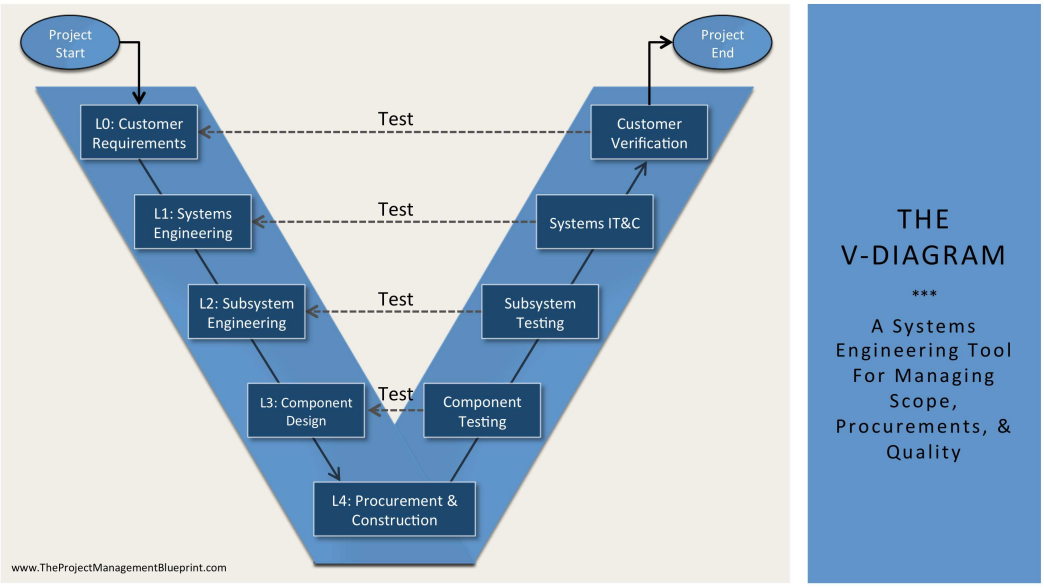
Validace je ověření správnosti výsledného řešení. Jedná se o poslední kontrolu v rámci procesu vývoje (viz obrázek 2.1). Probíhá na hotovém odladěném řešení nasazeném do podmínek ostrého provozu.

Proces vývoje se může pro různé projekty lišit. Pro vývoj řízení bezpečnostně kritických systémů však bývá preferován vodopádový model, který bývá často zobrazován pomocí takzvaného V-diagramu (viz obrázky 2.1 a 2.2). Tento model vývoje se sice nehodí pro rychlý vývoj, hodí se však pro práci se systémy, u kterých je vyžadována vyšší míra zajištěné bezpečnosti. Tento model vývoje totiž nastavuje v procesu vývoje pevné hranice a definuje jasné přechody mezi jednotlivými fázemi.

Na obrázku 2.2 je zobrazeno možné použití vodopádového modelu v procesu vývoje. Na začátku vývoje jsou specifikovány požadavky na systém (L0). V dalším kroku je vytvořen návrh, který tyto požadavky splňuje (L1). V dalších krocích se návrh dělí do menších částí (návrh části systému, komponenty, funkce), které jsou poté navrženy podrobněji (L2, L3). Během implementace návrhu (L4) jsou jednotlivé funkce implementovány. Tyto funkce se následně sdružují do komponent a integrují do systému. Po fázi implementace přichází na řadu fáze testování, která při ověřování postupuje opačně než fáze návrhu. Testuje se, zda byla daná funkce, komponenta nebo část systému vytvořena skutečně tak, jak byla navržena v příslušné úrovni ve fázi návrhu (L3, L2). V závěru procesu vývoje je ověřováno, jestli výsledný systém odpovídá původnímu návrhu (L1) a zda vyhovuje zadaným požadavkům (L0). Přerušované čáry znázorňují testování implementace oproti jednotlivým úrovním návrhu. Pokud implementované řešení nevyhovuje požadavkům daného návrhu, vrací se vývoj zpět do fáze, v níž došlo k zavedení chyby do systému.



Obrázek 2.1: V-diagram pro vývoj software podle normy IEC 60880



Obrázek 2.2: V-diagram s naznačeným účelem testování

2.4 Metody verifikace

K verifikaci lze využít některé z velkého výběru nástrojů, z nichž každý se hodí pro jinou fázi procesu vývoje a pro prokázání jiných skutečností. Nejčastějšími nástroji verifikace jsou dynamická a statická analýza systému a kontrola zavedených pravidel pro různé fáze procesu vývoje (návrh, implementace, testování, nasazení).

2.4.1 Dynamická analýza

Dynamická analýza je metoda verifikace, která se používá spolu s funkčním systémem nebo jeho simulací. Dynamická analýza používá pro ověření požadavků aktivní systém, který za běhu ovlivňuje tak, aby v něm vyvolala ověřované stavy. Dosažené výsledky se poté porovnávají s požadovanými výsledky a vyhodnocují. Nejčastěji používanou formou dynamické analýzy je testování [11].

Účinnost dynamické analýzy se mění podle druhu ověřovaného systému, podle nástrojů, které byly použity při implementaci a podle důrazu, který byl kladen na bezpečnost a na testování. Hlavní problém testování spočívá v tom, že mohou být ověřeny pouze scénáře, které někdo vymyslel, nebo které vyplynuly ze způsobu použití daného systému (například při testování uživatelem). Není však možné zaručeně ověřit celý systém. Ve výsledku se tedy vytváří takové scénáře, které mají za úkol ověřit co největší část chování daného systému. Pro tyto účely se používají různé přístupy k vytváření testů.

Jedno z hlavních rozdělení testů (na white-box a black-box testy) vyplývá ze způsobu, jakým byly navrženy testované scénáře. Při tvorbě white-box testů má jejich tvůrce k dispozici poznatky o testovaném řešení a dokáže tak vytvořit testy, které ověřují potenciálně nebezpečné části. Při tvorbě black-box testů má jejich tvůrce pouze znalost návrhu a požadavků na danou komponentu, a může se tak více soustředit na ověření původních požadavků.

Vykonání testů může být provedeno lidmi nebo může být prováděno automaticky. Často se tyto dva způsoby kombinují. Pro rozhraní funkcí, komponent a částí systému bývá vytvářen seznam automaticky prováděných testů, které musí být splněny po každé změně tohoto systému, a zároveň jsou vytvářeny scénáře pro manuální testy, které ověřují funkci celého systému.

2.4.2 Statická analýza

Statická analýza je metoda verifikace, která ke svému použití nepotřebuje funkční systém. Metody statické analýzy umožňují kromě kontroly chování hotového systému kontrolovat i způsob, jakým byl tento systém implementován. Mezi takovéto metody patří například technická přezkoumání (technical review), kdy starší inženýr kontroluje práci mladšího inženýra a poskytuje mu zpětnou vazbu o kvalitě jeho práce a nalezených odchylkách od přijatého standardu. Mezi statickou analýzu patří i vytváření různých metrik, které lze na implementovaném systému změřit. Tyto metriky lze pak dát do spojitosti s kvalitou a bezpečností vytvořeného řešení.

Další metodou je kontrola použití předepsaných postupů během procesu vývoje. V softwarovém inženýrství se například vedle sémantiky kódu kontroluje i

jeho syntaxe. Analýza kódu má zjistit, zda programátoři nepoužili zakázané konstrukce, které mohou být nebezpečné, neefektivní nebo nepřehledné. Sjednocení stylu psaní kódu eliminuje možnost špatné interpretace funkce kódu a přehlédnutí závadných částí. Ačkoliv se tyto požadavky do určité míry vzájemně vylučují, kvalitní kód by měl být důvěryhodný, efektivní, zabezpečený, přehledný, upravitelný a rozumně malý [12]. Jedním z hotových standardů je MISRA C [7] [6], která stanovuje bezpečné použití jazyka C při programování software pro bezpečnostně kritický systém. Zároveň existují nástroje, kterými lze ověřovat, zda kód tento standard splňuje.

Mezi metody statické analýzy patří i formální metody, které ke své funkci využívají model systému a formalizované požadavky na tento systém. Těmito metodami lze ověřit a dokázat, zda jsou na daném modelu splněny zadané požadavky.

2.4.3 Formální metody

Formální metody jsou matematicky založené metody verifikace používané během fáze specifikace, implementace a testování. Jejich cílem je pomocí matematických nástrojů zajistit robustnost výsledného řešení. Výhodou využití matematické analýzy je získání matematického důkazu, který zajišťuje absolutní platnost dokázaných předpokladů. Získání takového důkazu je však velmi náročné. Proto se tyto metody při vývoji standardních systémů nepoužívají a u velkých bezpečnostně kritických systémů jsou tyto metody používány pouze u kritických částí daného systému. Formální metody lze klasifikovat podle způsobu jejich použití [13]:

- Formální specifikace
- Formální vývoj a verifikace
- Prověření teorií

Při využití formální specifikace se pomocí formálních metod ověřuje, zda daný návrh odpovídá zadaným požadavkům. Pro tento předpoklad se následně hledá matematický důkaz. Tato varianta je nejméně náročným použitím formálních metod. Poznatky získané pomocí formální specifikace jsou v dalším procesu vývoje použity pouze pro informativní účely.

Při vyžití formálního vývoje a verifikace se formální metody používají i v rámci implementace a testování výsledného systému. V prvním případě je nejdříve získán důkaz o správnosti navrženého modelu a poté je přímo z návrhu modelu generována jeho implementace. Druhou možností je opačný postup, kdy je z hotového systému vytvořen zjednodušený, snadno ověřitelný model. V obou případech je nutné k převodu mezi jednotlivými fázemi využívat pouze transformace zachovávající ekvivalenci.

Posledním a nejnáročnějším využitím formální analýzy je prověření teorií. V tomto případě je nutné vytvořit plnohodnotný analytický důkaz dané teorie. Tento způsob se vyplatí používat pouze v tom případě, kdy by cena neodhalené chyby byla velmi vysoká (např. u kritických částí mikroprocesorů).

Kvůli vysokým nákladům na lidskou práci a pomalému postupu při analytickém vytváření důkazů, byly postupně vytvořeny automatické metody formální

analýzy, které umožňují provádět větší množství důkazů v kratším čase. Mezi automatické formální metody patří automatizované dokazování teorií, abstraktní interpretace a Model checking.

Při automatickém dokazování teorií se vytváří důkaz z popisu systému, pravidel chování a možností interakcí. Abstraktní interpretací je možné z hotového systému pomocí rozumné aproximace (sound aproximation) extrahovat informace o možnostech použití daného systému i bez provádění všech možných scénářů. Metoda Model checking ověřuje, zda se hodnoty zvolených parametrů systému chovají v souladu s požadavky na systém. Důkaz je poskytnut vyčerpávající kontrolou všech možných stavů systému.

3. Metoda Model checking

Formální metody jsou relativně novou možností pro prokázání bezpečnosti zkoumaného systému. Vznikly už v 80. letech, kdy jim jejich tvůrci předvídali široké nasazení v rámci vývoje bezpečnostně kritických aplikací. V praxi však tyto metody nebyly kvůli náročnosti a vysoké ceně použití využívány ani zdaleka tak často. Důkaz musel pomocí matematické analýzy vytvářet najatý matematik, vytvoření takového důkazu bylo zdlouhavé a drahé. Proto byly tyto metody používány pouze pro kontrolu těch nejkritičtějších částí ověřovaného systému. Pro lepší využití formálních metod v praxi musely být vytvořeny automatické metody, kterých bylo možné využít až díky většímu výkonu moderních počítačů [14].

Metoda Model checking je automatickou formální metodou statické analýzy. Používá se k vytvoření důkazu o tom, že daný model odpovídá zadaným požadavkům. To v praxi znamená, že neexistuje žádný stav systému, v němž by byly tyto požadavky porušeny.

Výhodou této metody je právě automatizace, která umožňuje ověřovat splnění požadavků při každé změně modelu. Tím lze postupně vytvářet návrh, který odpovídá zadaným požadavkům. Pokud je při použití metody zjištěno, že existuje stav modelu, který porušuje některý požadavek, je zároveň poskytnut protipříklad, kterým byla daná podmínka porušena. Návrhář může tento protipříklad prostudovat a snadněji tak najít a opravit zdroj chyby. Díky této automatizaci lze využít formální metody i v aplikacích, pro které by jejich použití bylo neekonomické, a lze tak zvyšovat bezpečnost těchto aplikací. Možnosti této metody ovšem stále nejsou neomezené a je nutné ji použít ve správné míře a správném kontextu.

3.1 Principy

Použití této metody probíhá tak, že jsou nejdříve stanoveny požadavky na model v lidsky srozumitelném jazyce. Tyto požadavky jsou poté převedeny do některého formálního jazyka (například temporální logiky). Dále je navržen model, který by měl odpovídat zadaným požadavkům. Tento model se spolu se zadanými formálními předá nástroji pro kontrolu modelu, který vyčerpávajícím způsobem ověří, zda neexistuje stav, který porušuje zadané požadavky. Při nálezu takového stavu poskytne tento nástroj informace o tom, jakým způsobem se do daného stavu dostal. Tento protipříklad lze dále použít pro nalezení zdroje chyby a pro její rychlejší opravu.

Princip vyčerpávajícího ověřování modelu spočívá v tom, že počítač rozvine všechny možné počáteční stavy pomocí povolených přechodů do všech možných stavů, které mohou v systému nastat. Počítač při tom uplatňuje vnitřní pravidla kontrolovaného modelu a zároveň prověřuje všechny možnosti, které těmito pravidly omezeny nejsou (reakce na náhodné jevy, na okolní podmínky, na nemodelovanou dynamiku systému nebo na interagující systémy). Tímto způsobem lze tedy model systému rozdělit na deterministickou a nedeterministickou část, kdy počítač ověřuje obě části zároveň.

Příklad použití deterministické proměnné je proměnná `state` v příkladu A.1, která má v sekci ASSIGN stanoven počáteční stav a jednoznačný přechod do dal-

ších stavů. Pokud by u této proměnné nebyl stanoven počáteční stav a jednoznačný přechod, stala by se nedeterministickou proměnnou, která by mohla v každém kroku nabývat hodnoty TRUE i hodnoty FALSE.

Pro snazší získání výsledků je doporučeno modelovat dynamiku systému co nejjednodušeji a zároveň se snažit minimalizovat nedeterministickou část modelu. Příliš velká nedeterministická část exponenciálně zvětšuje velikost stavového prostoru, který musí počítač prověřit. Vetším problémem však bývá složitost dynamiky celého systému, kdy je počítač nucen prověřovat mnohem větší časové úseky, což jeho práci zpomaluje. Nejproblémovější částí systému jsou proměnné s celočíselnými a reálnými hodnotami (teplota v K, rychlost otáček v m/s, uplynulá doba od startu programu v ms), které mohou nabývat obrovského množství stavů. Jejich použití prakticky znemožňuje využití metody Model checking. Proto se namísto jednotlivých hodnot kontrolují možné následky, které tyto hodnoty můžou mít.

Při ověřování modelu s celočíselnými či reálnými proměnnými je třeba převést model systému do modelu rozhodnutí. Model rozhodnutí je takový model, který namísto modelování hodnot jednotlivých parametrů modeluje různé výsledky kritických rozhodnutí (podmíněných přechodů do jiných stavů systému). Tato rozhodnutí a jejich následky lze dohledat z podmíněných akcí systému, kdy například pro tři rozmezí hodnot dané proměnné existují tři různé akce, které systém vykoná v dalším kroku. Model rozhodnutí tedy namísto proměnné, která může nabývat obrovského množství hodnot, zkoumá pouze tři možné alternativy, které mohou v systému nastat. Při převodu modelu systému do modelu rozhodnutí je třeba zachovávat pravidla, kterým je systém podřízen, a vazby mezi jednotlivými hodnotami. Jedna proměnná může být například ve dvou podmínkách vyhodnocena pro dvě různá pásma hodnot, přitom ale nemusí být v modelu rozhodnutí ověřovány čtyři možnosti, ale pouze tři (jedna možnost není proveditelná).

K zápisu požadavků na systém se používají různé formální jazyky. Mezi ně patří například jazyky temporální logiky (LTL a CTL), pravděpodobnostní temporální logiky (PLTL a PCTL), spojité stochastické logiky (CSL), specifikace proměnné (PSL) a další. V této práci bylo rozhodnuto o použití jazyků temporální logiky LTL a CTL. Tyto jazyky neobsahují nástroje pro práci s pravděpodobností a je jimi možné specifikovat časově závislé požadavky na systém.

3.2 Temporální logika

Temporální logika je matematická logika, která kromě klasické práce s výroky v neměnném prostředí umožňuje i práci s časově závislými výroky. Jedná se o logiku obohacenou o časové operátory, které jí umožňují vytvářet výroky s časově vázanými událostmi. Pokud je například v určitém kroku splněn požadavek A, musí být další dva kroky splněn požadavek B a až do splnění požadavku C musí být splněn požadavek D. Toto spojení času a logiky se skvěle hodí pro specifikaci požadavků na dynamický systém [1].

Základní logické operátory jsou přejaté z klasické logiky:

- p – atomický výrok
- ϕ – výrok (může být složený z dalších výroků)

- $\top$ (true) – tautologie (platí vždy)
- $\bot$ (false) – kontradikce (neplatí nikdy)
- $\neg\phi$ (not) – negace výroku
- $\phi \wedge \psi$ – konjunkce (musí platit oba výroky)
- $\phi \vee \psi$ – disjunkce (musí platit alespoň jeden výrok)
- $\phi \Rightarrow \psi$ – implikace (při platnosti ϕ musí platit ψ)
- $\phi \Leftrightarrow \psi$ – ekvivalence (oba výroky musí být platné ve stejných případech)

3.2.1 Časové operátory

Rozšířením klasické logiky, které dělá temporální logiku tak zajímavou, jsou časové operátory. Tyto operátory umožňují zapisovat podmínky, které jsou závislé na čase nebo na posloupnosti provedení. Při jejich zápisu se vždy vychází z přítomného stavu.

- $G\phi$ – globally (vždy musí platit ϕ)
- $F\phi$ – future (někdy musí nastat ϕ)
- $X\phi$ – next (v dalším kroku musí nastat ϕ)
- $\phi U \psi$ – until (ϕ musí platit dokud nezačne platit ψ, přičemž ψ musí někdy nastat)
- $\phi W \psi$ – weak until (ϕ musí platit dokud nezačne platit ψ, přičemž ψ nemusí nikdy nastat)
- $\phi R \psi$ – release (ϕ musí platit dokud nezačne platit ψ včetně momentu, kdy začne ψ platit, přičemž ψ nemusí nikdy nastat)

V následujících vysvětlivkách jsou pro přehlednost použity grafy, na kterých jsou zobrazeny příklady použití daných operátorů. Černé příklady vyhovují zadanému výroku, červené ho nesplňují. U příkladů, které využívají více dílčích výroků, je červený ten dílčí výrok, který porušuje celkový ověřovaný výrok. U těchto grafů je zobrazen stav výroku v čase, kde prvek nejvíce vlevo znázorňuje současný stav (krok 0) a každý prvek napravo od něj znázorňuje stav výroku v následujících krocích (čas se v tomto případě bere jako diskrétní). Platný výrok je značen symbolem $\bigotimes$, neplatný výrok je značen symbolem $\bigodot$, symbol $\cdots$ označuje posun o jeden krok do budoucnosti. Pro získání informace o stavu různých výroků v jednom časovém bodě stačí přečíst jednotlivé řádky daného sloupce (stavy všech výroků v jednom okamžiku).

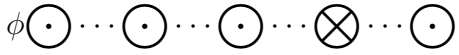
Operátor $\mathbf{G}\phi$ značí, že daný dílčí výrok musí být platný v přítomnosti a v každém následujícím okamžiku. Pro příklad (3.1) je ověřovaný výrok platný.

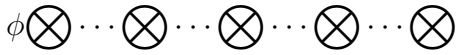
Pro příklad (3.2) ověřovaný výrok platný není. Důvodem je nenaplnění dílčího výroku v kroku 2.


(3.1)


(3.2)

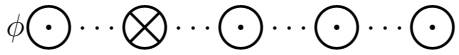
Operátor $\mathbf{F}\phi$ značí, že dílčí výrok ϕ musí platit alespoň jednou. Příklady (3.3) a (3.4) tento výrok splňují. Pro příklad (3.5) není tento výrok platný.


(3.3)


(3.4)


(3.5)

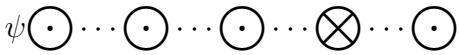
Operátor $\mathbf{X}\phi$ značí posun o jeden krok do budoucnosti. Jak je vidět z příkladů (3.6) a (3.7), výsledek takového výroku záleží pouze na platnosti dílčího výroku ϕ v následujícím kroku.


(3.6)

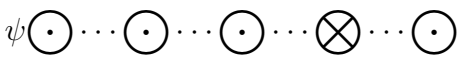

(3.7)

Operátor $\phi\mathbf{U}\psi$ značí, že dílčí výrok ϕ musí být platný až do doby, než bude platný dílčí výrok ψ. Výrok ψ přitom musí být někdy platný. Příklady (3.8) a (3.11) tomuto výroku vyhovují. Příklad (3.9) výroku nevyhovuje kvůli kroku 2, ve kterém přestal platit dílčí výrok ϕ (3.9a). Příklad (3.10) výroku nevyhovuje kvůli tomu, že dílčí požadavek ψ nikdy neplatil (3.10b).


(3.8a)

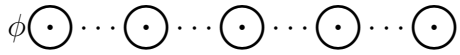

(3.8b)

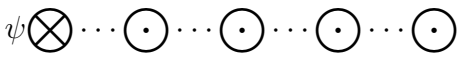

(3.9a)


(3.9b)


(3.10a)

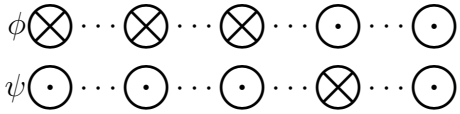

(3.10b)


(3.11a)


(3.11b)

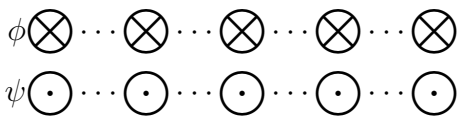
Operátor $\phi\mathbf{W}\psi$ funguje stejně jako operátor $\phi\mathbf{U}\psi$. Jediným rozdílem je to, že dílčí výrok ψ nemusí být nikdy platný. Příklad (3.12) vyhovuje jak operátoru

W tak operátoru **U**, příklad (3.13) vyhovuje pouze operátoru **W** (lze srovnat s příkladem (3.10)).



(3.12a)

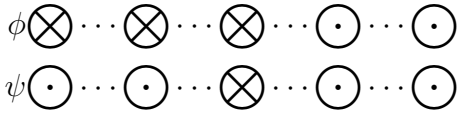
(3.12b)



(3.13a)

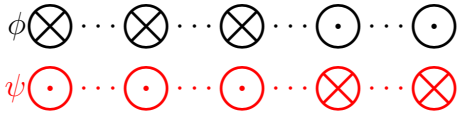
(3.13b)

Operátor $\phi\mathbf{R}\psi$ funguje téměř stejně jako operátor $\phi\mathbf{W}\psi$. Jediným rozdílem je to, že dílčí výrok ϕ musí být platný ještě tom kroku, v němž začne platný dílčí výrok ψ. Na příkladu (3.16) je vidět, že platnost dílčího výroku ψ nemusí nikdy nastat. Příklady (3.14) a (3.15) zobrazují rozdíl mezi operátory **W** a **U** a operátorem **R**. Příklad (3.15) výroku $\phi\mathbf{R}\psi$ nevyhovuje kvůli krokům 2 a 3, kdy ani v jednom případě nejsou platné oba dílčí výroky naráz (3.15b).



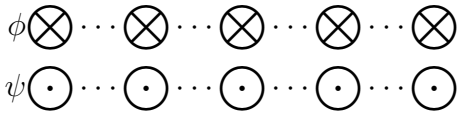
(3.14a)

(3.14b)



(3.15a)

(3.15b)

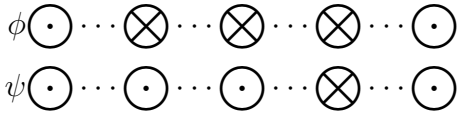


(3.16a)

(3.16b)

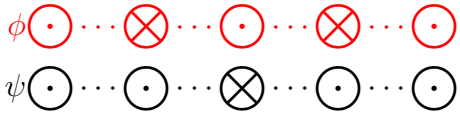
Kombinací těchto operátorů s operátory klasické matematické logiky lze vytvářet složité myšlenkové konstrukce. Pro názornost je uveden jen velmi jednoduchý výrok (3.17), který značí, že pokud kdykoliv začne platit dílčí výrok ϕ, musí platit až do doby, než začne platit dílčí výrok ψ (který někdy začít platit musí). Příklad (3.18) tomuto výroku vyhovuje. Příklad (3.19) danému výroku nevyhovuje neplatný kvůli kroku 4, kdy dílčí výrok ϕ přestal platit zatímco dílčí výrok ψ platit nezačal.

$$G(\phi \Rightarrow (\phi U \psi)) \tag{3.17}$$



(3.18a)

(3.18b)



(3.19a)

(3.19b)

3.2.2 Lineární temporální logika (LTL)

Lineární temporální logika je formální jazyk, kterým lze zapisovat časově lineární výroky. Tento způsob zápisu odpovídá procházení logického stromu do hloubky (kde hloubkou je čas a šířkou jsou alternativní stavy). Při ověřování modelu se postupně prochází jednotlivé časové posloupnosti stavů, kde je u každé posloupnosti ověřeno, zda vyhovuje danému výroku. Pokud existuje jakákoliv časová posloupnost, která výroku nevyhovuje, pak danému výroku nevyhovuje celý systém.

Na rozdíl od logiky výpočetního stromu (CTL) zde nejsou přítomny žádné kvantifikační operátory. Daný výrok musí být splněn ve všech možných stavech. Výsledkem je jednodušší a přehlednější zápis těchto výroků. Příkladem zápisu výroku v lineární temporální logice je příklad (3.20).

$$G((\phi \wedge \psi) \Rightarrow X(\phi U \psi)) \tag{3.20}$$

3.2.3 Logický výpočetní strom (CTL)

Jazyk logického výpočetního stromu je formální jazyk podobný jazyku LTL. Zvláštností tohoto jazyka jsou kvantifikátory temporálních operátorů. Ty umožňují volit počet časových posloupností (všechny nebo alespoň jeden), které musí dané temporální podmínce vyhovovat. Na rozdíl od jazyka LTL, který modeluje požadavky pouze pro jednotlivé časové posloupnosti stavů, umožňuje jazyk CTL pracovat s několika alternativami zároveň. Tento přístup odpovídá procházení logického stromu do šířky (kde hloubkou je čas a šířkou jsou alternativní stavy).

- A (allong all paths) – operace platí pro všechny možnosti
- E (exists) – existuje alespoň jedna možnost, pro kterou operace platí

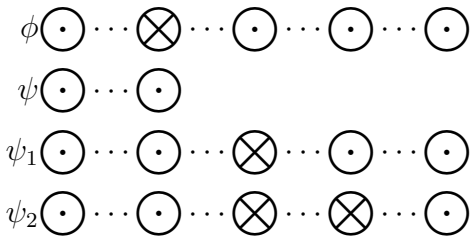
Tyto dva kvantifikátory se používají spolu s temporálními operátory. Značí, zda všechny možné časové posloupnosti musí vyhovovat dané operaci (**A**) nebo zda stačí, aby existovala jedna časová posloupnost, která dané operaci vyhovuje (**E**).

V následujících příkladech je ověřována celá množina možných časových posloupností naráz. Jednotlivé možnosti jsou značeny jako ψ_n. Větvení stromu je znázorněno jako ψ_{nm}, kde jsou větve ψ_{nm} rozvinuty ze stavu ψ_n. Posloupnost, která nevyhovuje zadanému výroku, ale přitom nebrání tomu, aby byl výrok platný, je obarvena oranžově. Pomocí následujících příkladů je ukázán rozdíl platnosti výroku při použití různých kvantifikátorů. Kvantifikován je výrok, který je zapsán v jazyce LTL v podobě (3.21). Ukázány jsou pouze tři ze čtyř možností využití kvantifikátorů u tohoto výroku (3.22), (3.25) a (3.28).

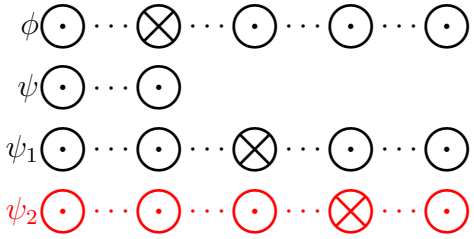
$$G(\phi \Rightarrow X\psi) \tag{3.21}$$

Výrok (3.22) je výrok, který musí být platný za všech okolností. Příklad (3.23) tomuto výroku vyhovuje. Naopak příklad (3.24) zadanému výroku nevyhovuje kvůli nevyhovující možné časové posloupnosti (3.24d).

$$AG(\phi \Rightarrow AX\psi) \tag{3.22}$$



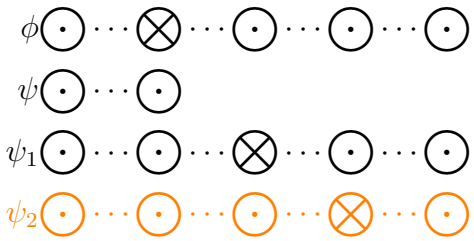
(3.23a)
(3.23b)
(3.23c)
(3.23d)



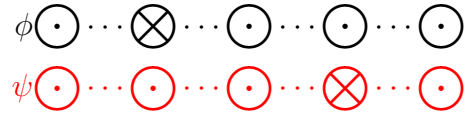
(3.24a)
(3.24b)
(3.24c)
(3.24d)

Výrok (3.25) je výrok, kde musí pro každé splnění ϕ existovat možná cesta, která v následujícím kroku splňuje ψ. V příkladu (3.26) existuje cesta, která danému výroku nevyhovuje (3.26d), přesto však vyhovuje zadanému výroku. Oproti tomu příklad (3.27) danému výroku neexistuje, neexistuje zde totiž ani jedna cesta, která by této temporální operaci vyhovovala.

$$AG(\phi \Rightarrow EX\psi) \tag{3.25}$$



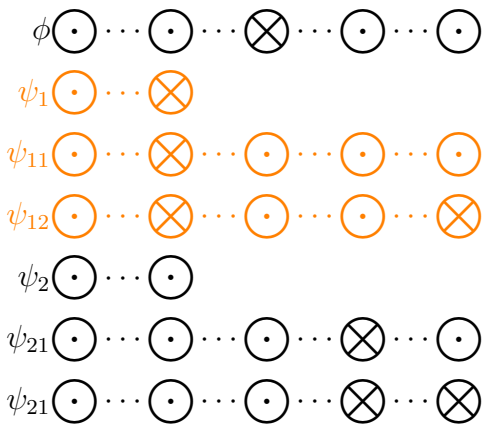
(3.26a)
(3.26b)
(3.26c)
(3.26d)



(3.27a)
(3.27b)

U výroku (3.28) stačí, aby existovala jedna cesta, pro níž je vnitřní podmínka splněna. Zároveň je ale nutné, aby tato vnitřní podmínka byla platná pro všechny cesty, které z tohoto bodu vycházejí. Na příkladu (3.29) je vidět, že vnitřnímu výroku nevyhovuje cesta (3.29d). Oproti tomu všechny cesty, které vycházejí z bodu (3.29e) vnitřnímu výroku vyhovují. Ve výsledku tedy celý tento příklad vyhovuje zadanému výroku.

$$EG(\phi \Rightarrow AX\psi) \tag{3.28}$$



(3.29a) (3.29b) (3.29c) (3.29d) (3.29e) (3.29f) (3.29g)

Je vidět, že jazyk CTL je oproti jazyku LTL méně přehledný. Umožňuje však vytvářet mnohem složitější konstrukce, ve kterých lze vzít v potaz i takové časové posloupnosti stavu, v nichž je časová operace platná pouze pro některou vybranou cestu.

3.3 Použití

Princip použití metody Model checking spočívá tom, že nástroj pro kontrolu modelu prochází stavový strom, který je generován pomocí deterministických pravidel a všech nedeterministických možností. Aby mohl nástroj pro ověření modelu poskytnou vyčerpávající důkaz, musí ověřit všechny možné stavy takto vygenerovaného stromu. Proto musí být nástroj pro ověření modelu výkonný a musí umět prořezávat stavový strom tak, aby nebyl opomenut žádný nebezpečný stav. Druhou možností pro urychlení procesu kontroly je úprava vlastního modelu tak, aby byl minimalizován počet procházených stavů.

Výběr nástroje pro kontrolu modelu přímo ovlivňuje možnosti využití metody Model checking v rámci procesu vývoje. Různé nástroje optimalizují svoji práci pomocí různých algoritmů. Jednou z možností optimalizace je hromadné ověřování několika stavů naráz. Další možností je prořezávání stavového stromu (například pomocí vyhledávání smyček). Jinou možností je stanovení omezení hloubky výpočtů, které limituje rozvinutí stavů systému do určitého časového okamžiku. Tato možnost však nemusí poskytnout důvěryhodný důkaz.

Další možnosti, jak zefektivnit použití metody Model checking, spočívají v modifikaci kontrolovaného modelu. První možností je nalezení dostatečně malého modelu, u kterého může být metoda použita. Druhou možností je převod modelu systému do modelu rozhodnutí, kterým se odstraní nedůležité závislosti a detaily (například opravdové hodnoty celých a reálných čísel). Tímto postupem se lze zbavit nepotřebné dynamiky, která exponenciálně zvětšuje procházený stavový prostor. Poslední možností je zjednodušení modelu, kdy se část dynamiky vypustí a nahradí se několika nedeterministickými stavy.

Pro použití metody Model checking je potřeba mít kromě kontrolovaného modelu i správně zadané podmínky na tento model. Tyto podmínky bývají většinou

zadány ve formě lidské řeči. Z této formy musí být před vlastní kontrolou převedeny do formálního jazyka (v této práci do temporální logiky). Tento převod vyžaduje jak znalost daného formálního jazyka, tak zkušenosti se zápisem formálních požadavků. Při převodu podmínek je třeba zajistit, aby tyto podmínky zůstaly ekvivalentní, jinak hrozí případ, ve kterém je ověřený model prohlášen za vyhovující podmínkám i přes to, že původní nezkomolenou podmínku nesplňuje.

4. Nástroje pro kontrolu modelu

Existuje mnoho nástrojů pro kontrolu modelu [15]. Každý z těchto nástrojů využívá jiný přístup k procházení stavového stromu, jiný způsob zápisu modelu a podporuje jiné formální jazyky pro zápis požadavků na model. Tyto nástroje se liší výkonem, možnostmi a příjemností uživatelského rozhraní.

4.1 Kritéria pro výběr nástroje

Aby bylo možné vybrat správný nástroj pro kontrolu modelu, je třeba nejdříve stanovit požadavky na takový nástroj. Neexistuje žádný univerzální nástroj pro kontrolu modelu, který by umožňoval využít všechny možnosti, které metoda Model checking nabízí. Proto záleží výběr nástroje kromě jeho výkonu i na typu řešené úlohy, složitosti podmínek a na kontextu v jakém má být metoda Model checking používána.

Výkon daného nástroje může být ovlivněn způsobem, jakým je nástroj napsán. Svou roli v tomto případě hraje optimalizace algoritmu programu, optimalizace prohledávání stavového stromu, ale i jazyk, který byl k sepsání programu použit. Mezi optimalizaci prohledávání stavového stromu lze zařadit například detekci smyček nebo ověřování celých částí stavového stromu zároveň.

Další požadavky mohou vyplývat z kontextu, v kterém má být metoda Model checking použita. V tomto případě může výběr omezovat vybraná platforma (Windows, Linux), příjemnost uživatelského rozhraní či možnost automatizace práce s tímto nástrojem.

4.1.1 Typy modelů

Jelikož existují různé typy problémů, existují také různé přístupy k využití metody Model checking. Některé z těchto přístupů lze kombinovat dohromady, zatímco jiné se vzájemně vylučují. Mezi základní přístupy ke kontrole modelu patří:

- Prostý (plain)
- S reálným časem (realtime)
- Časovaný (timed)
- Omezený (bounded)
- Pravděpodobnostní (probabilistic)
- Analýza kódu (code analysis)

Prostý způsob ověření počítá s využitím deterministického stavového modelu s diskrétním časem. U nemodelovaných částí nástroj ověřuje všechny stavy, kterých může model nabývat.

Způsoby ověření, které ověřují model pomocí reálného času (realtime) nebo časování (timed), modelují chování systému za běhu. Rozdíl mezi reálným časem a

časováním je ten, že časovaný model používá diskrétní čas, který je inkrementován v celém modelu naráz, zatímco v reálném čase se model spouští tak, aby byla co nejlépe napodobena skutečná dynamika systému.

Nástroj, který umožňuje využít omezené ověření modelu, se snaží zmenšit stavový strom tím, že určuje maximální dobu, po kterou je model sledován. Tímto opatřením jsou odebrány příliš dlouhé časové posloupnosti stavů a zkracuje se doba nutná pro výpočet důkazu. Důkaz, který byl získán pomocí omezeného ověření modelu, však nelze považovat za důvěryhodný. Způsob, jakým byl vytvořen, totiž nezajišťuje vyčerpávající ověření modelu.

Pravděpodobnostní způsob ověření modelu umožňuje ověřit statistické požadavky na systém. Nástroj používá model se zadanou pravděpodobností výskytu určitých stavů a přechodů. Tyto pravděpodobnosti jsou poté využity k výpočtu pravděpodobností výskytu dalších požadovaných stavů. Pomocí vhodných formálních jazyků lze poté specifikovat požadavky na systém, které na těchto pravděpodobnostech závisí.

Analýza kódu je způsob ověření, který nevyžaduje zápis modelu v určitém jazyce, ale ověřuje přímo zdrojový kód implementovaného řešení. Tato metoda se hodí pro ověření hotového řešení, je naopak velmi nevhodná pro využití při návrhu modelu systému.

4.1.2 Formální jazyky

Zatímco je pro zápis dynamiky modelů vždy používán jazyk daného nástroje, formální jazyky, které se používají pro zápis požadavků na systém, jsou obecně známé a standardizované. Tím je umožněno oddělit specifikaci požadavků na systém od samotného ověřování systému, a díky tomu je možné specifikovat požadavky nezávisle na návrhu systému a na použitém nástroji. Mezi formální jazyky používané pro specifikaci požadavků patří:

- Temporální logika (LTL, CTL, PSL)
- Modální logika (μ-calculus)
- Pravděpodobnostní logika (PLTL, PCTL, CSL)
- Asserce

Základními formálními jazyky jsou jazyky temporální logiky, jako například lineární temporální logika (Linear Temporal Logic – LTL), logika výpočetního stromu (Computation Tree Logic – CTL) a jazyk specifikace proměnných (Property Specification Language – PSL), které zavádějí do logických výroků operátory pro práci s časem. Z těchto jazyků vycházejí různé modifikace, které přinášejí svým uživatelům další možnosti pro specifikaci požadavků.

Jazyk μ-calculus je jazykem modální logiky. Do tohoto jazyka lze převést jakýkoliv výrok zapsaný pomocí temporální logiky. Možnosti tohoto jazyka jsou velmi široké, přehlednost zapsaných výroků je však menší, než u jednodušších jazyků.

Pro práci se stochastickými výroky byly vytvořeny jazyky pravděpodobnostní logiky, jako například pravděpodobnostní lineární temporální logika (Probabilistic LTL – PLTL), pravděpodobnostní logika výpočetního stromu (Probabilistic CTL

– PCTL) a průběhová stochastická logika (Continuous Stochastic Logic – CSL), pomocí kterých lze do výroků zavést i pravděpodobnosti výskytu jednotlivých stavů.

Využití assercí je způsob zajištění bezpečnosti, který často využívají programátoři při vývoji software. Jedná se o podmínky, jejichž nesplnění vede během testování k selhání systému a oznámení chyby programátorovi. Tyto podmínky bývají zabudovány do kódu tak, aby vždy uvědomily programátora o svém selhání. Asserce sice nepatří mezi formální jazyky, umožňují však v rizikových stavech vyhodnotit požadované logické podmínky a zastupují tak funkci formálních jazyků při ověřování požadavků na systém.

4.2 Porovnání nástrojů

O výběru správného nástroje pro kontrolu modelu existuje velké množství článků (například článek [16]), v nichž bývá porovnávána rychlost, možnosti použití a uživatelská příjemnost určitých nástrojů. Většina článků se však shoduje na tom, že nelze vybrat jeden univerzální nástroj, ale že je třeba tento výběr přizpůsobit určitým požadavkům.

V této práci byly zvoleny požadavky na nástroj tak, aby co nejlépe odpovídaly způsobu, jakým má být tento nástroj používán. Vzhledem k typu zadaných modelů (viz kapitolu 5.1 a 5.2) byl požadován nástroj, který používá prosté nebo časované ověření modelu (viz kapitolu 4.1.1) a umí ověřovat požadavky zadané ve formálním jazyku LTL nebo CTL. Ze způsobu použití tohoto nástroje pak vyplývají požadavky na platformu, výkon a licenci. Požadovanou platformou je operační systém Windows a případně operační systém Linux. Pro zvýšení výkonu jsou vybrány pouze nástroje, které byly vytvořeny v "rychlých jazycích"(například C/C++). Nástroj musí být publikován pod volnou licencí, aby bylo možné ho použít alespoň v laboratorních podmínkách. Pro snadné použití je dále vyžadováno, aby nástroj generoval protipříklady při porušení zadaných požadavků a aby ho bylo možné ovládat pomocí příkazů (pro případ automatizace).

Pro výběr vhodného nástroje byl použit seznam dostupných nástrojů pro kontrolu modelu [15]. Podle zadaných kritérií byly vybrány následující nástroje:

- CWB-NC
- Edingburgh CWB
- LTSmin
- MCMAS
- NuSMV
- ProB
- SPIN

Ve výsledku byl pro další práci vybrán nástroj NuSMV, který umožňuje prosté ověření modelu pomocí podmínek zapsaných v jazycích LTL, CTL a PSL. Funguje na operačním systému Windows, Linux i Mac pod volnou licencí. Při nalezení

nesrovnalostí mezi modelem a požadavky vrací protipříklad. Lze ho ovládat pouze pomocí příkazů, GUI chybí. Navíc je tento nástroj pro svůj výkon, jednoduché používání a možnosti použití často doporučován a používán ve vědeckých článcích věnujících se bezpečnostně kritickým systémům a metodě Model checking.

4.3 NuSMV

Nástroj NuSMV je rozšířením nástroje SMV, prvního nástroje pro kontrolu modelu založeného na grafech binárních rozhodnutí (Binary Decision Diagram – BDD). Druhá verze tohoto nástroje (NuSMV 2) je již vyvíjena jako OpenSource a obsahuje navíc kontrolu modelu založenou na řešení Booleovského uspokojení problému (Boolean Satisfability Problem – SAT). [17]

Nástroj NuSMV je ovladatelný pomocí příkazů, ověřovaný model a požadavky jsou definovány v textovém souboru (viz manuál [18]). Jedná se o často používaný a doporučovaný nástroj pro kontrolu modelu bezpečnostně kritických systémů.

Tvůrci nástroje NuSMV vydali v nedávné době nový nástroj NuXMV, který by měl pravděpodobně postupně nahradit stávající nástroj NuSMV. Tento nový nástroj dokáže pracovat s celočíselnými a reálnými proměnnými a sám dokáže převádět model systému na model rozhodnutí (viz manuál [19]). Tento nástroj může být prozkoumán a použit v dalších aplikacích. Pro názornost, dobré reference a přehledný návod k použití byl však nyní pro další práci vybrán nástroj NuSMV.

4.3.1 Ovládání

Ovládání nástroje NuSMV probíhá zadáváním příkazů v příkazové řádce. Při ověření modelu je jedním z parametrů cesta k souboru s modelem a požadavky, dalšími parametry lze modifikovat chování tohoto nástroje. Pomocí nástroje lze spustit automatickou kontrolu nebo interaktivní procházení stavového stromu.

```
NuSMV model.smv
```

NuSMV umožňuje i ohraničenou kontrolu modelu, kterou lze zapnout pomocí argumentu `-bmc`. Tato kontrola ověřuje systém pouze do určitého počtu kroků. Nevýhodou této kontroly je nedůvěryhodnost získaného důkazu.

Interaktivní mód umožňuje vytvořit instanci kontrolovaného modelu a řízeně s ním procházet požadovanými stavy pomocí vybraných operací.

```
NuSMV -int
```

Po skončení kontroly modelu jsou zobrazeny závěry o ověřovaných požadavcích. Výrok, jehož platnost byla dokázána, vypadá následovně:

```
-- specification  G ((SP_NEW & EP) -> SSW_BP)  is true
```

Naopak výrok, který byl během kontroly porušen je zobrazen spolu s protipříkladem:

```
-- specification  G ((SP_NEW & EP) -> SSW_BP)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
  ...
```

4.3.2 Model

Ověřovaný model je zapsán v jazyce SMV. Tento jazyk umožňuje vytvářet vzájemně propojené moduly, které mají definované stavové proměnné (boolean, pole, omezená čísla), u kterých lze definovat jejich chování v čase. Definovány mohou být časově závislé i nezávislé proměnné. U proměnných bez definovaného chování jsou ověřeny všechny hodnoty, kterých můžou tyto proměnné nabývat.

Moduly jsou definovány klíčovým slovem `MODULE`, názvem a vstupními parametry. Těmito parametry reaguje modul na nadřazený modul (okolní prostředí). Nadřazený modul má přitom k dispozici všechny informace o použitém modulu. Hlavní modul má název `main`, neobsahuje žádné vstupní parametry a může obsahovat ověřované požadavky na systém.

Časově nezávislé proměnné jsou definovány v bloku uvozeném klíčovým slovem `DEFINE`. Tyto proměnné jsou vlastně zástupné proměnné pro definované výrazy, které lze použít pro zpřehlednění zápisu. Nelze jim explicitně zadávat hodnotu, jejich hodnota přímo vyplývá z hodnoty výrazu.

Časově závislé proměnné jsou deklarovány v bloku uvozeném klíčovým slovem `VAR`, kde je jim přiřazen typ a možné hodnoty, kterých mohou nabývat. Proměnnou může být i instance jiného modulu.

Časově závislým proměnným lze dále definovat chování pomocí počátečních podmínek a možných přechodů mezi možnými hodnotami. Toto chování systému je definováno v bloku uvozeným klíčovým slovem `ASSIGN`. Počáteční podmínky dané proměnné lze zadat pomocí příkazu `init`, přechod mezi jednotlivými stavy je definován pomocí příkazu `next`.

Požadavky na zadaný model je možné zapsat do modulu `main` pomocí klíčových slov `LTLSPEC` pro specifikaci požadavků v jazyce LTL a `CTLSPEC` pro specifikaci požadavků v jazyce CTL.

Výsledný kód modelu a požadavků může vypadat následovně:

```
MODULE Memory(input1)
VAR
    memory : boolean;
DEFINE
    memoryChanged := (memory != input1);
ASSIGN
    init(memory) := FALSE;
    next(memory) := input1;

MODULE main
VAR
    random1 : boolean;
    memory1 : Memory(random1)
```

```
LTLSPEC G((random1 & X(!random1)) -> X(memory1.memoryChanged))
LTLSPEC G((!random1 & X(random1)) -> X(memory1.memoryChanged))
CTLSPEC AG((random1 & EX(!random1)) -> EX(memory1.memoryChanged))
```

Daný příklad definuje modul `Memory`, který přijímá parametr `input1`, jehož hodnotu ukládá do proměnné `memory`. Součástí tohoto modulu je parametr `memoryChanged`, který je vyhodnocen jako pravdivý (TRUE), pokud hodnota v paměti nesouhlasí s hodnotou předanou na vstupu.

Modul `main` obsahuje proměnnou `random1`, jejíž chování není definováno a která tak může v každém kroku nabývat obou hodnot (TRUE i FALSE). Druhou proměnnou je `memory1`, která je instancí modulu `Memory` s vstupním parametrem `random1`. K proměnným modulu se přistupuje pomocí tečkového zápisu (například `memory1.memoryChanged`).

Poslední částí modelu jsou specifikace požadavků na model. Tyto požadavky kontrolují, zda se po změně hodnoty parametru `random1` nastaví hodnota parametru `memoryChanged` modulu `memory1` na TRUE. V tomto případě je výhodnější zapsat tento požadavek pomocí jazyka LTL.

4.4 SmvBuilder

Ačkoliv jsou možnosti nástroje NuSMV relativně široké, práce s tímto nástrojem není vždy uživatelsky příjemná. Nepříjemným problémem je nemožnost rozdělení modelu do více zdrojových souborů. NuSMV tak sice umožňuje využití separovaných modulů, neumožňuje však umístit jejich definice do jednotlivých souborů. Přitom by tato funkce umožnila opakované využívání hotových a vyzkoušených komponent a zároveň by umožnila přehlednější vytváření navrhovaného modelu.

Různé programy (například Eclipse) nástroj NuSMV integrovaly nebo ho rozšířily o další funkce. Pro účely této práce však bylo jednodušší vytvořit vlastní program, který umožňuje sestavení modelu z více souborů a který bude dále rozšiřován podle aktuálních potřeb. Tento program byl nazván SmvBuilder.

SmvBuilder v první verzi umožňuje automaticky sestavit celistvý model z jednotlivých SMV souborů. Program se spouští z příkazové řádky s parametrem s cestou k hlavnímu SMV souboru, s přepínačem -o a cestou k výstupnímu souboru.

```
SmvBuilder main.svm -o model.smv
NuSMV model.smv
```

Při použití programu SmvBuilder je možné přidat do SMV modelu příkaz `INCLUDE` s cestou ke vkládanému souboru. Tato funkce byla navržena tak, aby fungovala jako zjednodušená funkce INCLUDE používaná v programovacím jazyce C. V první verzi zatím program umožňuje pouze relativní procházení adresáře v němž je soubor umístěn.

Pro příklad je uveden model, který je definován ve dvou souborech. Soubor *test_ rs.smv* je hlavním souborem modelu a obsahuje modul `main`:

```
INCLUDE "Blocks/rs.smv"

MODULE main
VAR
    rs_set      : boolean;
    rs_reset    : boolean;
    rs1         : Rs(rs_set, rs_reset);
```

Soubor *Blocks/rs.smv* poté definuje modul `Rs` pro modelování klopného obvodu. Obsah souboru *Blocks/rs.smv* je vypsán v příloze A.1, funkce modulu `Rs` je popsána v rámci kapitoly 5.1.3.

Program SmvBuilder umožňuje opakovaně využívat obecné a ověřené moduly, které by jinak musely být v každém takovém modelu definovány znovu. To se hodí pro kontrolu modelů, které ve své struktuře opakovaně používají standardizované konstrukce (například funkční bloky).

Jelikož je vývoj tohoto programu plně v kompetenci výzkumu vývoje bezpečnostně kritických systémů na katedře kybernetiky Západočeské Univerzity, je možné tento program dále rozšiřovat podle aktuálních požadavků tohoto výzkumu. Mezi funkce, které by tento nástroj mohl dále implementovat, patří parametrizované spouštění nástroje NuSMV, grafické rozhraní pro použití interaktivního módu a zobrazení protipříkladů. Důvodem pro vytvoření grafického rozhraní je nepřehledný zápis, který NuSMV pro tyto potřeby používá.

5. Příklady aplikací

Pro praktickou aplikaci metody Model checking byly vybrány dva modely: model bezrázového přepínání regulátoru výkonu jaderného reaktoru a model trhací stanice. Oba modely byly získány jako blokové schéma regulátoru vytvořeného v řídicím systému REX. Pro oba modely byly získány požadavky zapsané ve formě lidské řeči, které byly následně přepsány do formálního jazyka. Oba modely byly poté převedeny do jazyka SMV a ověřeny pomocí nástroje NuSMV.

Při ověřování modelů zapsaných pomocí jazyka SMV byl použit nástroj SmvBuilder, který umožnil lépe strukturovat zdrojový kód obou modelů do více dílčích souborů, čímž se zvýšila přehlednost celého modelu a umožnilo se opakované využití některých modulů.

5.1 Bezrázové přepínání regulátoru výkonu jaderného reaktoru

V rámci výzkumu v projektu CANUT byl vytvořen automatický regulátor výkonu výzkumného jaderného reaktoru LVR-15 (viz článek [20]). Tento regulátor má několik dílčích regulátorů, které umožňují regulaci požadované hustoty neutronového toku, relativní rychlosti změny hustoty neutronového toku a tepelného výkonu reaktoru.

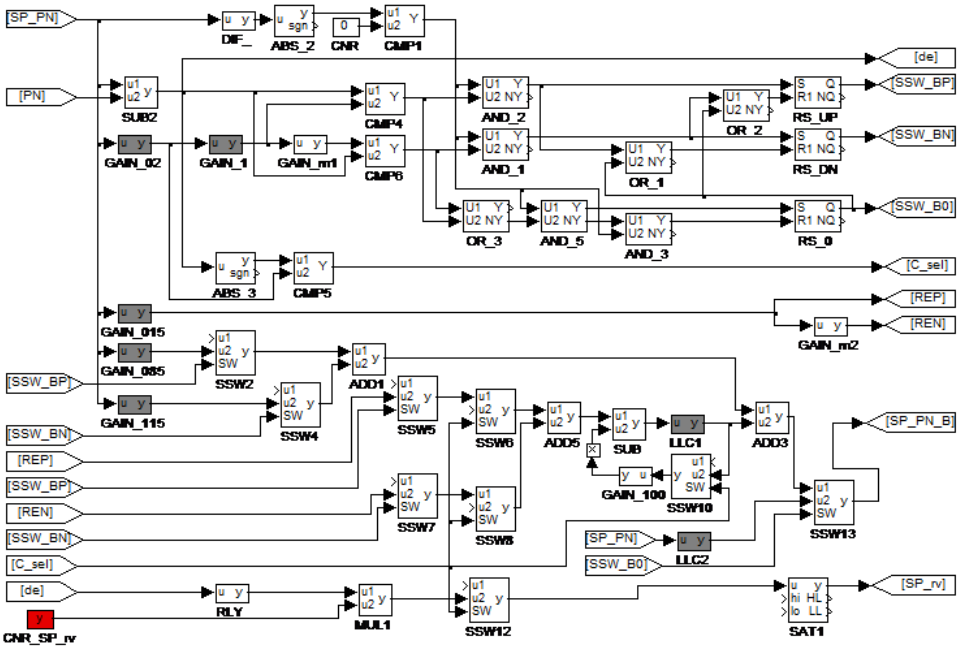
Jednou z vlastností regulátoru je automatický bezrázový přechod mezi regulací relativní rychlosti změny hustoty neutronového toku a regulací hustoty neutronového toku. Tato vlastnost umožňuje spolu s dílčími regulátory najíždět na výkon při změně o několik řádů.

Zatímco jednotlivé regulátory pracují spojitě, model bezrázového přepínání lze částečně přepsat na stavový model, proto byl tento model vybrán k ověření metodou Model checking.

5.1.1 Model

Model celého regulátoru výkonu je popsán v článku [20]. Protože však byl z tohoto modelu ověřen pouze model bezrázového přepínání, nebude zde o celém modelu regulátoru dále pojednáváno.

Celý model bezrázového přepínání je zobrazen na obrázku 5.1. Z tohoto modelu je ovšem pro ověření požadavků podstatná pouze jeho horní polovina, jmenovitě vstupy SP_PN a PN a výstupy SSW_BP, SSW_BN a SSW_B0.



Obrázek 5.1: Model bezrázového přepínání

Parametr SP_PN obsahuje požadovanou hodnotu hustoty neutronového toku. Parametr PN obsahuje aktuální hodnotu hustoty neutronového toku. Výsledek bloku SUB2 je označován jako regulační odchylka E, kde $E = SP\_PN - -PN$. Jako α je označena hranice necitlivosti. Výstup bloku CMP4 je označen jako parametr EP, jehož logická hodnota je udána splněním podmínky $E > \alpha * SP\_PN$. Výstup bloku CMP6 je označen jako parametr EN, jehož logická hodnota je udána splněním podmínky $E < -\alpha * SP\_PN$. Výstup bloku CMP1 je označen jako parametr SP_NEW, který nabývá hodnoty TRUE při změně parametru SP_PN a hodnoty FALSE jindy. Bloky RS_UP, RS_DN a RS_0 jsou klopné obvody s dominantním resetem. Parametry SSW_BP, SSW_BN a SSW_B0 jsou výstupní parametry, které využívá regulátor výkonu ke své další funkci.

5.1.2 Požadavky na model

Aby byla zajištěna nezávislost návrhu modelu a jeho ověření, byly od autora modelu získány požadavky, které by měl tento model splňovat. Tyto požadavky byly zapsány v lidské řeči:

1. Při změně SP_PN, kde je splněno EP, zapnout SSW_BP.
2. Při změně SP_PN, kde je splněno EN, zapnout SSW_BN.
3. Při změně SP_PN, kde není splněno EP ani EN, zapnout SSW_B0.
4. Při změně SP_PN, kde je splněno EP nebo SSW_B0, vypnout SSW_BN.
5. Při změně SP_PN, kde je splněno EN nebo SSW_B0, vypnout SSW_BP.

6. Při změně SP_PN, kde je splněno EP nebo EN, vypnout SSW_B0.

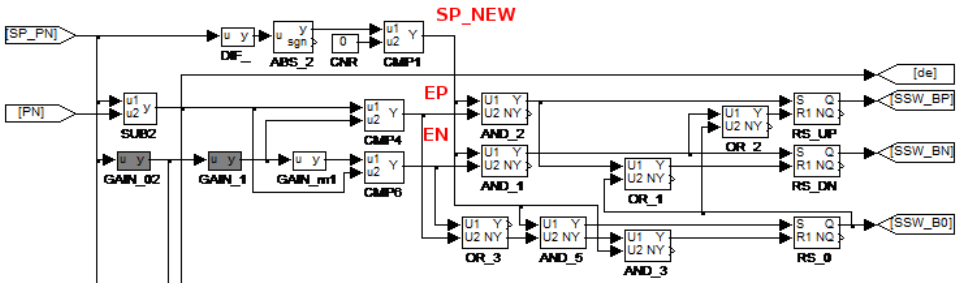
Tyto požadavky byly převedeny do formálního jazyka LTL:

1. G((SP_NEW & EP) -> SSW_BP)
2. G((SP_NEW & EN) -> SSW_BN)
3. G((SP_NEW & !EP & !EN) -> SSW_B0)
4. G((SP_NEW & (EP | lastSSW_B0)) -> !SSW_BN)
5. G((SP_NEW & (EN | lastSSW_B0)) -> !SSW_BP)
6. G((SP_NEW & (EN | EP)) -> !SSW_B0)

Takto definované požadavky byly následně zapsány do souboru s ověřovaným modelem. Ačkoliv tyto podmínky neobsahují téměř žádné logické operátory, díky paměti obsažené v modulu klopného obvodu se jedná o systém s dynamikou, který nemohl být ověřen klasickou logikou.

5.1.3 Model rozhodnutí

Aby bylo možné v tomto případě využít metodu Model checking, bylo nutné převést zadaný model systému na model rozhodnutí. V tomto případě byl převod velmi snadný, stačilo využít již definované logické proměnné SP_NEW, EP a EN, které byly definovány jako výstupy bloků CMP1, CMP4 a CMP6 (viz obrázek 5.1).



Obrázek 5.2: Model rozhodnutí bezrázového přepínání

Aby nebyly zanedbány důležité vazby mezi těmito novými vstupními proměnnými, je třeba zajistit, aby chování těchto proměnných odpovídalo chování daných signálů v reálném systému. V tomto případě je nutné brát v potaz to, že proměnné EP a EN nemohou být pravdivé zároveň a při každé změně parametrů EP a EN je parametr SP_NEW pravdivý. Parametr SP_NEW však může být pravdivý i bez změny hodnot parametrů EP a EN.

První částí, která byla vymodelována pomocí jazyka SMV, byl modul klopného obvodu RS. Tento modul byl vytvořen tak, aby svým chováním odpovídal funkčnímu bloku RS řídicího systému REX (viz manuál [21]). Modul `rs` byl definován v souboru *Blocks/rs.smv*, jehož obsah je vypsán v příloze A.1.

Modul `rs` má, stejně jako jeho předobraz, dva vstupní parametry `s` a `r1`, udržuje si vnitřní paměť `state` a jeho výstupem je parametr `q`, který svým chováním odpovídá parametru Q funkčního bloku RS. Hodnota parametru `q` závisí na aktuálních hodnotách parametrů `s` a `r1` a na hodnotě paměti, která je definována tím, který vstupní parametr byl pravdivý jako poslední. Parametr `r1` je v tomto případě dominantní.

Druhou částí modelu, která byla vymodelována pomocí jazyka SMV, byl celý model rozhodnutí bezrázového přepínání. Tento model byl definován v souboru *reactor.smv* v modulu `main` následovně:

```
INCLUDE "Blocks/rs.smv"

MODULE main
VAR
    --small change of SP_PN
    SP_PN       : boolean;
    --CMP4 and CMP6
    DE          : {ep, e0, en};

    RS_UP       : Rs(RS_UP_S, RS_UP_R1);
    RS_DN       : Rs(RS_DN_S, RS_DN_R1);
    RS_0        : Rs(RS_0_S, RS_0_R1);

    lastSSW_B0  : boolean;
    lastDE      : {ep, e0, en};

DEFINE
    --CMP4
    EP          := (DE = ep);
    --CMP6
    EN          := (DE = en);
    --CMP1
    SP_NEW      := (lastDE != DE) | SP_PN;

    SSW_BP      := RS_UP.q;
    SSW_BN      := RS_DN.q;
    SSW_B0      := RS_0.q;

    RS_UP_S     := SP_NEW & EP;
    RS_UP_R1    := RS_0.q | (SP_NEW & EN);
    RS_DN_S     := SP_NEW & EN;
    RS_DN_R1    := RS_0.q | (SP_NEW & EP);
    RS_0_S      := SP_NEW & (EP | EN);
    RS_0_R1     := !RS_0_S & SP_NEW;

ASSIGN
    init(lastSSW_B0) := FALSE;

    next(lastSSW_B0) := SSW_B0;
```

```
next(lastDE)     := DE;
```

Parametr `DE` byl vytvořen pro to, aby nebyly zanedbány vazby mezi parametry `EP` a `EN`. Tento parametr může nabývat tří stavů (ep, e0, en), tím je zabráněno současnému splnění podmínek `EP` a `EN`. Parametr `SP_PN` v tomto případě označuje takovou změnu původního signálu SP_PN, která nezpůsobí změnu hodnoty parametru `DE`. Parametr `lastSSW_B0` je použit pouze jako paměť pro účely ověření modelu a parametr `lastDE` slouží jako paměť pro vyhodnocení hodnoty parametru `SP_NEW`. V modelu jsou použity tři instance modulu `rs`: `RS_UP`, `RS_DN` a `RS_0`. Ostatní proměnné jsou definovány tak, aby odpovídaly zapojení logických signálů v modelu systému.

5.1.4 Výsledky

Ověřením modelu pomocí metody Model checking bylo zjištěno, že ověřovaný model neodpovídá požadavkům 1, 2, 3 a 6. Pro každý z těchto požadavků byly získány protipříklady, které tyto požadavky porušují. Při hledání zdroje těchto chyb bylo zjištěno, že hledanou příčinou je zacyklení klopného obvodu RS_0 při použití modelu bez velkého skoku na začátku simulace.

Po konzultaci chyby s autorem modelu bylo zjištěno, že se tato chyba během návrhu regulátoru opravdu objevila. Tato chyba se začala objevovat po upravení chování funkčního bloku DIF, který po úpravě přestal generovat pulz při zapnutí systému. V důsledku této změny přestal blok DIF_ v modelu bezrázového přepínání generovat při startu zmíněný pulz a změnil tak chování celého modelu. Zmíněná chyba byla během návrhu regulátoru nalezena a opravena zadáním počátečních podmínek.

V tomto případě tedy metoda Model checking odhalila skrytou vadu, která se během návrhu modelu regulátoru skutečně objevila. Zatímco při návrhu modelu regulátoru byl zdroj této vady nalezen až po několika dnech pátrání, metoda Model checking umožnila jeho nalezení pomocí protipříkladů prakticky ihned.

5.2 Trhací stanice

Dalším systémem, který je možné označit jako bezpečnostně kritický, je trhací stanice ATEGA pro firmu SHAPE Corp. Tato trhací stanice je poloautomatizovaná stanice pro zkoušení vlastností materiálu pomocí tahu.

Provozování této stanice probíhá tak, že obsluha připraví stanici do stavu, kdy je možné do stanice upevnit zkoušený vzorek. Po upevnění vzorku a uzavření ochranného krytu může obsluha spustit automatizovaný test, který napíná testovaný materiál až do bodu přetrhnutí. Výsledky testu jsou poté zaznamenány a uloženy pro další zpracování. Obsluha stanice následně otevře kryt, vyjme obě části testovaného vzorku a pokračuje v testování.

Součástí řídicího systému této stanice je ochranná část, která znemožňuje provádění nebezpečných operací (například spuštění testu bez uzavření krytu).

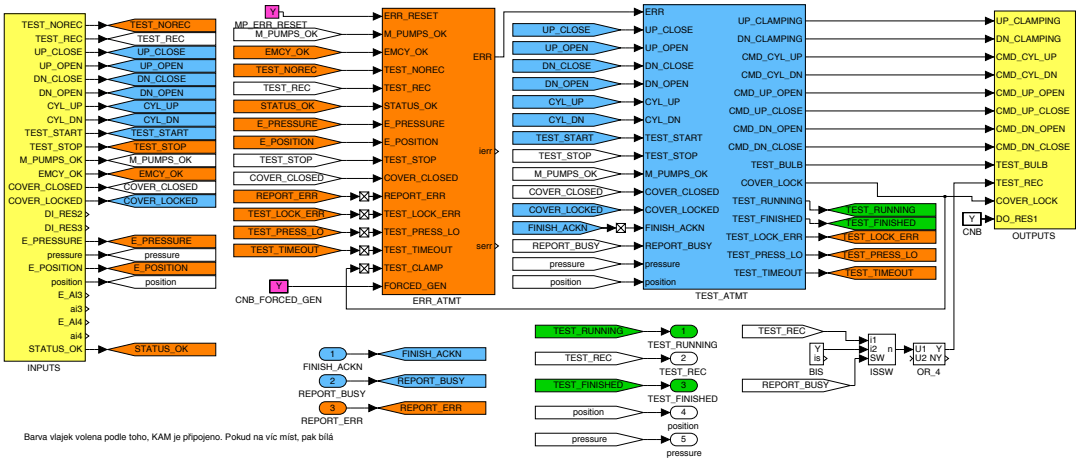
5.2.1 Model

Model řízení trhací stanice ATEGA je z velké části navržen jako stavový model, proto byl vybrán jako ideální model k pokusu, v rámci kterého bude převeden

celý model řízení do modelu SMV při co největším možném zachování ekvivalence těchto modelů. Důvodem pro provedení tohoto experimentu byla potřeba zjistit limity nástroje NuSMV a možnosti automatizace převodu modelu ze schématu řídicího systému REX do modelu SMV.

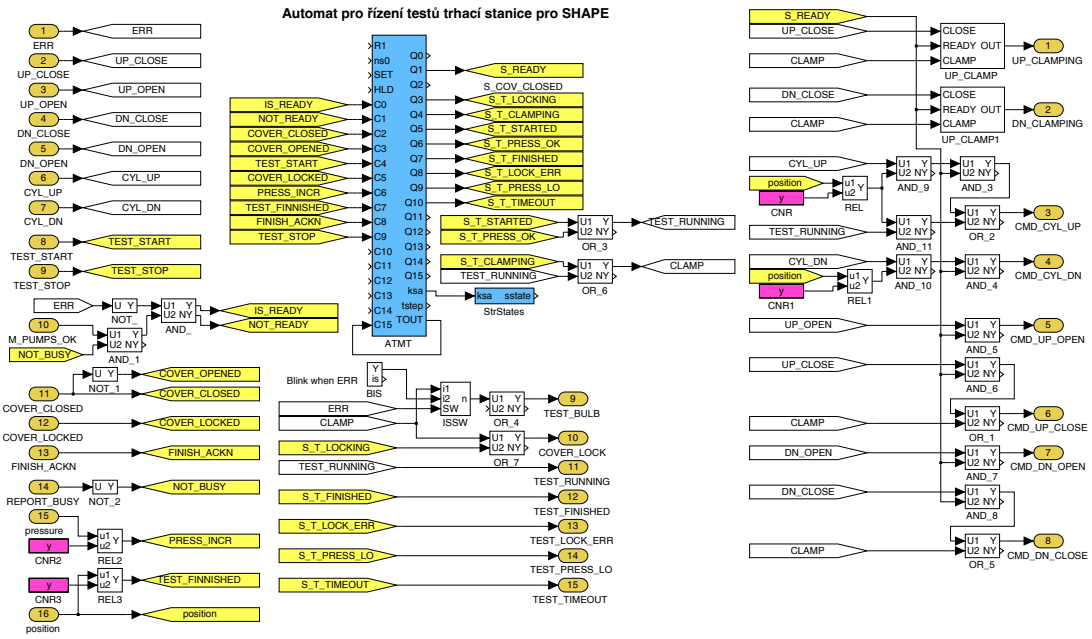
Model řízení trhací stanice ATEGA je oproti modelu bezrázového řízení (viz kapitolu 5.1) velmi rozsáhlý. Skládá se z úlohy `main`, která má za úkol řídit trhací stanici jako takovou, a z úlohy `report`, jejíž úkolem je zaznamenat výsledky testů do nadřazeného systému.

Úloha `main` (viz obrázek 5.3) se skládá ze dvou velkých subsystémů, názorně značených vstupů a výstupů a velkého množství logických proměnných, které jsou značeny velkými písmeny. Toto značení je uplatňováno v celém modelu řízení a značně ulehčuje přepis modelu řízení do modelu SMV.



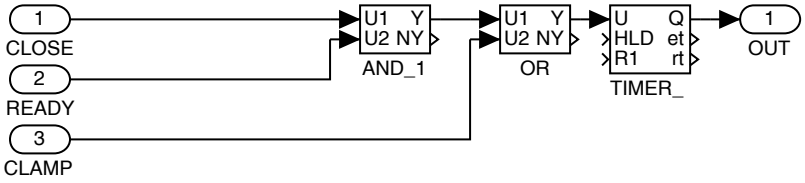
Obrázek 5.3: Hlavní úloha řízení trhací stanice SHAPE

Účelem subsystému `TEST_ATMT` (viz obrázek 5.4) je řídit trhací stanici během testů a reagovat na příkazy obsluhy. Subsystém se skládá ze stavového automatu ATMT, logických operací se signály a subsystémů `UP_CLAMP`.



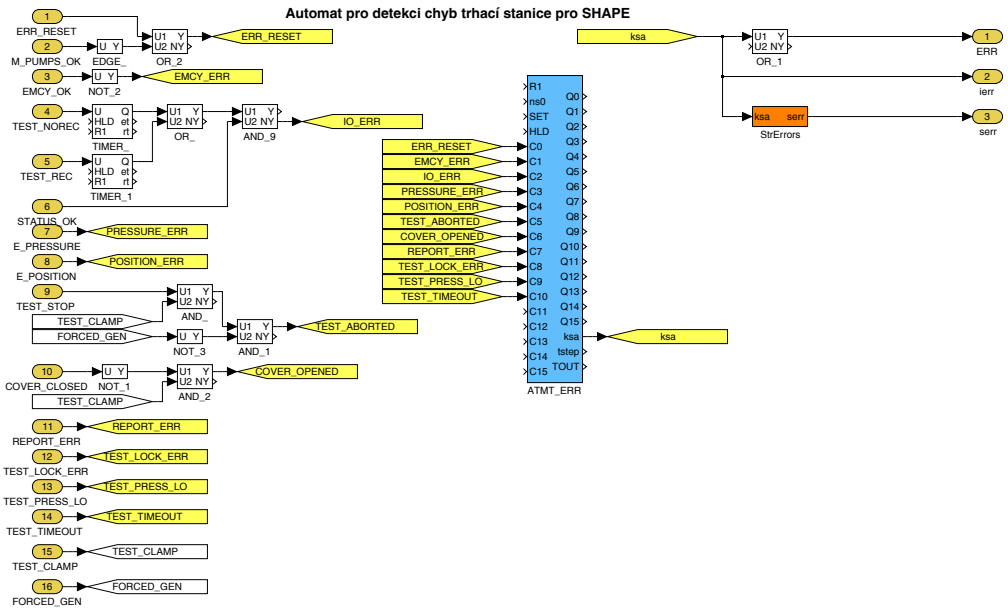
Obrázek 5.4: Automat pro řízení testů

Účelem subsystému `UP_CLAMP` (viz obrázek 5.5) je zajistit správné utažení úchytů testovaného materiálu.



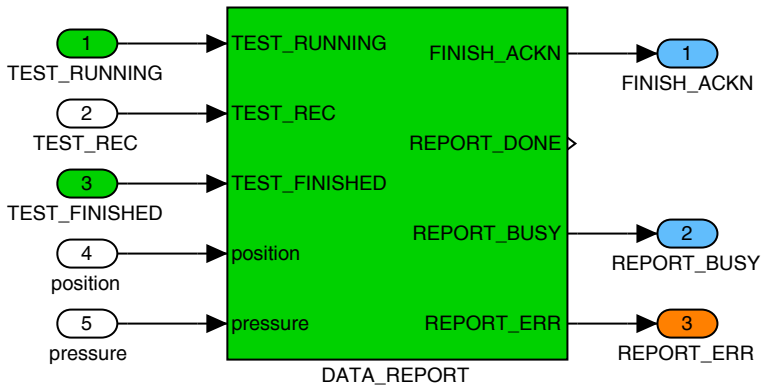
Obrázek 5.5: Subsystém UP_CLAMP a UP_CLAMP1

Účelem subsystému `ERR_ATMT` (viz obrázek 5.6) je detekovat a zpracovávat chybné stavy trhací stanice. Tento subsystém se skládá ze stavového automatu ATMT, který monitoruje signály oznamující výskyt některého chybného stavu a signál pro resetování.

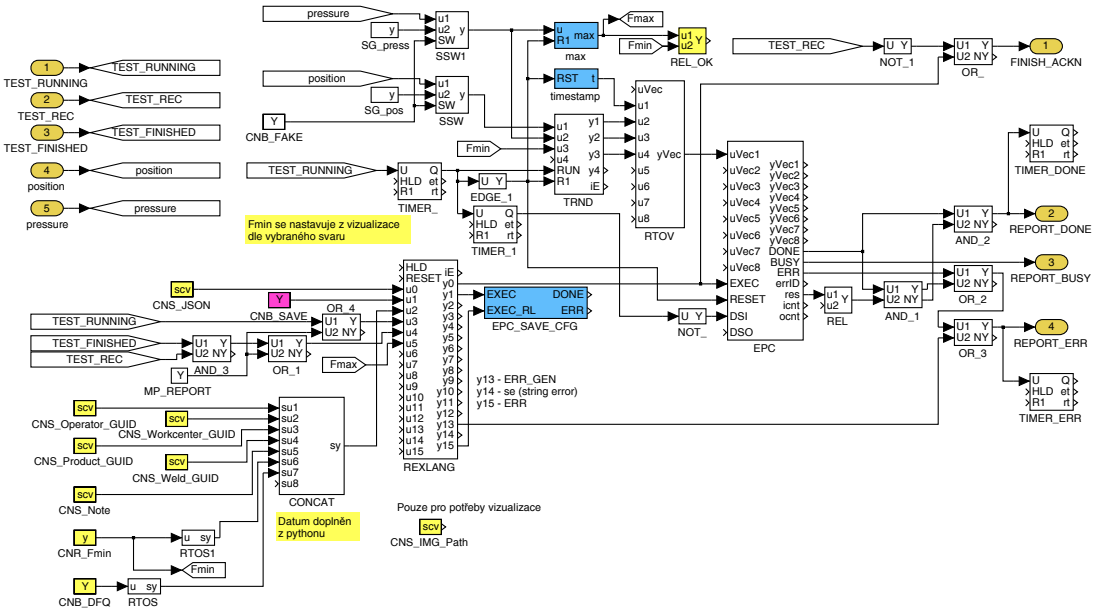


Obrázek 5.6: Automat pro detekci chyb

Účelem úlohy `report` (viz obrázek 5.7) je poskytovat informace o provedených testech nadřazenému systému. Jedinou částí této úlohy je subsystém `DATA_REPORT` (viz obrázek 5.8). Tato úloha bude pro svou složitost a malý vliv na bezpečnost systému v modelu SMV zanedbána.



Obrázek 5.7: Úloha report



Obrázek 5.8: Subsystém pro reportování chyb

5.2.2 Požadavky na model

Pro zadaný model byly pro příklad od autorů získány následující požadavky na ověřovaný systém:

1. Při zamčení krytu (COVER_LOCKED) je odpojeno manuální ovládání horního pístu a nelze poté generovat signál CMD_CYL_UP.
2. Při zamčení krytu (COVER_LOCKED) je odpojeno manuální ovládání dolního pístu a nelze poté generovat signál CMD_CYL_DN.
3. Pokud není kryt uzavřen (COVER_CLOSED) a přitom je spuštěn nový test (TEST_START), musí být zapnut signál ERR.

Tyto požadavky byly převedeny do formálního jazyka LTL:

1. G (coverLock -> X !cmdCylUp);
2. G (coverLock -> X !cmdCylDn);
3. G (!coverClosed & testStart -> err.err);

Takto definované podmínky byly zapsány do souboru s ověřovaným modelem.

5.2.3 Převod modelu systému

Protože byl model řízení trhací stanice ATEGA od počátku z velké části stavovým modelem, nebylo nutné ho pro kontrolu převádět do modelu rozhodnutí. Pro možnosti kontroly byly modelovány pouze logické proměnné, které byly přepsány tak, aby oba modely byly co nejvíce ekvivalentní.

Protože se v tomto případě jedná o rozsáhlý model, byly jeho zdrojové soubory připojeny jako příloha B. Model byl rozdělen do následujících souborů:

- shape.smv
- err_atmt.smv
- clamp.smv
- timer.smv
- test_atmt.smv
- edge.smv

Soubor *shape.smv* je hlavním souborem celého modelu, obsahuje modul `main`, který modeluje stejnojmennou úlohu modelu řízení. Zmíněné soubory *clamp.smv*, *text_atmt.smv* a *err_atmt.smv* obsahují stejnojmenné moduly modelující stejnojmenné subsystémy modelu řízení. V souboru *timer.smv* je definován modul `TimerDelayedOff`, který modeluje funkční bloky TIMER v subsystému `UP_CLAMP` a v subsystému `ERR_ATMT`, a soubor *edge.smv* obsahuje modul `RisingEdge`, který modeluje funkční blok EDGE v subsystému `ERR_ATMT`.

U všech částí daného modelu probíhal převod do modelu SMV tak, aby chování výsledného modelu co nejvíce odpovídalo chování modelu řízení. Výjimkou je subsystém `shape_DATA_REPORT`, který je příliš složitý a zároveň nemá přímý vliv na bezpečnost systému, a proto byl z modelu vypuštěn a nahrazen modulem s neurčitým výstupem.

5.2.4 Výsledky

Kontrolou modelu trhací stanice ATEGA pomocí nástroje SmvBuilder a nástroje NuSMV bylo zjištěno, že přepsaný model nevyhovuje požadavkům 1 a 3. Tuto informaci však nebylo možné ověřit kvůli nepřehlednému zápisu protipříkladů generovaných nástrojem NuSMV. Oba vygenerované protipříklady se skládají zhruba z 17 stavů a jsou zapsány na zhruba 400 řádcích. U modelu s takovým množstvím parametrů je velmi obtížné sledovat vývoj určitých parametrů a zjistit tak zdroj chyby.

Tímto pokusem bylo zjištěno, že metodu Model checking lze spolu s nástrojem NuSMV používat i pro rozsáhlejší modely. Problémem je formát protipříkladů generovaných nástrojem NuSMV, který je nepřehledný a u velkých modelů těžko použitelný. Dále bylo zjištěno, že je možné v určitých případech vytvářet pomocí jednoduchého přepisu SMV modely přímo ze schémat funkčních bloků řídicího systému REX.

6. Problémy využití v praxi

Metoda Model checking je velmi účinná při nalézání vad v modelech systémů, její široké použití je však stále omezeno mnoha faktory, které musí být před jejím nasazením eliminovány. V této kapitole jsou uvedeny některé možnosti výzkumu a vývoje, které by mohly vést ke snazšímu použití této metody v praxi a k postupnému zařazení této metody mezi standardní vývojové nástroje.

6.1 Přehledné zobrazení stavů systému

Jedním z problémů, které byly během praktických pokusů nalezeny, je nepřehledné zobrazení stavů nástrojem NuSMV. Tyto stavy jsou zobrazovány v interaktivním módu a při získání protipříkladů, které porušují některé ze zadaných požadavků.

Aby bylo možné protipříklad využít pro hledání chyby, je nezbytně nutné, aby se v prohledávaných stavech dalo snadno orientovat a aby bylo možné jednoduše vyčíst hodnoty určitého parametru v různých krocích. Proto by bylo vhodné věnovat část dalšího úsilí nalezení nebo vytvoření nástroje, který by umožnil transformovat výstupy z nástroje NuSMV do přehlednějšího grafického zobrazení.

6.2 Automatický převod modelu

Pro použití metody Model checking je potřeba vytvořit stavový model ověřovaného systému. Tento model může být vytvořen jako model rozhodnutí nebo jím může být modelována pouze stavová část ověřovaného systému. Pro snazší použití metody Model checking by mělo být možné generovat tento stavový model z modelu systému automaticky nebo alespoň poloautomaticky.

V kapitole 5.2.1 jsou zmíněna pravidla pro zápis logických signálů. Při dodržování podobného seznamu pravidel by bylo možné vytvořit stavový model přímo ze schématu řídicího algoritmu. Pro zajištění bezpečnosti by dále bylo možné definovat knihovnu standardních modulů, které by svým chováním odpovídaly určitým funkčním blokům a které by byly v takto generovaných modelech používány. Tyto standardní moduly by také musely být ověřeny pomocí metody Model checking.

6.3 Specifikace požadavků v temporální logice

Při využití metody Model checking se objevuje další problém, který je způsoben potřebou vytvářet požadavky ve formálních jazycích. Zatímco u jednoduchých požadavků je převod mezi lidskou řečí a formálním jazykem jednoduchý. U složitějších požadavků je třeba mít odborníka se znalostmi formálních jazyků, který tyto požadavky dokáže specifikovat.

Aby mohl metodu Model checking využívat i méně znalý návrhář, bylo by vhodné vytvořit jakýsi překladač mezi formálním jazykem a lidskou řečí. Tento překladač by pravděpodobně negeneroval z lidské řeči formální požadavky, ale

umožnil by například pomocí vizualizace zpětnou kontrolu navržených požadavků nebo dokonce postupné navrhování celého výroku.

7. Závěr

V této práci byla popsána specifika bezpečnostně kritických systémů, principy metody Model checking a důvody pro její využití během návrhu a kontroly modelů bezpečnostně kritických aplikací. Byly představeny nástroje, které umožňují tuto metodu využívat, a byly otestovány dva modely řízení bezpečnostně kritických systémů.

Bylo zjištěno, že je výhodné využít metodu Model checking pro kontrolu kritických částí řízení, kde lze v navrženém modelu pomocí správně definovaných požadavků nalézt skryté chyby, jejichž původ je poté možné snadno dohledat. Zároveň je za určitých podmínek výhodné využít tuto metodu i pro složitější modely s menšími nároky na bezpečnost.

Byly popsány možné směry dalšího výzkumu a vývoje, které se zabývají zjednodušením použití metody Model checking a zpřístupněním jejího použití širšímu okruhu uživatelů. Toho je možné dosáhnout pomocí automatizace převodu modelu systému do kontrolovaného modelu a názornějším zobrazením průběhů jednotlivých stavů a významu zadaných formálních požadavků.

Na základě dosažených výsledků při použití metody Model checking bylo rozhodnuto o pokračování výzkumu využití této metody při návrhu řízení bezpečnostně kritických systémů. Tento výzkum bude probíhat například v projektu CANUT v rámci aktivity "VaV bezpečnostně kritických ochranných systémů", kde budou zkoumány bezpečnostně kritické systémy řízení z oblasti jaderné energetiky.

Seznam použité literatury

[1] Christel Baier and Joost-Pieter Katoen *Principles of Model Checking* The MIT Press, Cambridge, Massachusetts 2008.

[2] Wikipedia *Safety-critical system* [online] [cit. 1.5.2017].
Dostupné z: `https://en.wikipedia.org/wiki/Safety-critical_system`

[3] Wikipedia *Software development process* [online] [cit. 1.5.2017].
Dostupné z:
`https://en.wikipedia.org/wiki/Software_development_process`

[4] Česká verze mezinárodní normy *ČSN IEC 60880: Jaderné elektrárny - Systémy kontroly a řízení důležité pro bezpečnost - Softwarová hlediska počítačových systémů vykonávajících funkce kategorie A* Český normalizační institut, Praha 2008.

[5] International Organization for Standardization *ISO/TS 16949: The Automotive quality management system* 2016.

[6] MISRA *MISRA C: 2012* [online] [cit. 1.5.2017].
Dostupné z:
`https://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/.aspx`

[7] Wikipedia *MISRA C* [online] [cit. 1.5.2017].
Dostupné z: `https://cs.wikipedia.org/wiki/MISRA_C`

[8] Wikipedia *Requirements analysis* [online] [cit. 1.5.2017].
Dostupné z: `https://en.wikipedia.org/wiki/Requirements_analysis`

[9] Wikipedia *Failure mode and effects analysis* [online] [cit. 1.5.2017].
Dostupné z:
`https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis`

[10] Wikipedia *Fault tolerance* [online] [cit. 1.5.2017].
Dostupné z: `https://en.wikipedia.org/wiki/Fault_tolerance`

[11] Wikipedia *Software testing* [online] [cit. 1.5.2017].
Dostupné z: `https://en.wikipedia.org/wiki/Software_testing`

[12] Wikipedia *Software quality* [online] [cit. 1.5.2017].
Dostupné z: `https://en.wikipedia.org/wiki/Software_quality`

[13] Wikipedia *Formal methods* [online] [cit. 1.5.2017].
Dostupné z: `https://en.wikipedia.org/wiki/Formal_methods`

[14] Ian Sommerville *Softwarové inženýrství* Computer Press, Brno 2013.

[15] Wikipedia *List of model checking tools* [online] [cit. 1.5.2017].
Dostupné z:
`https://en.wikipedia.org/wiki/List_of_model_checking_tools`

[16] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, M. Ouenzar *Comparison of Model Checking Tools for Information Systems* In: Dong J.S., Zhu H. (eds) Formal Methods and Software Engineering. ICFEM 2010. Lecture Notes in Computer Science, vol 6447. Springer, Berlin, Heidelberg.

[17] Wikipedia *NuSMV* [online] [cit. 1.5.2017].
Dostupné z: `https://en.wikipedia.org/wiki/NuSMV`

[18] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev *NuSMV 2.6 User Manual* [online] [cit. 1.5.2017].
Dostupné z: `http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`

[19] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta *nuXmv 1.1.1 User Manual* [online] [cit. 1.5.2017].
Dostupné z:
`https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`

[20] M. Schlegel, J. Königsmarková, P. Balda, J. Sobota, M. Cédl, O. Žáček, V. Juříček, P. Rosík *Regulátor výkonu výzkumného jaderného reaktoru LVR-15* In: Automatizace, regulace a procesy 2014.

[21] REX Controls s.r.o. *Funkční bloky systému REX* verze 2.50.1.

Seznam použitých zkratek

MISRA C: Standard pro psaní v jazyce C

IEC: International Electrotechnical Commission

LTL: Linear Temporal Logic

CTL: Computation Tree Logic

PSL: Property Specification Language

PLTL: Probabilistic LTL

PCTL: Probabilistic CTL

CSL: Continuous Stochastic Logic

SMV: Symbolic model verifier

NuSMV: New SMV

GUI: Graphic User Interface

BDD: Binary Decision Diagram

SAT: Boolean Satisfability Problem

BMC: Bounded Model Checking

LVR-15: Lehkovodní reaktor tankového typu

CANUT: Centre for Advanced Nuclear Technologies

Přílohy

A Moduly modelující obecné funkční bloky

V této sekci jsou vypsány definice obecných modulů modelující funkční bloky řídicího systému REX. Tyto moduly jsou uloženy ve stejnojmenných SMV souborech v adresáři *Blocks*.

A.1 Modul Rs

```
MODULE Rs(s, r1)
VAR
    state : boolean;

DEFINE
    q := (s | state) & !r1;

ASSIGN
    init(state) := FALSE;

    next(state) :=
        case
            r1 : FALSE;
            s : TRUE;
            TRUE : state;
        esac;
```

B Model trhací stanice

V této sekci je vypsán obsah zdrojových SMV souborů modelu trhací stanice ATEGA.

B.1 shape.smv

```
INCLUDE "test_atmt.smv"
INCLUDE "err_atmt.smv"
INCLUDE "data_report.smv"

MODULE main
VAR
    testNorec       : boolean;
    testRec         : boolean;
    upClose         : boolean;
    upOpen          : boolean;
    dnClose         : boolean;
    dnOpen          : boolean;
    cylUp           : boolean;
```

```
    cylDn           : boolean;
    testStart       : boolean;
    testStop        : boolean;
    mPumpsOk        : boolean;
    emcyOk          : boolean;
    coverClosed     : boolean;
    coverLocked     : boolean;
    ePressure       : boolean;
    ePosition       : boolean;
    statusOk        : boolean;
    mpErrReset      : boolean;
    cnbForcedGen    : boolean;
    finishAckn      : boolean;
    reportBusy      : boolean;
    reportErr       : boolean;

    nextFinishAckn  : boolean;
    nextReportErr   : boolean;
    nextTestLockErr : boolean;
    nextTestPressLo : boolean;
    nextTestTimeout : boolean;
    nextTestClamp   : boolean;
    forcedGen       : boolean;

    err             : ErrAtmt(mpErrReset, mPumpsOk, emcyOk,
        testNorec, testRec, statusOk, ePressure, ePosition,
        testStop, coverClosed, nextReportErr, nextTestLockErr,
        nextTestPressLo, nextTestTimeout, nextTestClamp,
        forcedGen);

    test            : TestAtmt(err.err, upClose, upOpen, dnClose,
        dnOpen, cylUp, cylDn, testStart, testStop, mPumpsOk,
        coverClosed, coverLocked, nextFinishAckn, reportBusy);

    report          : dataReport(test.coverLock, testRec,
        test.testFinished);

DEFINE
    upClamping   := test.upClamping;
    dnClamping   := test.dnClamping;
    cmdCylUp     := test.cmdCylUp;
    cmdCylDn     := test.cmdCylDn;
    cmdUpOpen    := test.cmdUpOpen;
    cmdUpClose   := test.cmdUpClose;
    cmdDnOpen    := test.cmdDnOpen;
    cmdDnClose   := test.cmdDnClose;
    testBulb     := test.testBulb;
    coverLock    := test.coverLock;
```

```
    doRes1       := FALSE;
    testRunning  := test.testRunning;
    testFinished := test.testFinished;
    testPressLo  := test.testPressLo;
    testTimeout  := test.testTimeout;
ASSIGN
    next(testRec) :=
        case
            reportBusy  : testRec;
            TRUE        : {TRUE, FALSE};
        esac;

    init(nextFinishAckn)  := FALSE;
    init(nextReportErr)   := FALSE;
    init(nextTestLockErr) := FALSE;
    init(nextTestPressLo) := FALSE;
    init(nextTestTimeout) := FALSE;
    init(nextTestClamp)   := FALSE;

    next(nextFinishAckn)  := finishAckn;
    next(nextReportErr)   := reportErr;
    next(nextTestLockErr) := test.testLockErr;
    next(nextTestPressLo) := test.testPressLo;
    next(nextTestTimeout) := test.testTimeout;
    next(nextTestClamp)   := test.coverLock;
```

B.2 err_atmt.smv

```
INCLUDE "edge.smv"

MODULE ErrAtmt(errResetIn, mPumpsOk, emcyOk, testNorec, testRec,
    statusOk, ePressure, ePosition, testStop, coverClosed,
    reportErr, testLockErr, testPressLo, testTimeout, testClamp,
    forcedGen)

VAR
    state  : {q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10};
    edge   : RisingEdge(mPumpsOk);
    timer  : TimerDelayedOff(testNorec);
    timer1 : TimerDelayedOff(testRec);
DEFINE
    err := !(state = q0);

    c0  := errReset;
    c1  := emcyErr;
    c2  := ioErr;
    c3  := pressureErr;
    c4  := positionErr;
```

```
    c5  := testAborted;
    c6  := coverOpened;
    c7  := reportErr;
    c8  := testLockErr;
    c9  := testPressLo;
    c10 := testTimeout;

    errReset    := errResetIn | edge.out;
    emcyErr     := !emcyOk;
    ioErr       := !(statusOk & (timer.out | timer1.out));
    pressureErr := ePressure;
    positionErr := ePosition;
    testAborted := (testStop & testClamp & !forcedGen);
    coverOpened := !coverClosed & testClamp;

ASSIGN
    next(state) :=
        case
             (state = q0) & c1      : q1;
             (state = q0) & c2      : q2;
             (state = q0) & c3      : q3;
             (state = q0) & c4      : q4;
             (state = q0) & c5      : q5;
             (state = q0) & c6      : q6;
             (state = q0) & c7      : q7;
             (state = q0) & c8      : q8;
             (state = q0) & c9      : q9;
             (state = q0) & c10     : q10;
             (state = q1) & c0      : q0;
             (state = q2) & c0      : q0;
             (state = q3) & c0      : q0;
             (state = q4) & c0      : q0;
             (state = q5) & c0      : q0;
             (state = q6) & c0      : q0;
             (state = q7) & c0      : q0;
             (state = q8) & c0      : q0;
             (state = q9) & c0      : q0;
             (state = q10) & c0     : q0;
             TRUE                   : state;
        esac;
```

B.3 clamp.smv

```
INCLUDE "timer.smv"

MODULE Clamp(close, ready, clamp)
VAR
    timer  : TimerDelayedOff(signal);
```

```
DEFINE
    signal := (close & ready) | clamp;
    out    := timer.out;
```

B.4 timer.smv

```
MODULE TimerDelayedOff(signal)
VAR
    timer : 0..10;

DEFINE
    out := signal | (timer != 10);

ASSIGN
    init(timer) := 0;

    next(timer) :=
        case
            signal      : 0;
            timer = 10  : 10;
            TRUE        : timer + 1;
        esac;
```

B.5 test_atmt.smv

```
INCLUDE "clamp.smv"

MODULE TestAtmt(err, upClose, upOpen, dnClose, dnOpen, cylUp,
    cylDn, testStart, testStop, mPumpsOk, coverClosed,
    coverLocked, finishAckn, reportBusy)

VAR
    state       : {q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10};
    upClamp     : Clamp(upClose, sReady, clamp);
    upClamp1    : Clamp(dnClose, sReady, clamp);
    pressIncr   : boolean;
    bis         : boolean;
    rel         : boolean;
    rel1        : boolean;
    toutInc     : 0..180;

DEFINE
    c0  := isReady;
    c1  := notReady;
    c2  := coverClosed;
    c3  := coverOpened;
    c4  := testStart;
```

```
c5  := coverLocked;
c6  := pressIncr;
c7  := testFinished;
c8  := finishAckn;
c9  := testStop;
c15 := tout;

sReady     := (state = q1);
sTLocking  := (state = q3);
sTClamping := (state = q4);
sTStarted  := (state = q5);
sTPressOk  := (state = q6);
sTFinished := (state = q7);
sTLockErr  := (state = q8);
sTPressLo  := (state = q9);
sTTimeout  := (state = q10);

stateChanged :=
    (
        ((state = q0) & c0) |
        ((state = q1) & (c2 | c1)) |
        ((state = q2) & (c4 | c1 | c3)) |
        ((state = q3) & (c5 | c1 | c15)) |
        ((state = q4) & (c15 | c1)) |
        ((state = q5) & (c1 | c6 | c15)) |
        ((state = q6) & (c7 | c1 | c15 | c9)) |
        ((state = q7) & (c8 | c1)) |
        ((state = q8) & c0) |
        ((state = q9) & c0) |
        ((state = q10) & c0)
    );

tout :=
    (
        (toutInc >= 2 & state = q3) |
        (toutInc >= 10 & ((state = q5) | (state = q6))) |
        (toutInc >= 180 & (state = q7)) |
        (toutInc >= 1 & (state != q3 & state != q5 &
            state != q6 & state != q7))
    );

coverOpened := !coverClosed;
notBusy := !reportBusy;
isReady := (!err & mPumpsOk & notBusy);
notReady := !isReady;

testRunning := sTStarted | sTPressOk;
clamp := sTClamping | testRunning;
```

```
    upClamping   := upClamp.out;
    dnClamping   := upClamp1.out;
    cmdCylUp     := (cylUp & rel & sReady) | (testRunning & rel);
    cmdCylDn     := cylDn & rel1 & sReady;
    cmdUpOpen    := upOpen & sReady;
    cmdUpClose   := (upClose & sReady) | clamp;
    cmdDnOpen    := dnOpen & sReady;
    cmdDnClose   := (dnClose & sReady) | clamp;
    testBulb     := (err & bis) | (!err & clamp);
    coverLock    := clamp | sTLocking;
    testFinished := sTFinished;
    testLockErr  := sTLockErr;
    testPressLo  := sTPressLo;
    testTimeout  := sTTimeout;

ASSIGN
    init(toutInc) := 0;

    next(state) :=
        case
            (state = q0) & c0   : q1;
            (state = q1) & c2   : q2;
            (state = q1) & c1   : q0;
            (state = q2) & c4   : q3;
            (state = q2) & c1   : q0;
            (state = q2) & c3   : q1;
            (state = q3) & c5   : q4;
            (state = q3) & c1   : q0;
            (state = q3) & c15  : q8;
            (state = q4) & c15  : q5;
            (state = q4) & c1   : q0;
            (state = q5) & c1   : q0;
            (state = q5) & c6   : q6;
            (state = q5) & c15  : q9;
            (state = q6) & c7   : q7;
            (state = q6) & c1   : q0;
            (state = q6) & c15  : q10;
            (state = q6) & c9   : q7;
            (state = q7) & c8   : q0;
            (state = q7) & c1   : q0;
            (state = q8) & c0   : q1;
            (state = q9) & c0   : q1;
            (state = q10) & c0  : q1;
            TRUE                : state;
        esac;

    next(toutInc) :=
```

```
        case
            stateChanged                        : 0;
            (toutInc = 1 & !(state = q3))       : 0;
            toutInc = 180                       : toutInc;
            TRUE                                : toutInc + 1;
        esac;
```

B.6 edge.smv

```
MODULE RisingEdge(signal)
VAR
    last : boolean;

DEFINE
    out := (!last & signal);

ASSIGN
    init(last) := FALSE;

    next(last) := signal;
```