Univerzita Tomáše Bati ve Zlíně

Fakulta Aplikované Informatiky

Ing. Michal Bližňák

Disertační práce

Aplikace formálních metod návrhu a tvorby softwarového vybavení na embedded systémy

Studijní obor: Technická kybernetika

Školitel: Prof. Ing. Vladimír Vašek, CSc.

Zlín, Česká Republika, 2007

Poděkování

Rád bych poděkoval svému školiteli Prof. Ing. Vladimíru Vaškovi, CSc. za odborné vedení a cenné rady během celého studia v rámci doktorského studijního programu. Dále bych chtěl poděkovat Doc. Dr. Ing. Dušanu Kolářovi za konzultace, cenné rady a připomínky při praktické realizaci disertační práce.

V neposlední řadě bych chtěl poděkovat svým rodičům a blízkým za jejich shovívavost, podporu a trpělivost, bez níž by bylo dokončení této práce daleko obtížnější.

RESUMÉ

Embedded systémy hrají v současné době důležitou roli v našem každodenním životě. Setkáváme se s nimi ve spotřební elektronice, průmyslových robotech, zdravotnické technice, automobilovém průmyslu, letectví a v mnoha dalších odvětvích.

Navzdory jejich obecnému rozšíření je jejich programování stále daleko méně komfortní, než programování klasických desktopových aplikací. Programátoři musí často detailně znát hardwarové prostředí, pro které aplikace vyvíjí a jimi tvořený kód musí být co nejefektivnější a šetrný k využívání systémových i HW prostředků, které jsou velmi často citelně omezeny. Proto jsou nezřídka využívány pouze nízkoúrovňové programovací jazyky a technologie, což může vést k neefektivnímu procesu tvorby SW a jeho chybovosti.

Cílem této práce je ukázat, že i v oblasti embedded systémů lze úspěšně používat některé moderní metody formálního návrhu a tvorby aplikací k tomu, abychom byli schopni rychle a efektivně vytvářet optimalizovaný, produkční programový kód embedded aplikací.

SUMMARY

Embedded systems are omnipresent and play significant roles in modern-day life. They can be found in consumer electronics, such as digital cameras, DVD players, industrial robots, medical equipment, automotive designs and many other areas.

In contrast to their spread, a programming of embedded systems is a special discipline and demands that embedded systems developers have working knowledge of a multitude of technology areas. Moreover, embedded software applications must be highly optimized due to memory usage aspects and algorithm (or source code) quality, because available system resources can be strongly limited. In many cases, developers are using only low-level programming technologies and languages to meet all requirements of target systems. It is obvious that this way of software development can be time-consuming and inefficient.

The goal of this thesis is to point that some specific formal languages can be effectively used for producing of highly optimized, production-ready source code that fulfills necessary requirements of target embedded systems.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ANSI	American National Standards Institute
API	Application Programming Interface
COM	Component Object Model
CPU	Central Processing Unit
DVD	Digital Video Disk
FSM	Finite State Machine
GPOS	General-Purpose Operating Systém
HAL	Hardware Abstraction Layer
HW	Hardware
I/O	Input / Output
IDE	Integrated Development Environment
KSA	Konečný stavový automat
MCU	Micro Controller Unit
OS	Operating Systém
PC	Personal Computer
PE	Processor Expert
RAD	Rapid Application Development
RAM	Random Access Memory
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SW	Software

UML	Unified Modeling Language
WYSIWYG	What You See Is What You Get
XML	eXtensible Markup Language

Q	Konečná neprázdná množina stavů
Σ	Konečná neprázdná množina vstupních symbolů
O	Konečná neprázdná množina výstupních symbolů
δ	Přechodová funkce
λ	Výstupní funkce
q_0	Počáteční (iniciální) stav
F	Konečná neprázdná množina koncových stavů
Σ^*	Množina všech posloupností abecedy Σ
$\sim$	Relace ekvivalence
w	Slovo (posloupnost) abecedy
δ^*	Rozšířená přechodová funkce

ÚVOD

1 SOUČASNÝ STAV ŘEŠENÉ PROBLEMATIKY

Pod pojmem embedded systémy (často též označované také jako vestavěné, či vestavné systémy), rozumíme systémy s integrovaným výpočetním subsystémem a příslušnými I/O periferiemi. Lze je v současné době nalézt v širokém spektru spotřební elektroniky či průmyslových zařízeních; ať už se jedná o digitální kamery, audio a DVD přehrávače, zdravotnické diagnostické přístroje, řídicí a monitorovací průmyslové technologie, či řídicí systémy v osobní dopravě, letectví, či vojenské technice [1].

Navzdory tomuto, relativně širokému, rozšíření embedded systémů, je jejich programování i v současné době stále mnohem méně komfortní a efektivní, než je tomu např. u programování softwarových aplikací určených pro běžné desktopové počítače. To je způsobeno především faktem, že programátor embedded systémů musí zvládnout nejen samotný programovací jazyk, ale musí se také podobně seznámit s hardwarovou strukturou a možnostmi cílového systému pro který je aplikace vytvářena a jím vytvářený produkt musí efektivně využívat všech prostředků, které mu daná HW platforma poskytuje [1][2]. Zároveň je nutno podotknout, že systémové prostředky embedded systémů jsou ve většině případů značně omezené, zejména pak výkon MCU (Micro Controller Unit; což je obdoba CPU) či velikost operační paměti, a proto je žádoucí vytvářet optimalizované aplikace splňující specifická omezení a podmínky dané cílovou platformou [2].

Aby bylo možno dosáhnout co nejvyšší efektivity využití HW prostředků cílové platformy, je často nutné používat pouze striktně vymezené programovací jazyky (často jen Assembler a ANSI C/C++) a technologie, umožňující přímé řízení interních a externích periferií cílového MCU. Je zřejmé, že tento způsob programování je časově náročný a náchylný na vznik programových chyb; ať již logických, nebo syntaktických. Navíc, jak vyplývá z prostého faktu, že vytvářená aplikace využívá přímo zařízení nabízené cílovou platformou, bude tato aplikace s velkou pravděpodobností nepřenosná na jiné embedded systémy, a to i v případě, že se tyto systémy budou lišit pouze minimálně.

Otázka tedy zní, jakým způsobem by bylo možné zrychlit a zefektivnit vývojový proces aplikací určených (nejen) pro embedded systémy?

Jedna z možných cest spočívá ve využití některé z dostupných metod formálního návrhu a vývoje softwarových aplikací. Tyto moderní vývojové metody zaručují rychlý, intuitivní a bezpečný (z hlediska minimalizace chybovosti zdrojového kódu) návrh aplikací a co je důležité, formální popis aplikační logiky je platformně nezávislý. V současné době jsou tyto, nebo podobné, techniky návrhu SW implementovány v mnoha integrovaných vývojových prostředích podporujících zrychlený vývojový proces aplikací (RAD IDE – Rapid Application Development Integrated Development Environment) pro klasické počítače, ve vývojových prostředích určených pro embedded systémy však prozatím většinou chybí.

Zde je potřeba podotknout, že ne všechny techniky formálního návrhu aplikací jsou vhodné pro použití u embedded systémů. Je tomu tak zejména z důvodu již zmiňovaných nároků na optimalizaci zdrojového kódu aplikace a to zejména z hlediska objemu výsledného kódu a efektivního a šetrného nakládání s dostupnými systémovými prostředky. Dalším omezujícím faktorem je existence (respektive neexistence) vhodných překladačů zdrojových kódů určených pro cílové systémy.

V této práci se proto zaměřím zejména na techniky využitelné společně s **procedurálně orientovanými programovacími jazyky**, jakými jsou např. ANSI C a Pascal. Za tohoto předpokladu lze pro popis aplikační logiky vytvářeného programu použít obecně dobře známých **konečných stavových automatů**/diagramů (Finite State Machines/Charts – FSM), z jejichž struktury lze za použití vhodných nástrojů přímo generovat zdrojový kód vytvářené aplikace. Jak si ukážeme dále, tento aplikační kód může být zapsán pomocí různých programovacích jazyků a navíc lze v rámci zvoleného výstupního jazyka použít rozličné generující algoritmy ovlivňující specifické vlastnosti výsledného zdrojového kódu.

Využití FSM zajišťuje nejen přehledný způsob vývoje programu a jeho deterministické chování, ale navíc umožňuje aplikaci rozličných ověřujících a

optimalizačních algoritmů, které mohou dále zkvalitňovat generovaný kód a potažmo i výslednou softwarovou aplikaci. Důležitým faktem také je, že existuje několik modifikací základních stavových automatů, které umožňují definovat kromě jednoduchých, jednovláknových úloh, také hierarchicky členěné, či paralelně pracující systémy. Těmito modifikacemi mohou být např. stavové automaty definované ve standardu UML [7][11], či Harelovy stavové diagramy [9].

1.1 Vývojové nástroje využívající formální metody návrhu aplikací

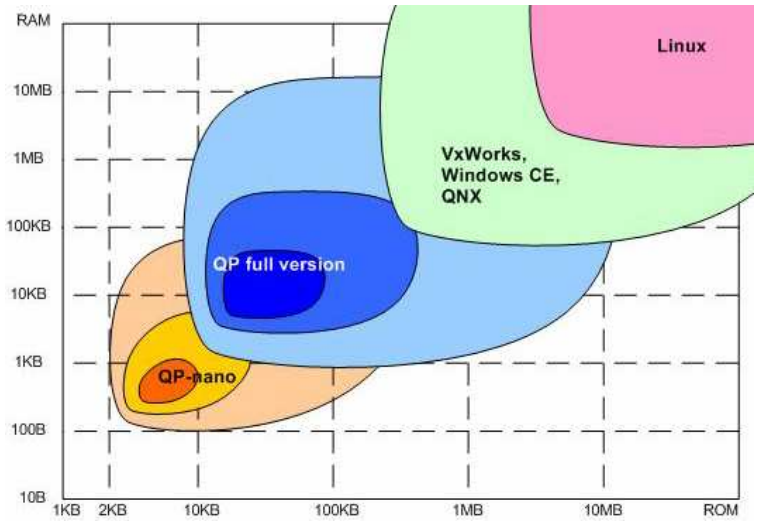
Vývojové nástroje využívající některé ze známých formálních metod návrhu a tvorby softwarových aplikací, které jsou v současné době dostupné jak komerční formou, tak i jako open source projekty, lze v zásadě rozdělit na dvě skupiny:

- nástroje poskytující/generující softwarovou aplikační vrstvu pomocí níž lze vytvářet programy s funkcionalitou založenou na stavových automatech,

- vizuální návrhové nástroje schopné vizualizovat behaviorální a logickou strukturu aplikací, generovat programový kód, či provádět reverzní inženýring již existujícího programového kódu (graficky vizualizovat strukturu programového kódu).

Obě tyto skupiny zahrnují aplikace vhodné jak pro velké, desktopové systémy, tak pro embedded systémy; důležitý je především fakt, s jakými programovacími jazyky mohou tyto nástroje pracovat a jaké paměťové nároky budou aplikace jimi vytvořené mít. Existují nástroje podporující čistě jazyk ANSI C, ale také nástroje umožňující generovat, nebo zpracovávat programový kód vytvořený pomocí celé škály klasických, i moderních programovacích jazyků, jakými jsou např. C/C++, Java, Python, C#, Perl, Ruby, Tcl a podobně.

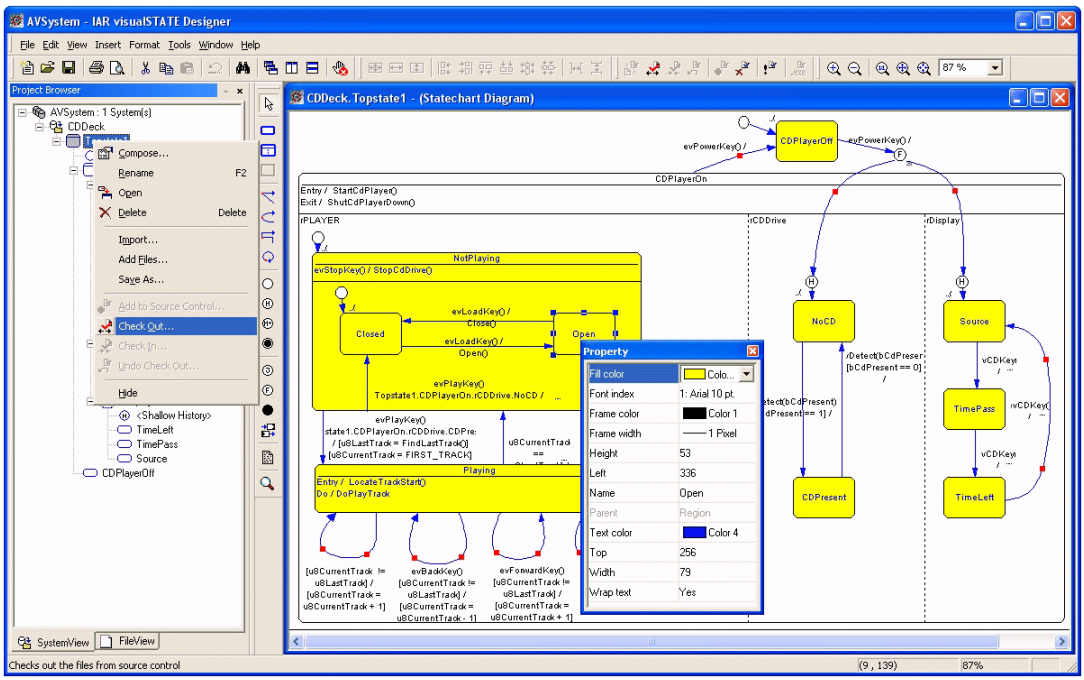
Mezi zástupce první skupiny nástrojů patří například vývojové nástroje a SW knihovny State Map Compiler (SMC) [15], UML StateWizard [16], nebo Quantum Leaps [13].



Obr. 1 Paměťové nároky systému Quantum Leaps [13]

Zejména poslední zmiňovaný nástroj poskytuje velmi silnou podporu pro tvorbu aplikací založených na hierarchických, paralelně pracujících stavových automatech, určených pro běh na embedded systémech a to jak s vlastním RTOS, tak i bez něj. Výhodou tohoto produktu jsou zejména jeho velmi nízké nároky na velikost operační paměti cílového systému, jak je znázorněno v obrázku 1.

Klasickými zástupci druhé zmiňované kategorie jsou například komerční produkty Rational Suite Development Studio Real-Time [22], BetterState [23], Stateflow [24], IAR VisualState [20], Enterprise Architect [19], nebo jejich open source a community-free alternativy ArgoUML [17], Poseidon for UML [18], StarUML a mnohé další. Společným rysem těchto nástrojů je, že poskytují možnost vizuálního návrhu základní **kostry aplikace** pomocí UML schémat a na základě těchto diagramů generovat výstupní programový kód za použití zvoleného (podporovaného) programovacího jazyka.



Obr. 2 Vývojové prostředí IAR VisualState [19]

Je nutné si uvědomit, že se jedná o velmi obecný programový kód (tzv. housekeeping code [10]), který neobsahuje žádný aplikačně, či systémově specifický kód. Veškeré programové fragmenty obsahující systémová volání – zejména pak části kódu představující podmínky (události) a akce přechodů stavových automatů, musí programátor vytvořit ručně na základě znalosti API cílové platformy [10].

1.2 Vývojové nástroje určené pro embedded systémy

Vývojové nástroje speciálně určené pro vývoj aplikací nasazovaných v embedded systémech se od klasických integrovaných vývojových prostředí (dále jen IDE) určených pro desktopové systémy liší zejména silnou podporou nízkoúrovňového programování (použití assembleru, disassembleru, práce s registry MCU, apod.) a možností integrovat překladače a ladicí nástroje určené pro podporovaný programovací jazyk a cílovou platformu. U těchto nástrojů se jen velmi zřídka setkáváme s podporou některých technologií zrychleného vývoje aplikací (tzv. RAD systémy); ve většině případů je programátor nucen využívat klasické programovací postupy, tzn. ručně vytvářet veškerý

programový kód. Mezi tyto nástroje patří zejména vývojová prostředí a překladače dodávané přímo výrobci daného hardware, nebo alternativní open source projekty (dobrým startovním bodem pro jejich hledání je internetový portál Programmers Heaven [38]).

Existují ale také vývojová prostředí, poskytující RAD nástavbu klasických překladačů. Příkladem může být např. produkt české firmy UNIS s názvem ProcessorExpert [20], který využívá komponentní technologii (přístup podobný např. programování v Delphi) pro tvorbu emdedded aplikací, který umožňuje automaticky inicializovat a ovládat interní i externí periferie cílového MCU. Jednotlivé komponenty (v terminologii aplikace ProcessorExpert nazývané Beany), umožňují detailní nastavení vlastností příslušných periferií a poskytují univerzální API pro ovládání funkcionality dané periferie. Tato aplikace je v současné době distribuována samostatně, nebo jako rozšíření známého IDE CodeWarrior [30] a podporuje MCU výrobců Freescale, Fujitsu a National Semiconductors.

1.3 Embedded OS a nástroje pro implementaci HAL

Pro efektivní vývoj embedded aplikací využívajících hardwarových možností cílové platformy je nezbytná existence tzv. **vrstvy hardwarové abstrakce** (anglicky HAL; Hardware Abstraction Layer), která by umožňovala unifikovaný přístup na ovládaný hardware.

U klasických desktopových, nebo mainframeových systémů je touto HAL vrstvou vlastní operační systém nainstalovaný na PC (anglicky GPOS; General-Purpose Operating System). Tento OS se pomocí svých ovladačů sám stará o přímý přístup k hardwarovým periferiím PC a uživateli/programátorovi nabízí standardní API pro vývoj aplikací běžících na daném OS. Ve světě embedded systémů existují také takové OS, ty se však od klasických desktopových OS v jistých ohledech liší. Rozdíl je především v nárocích kladených na spolehlivý a časově deterministický běh emdedded OS, který musí ve většině případů pracovat tzv. v reálném čase, tzn. reakce operačního systému na vnější i vnitřní podněty musí být rychlé, spolehlivé a předvídatelné. Takové OS jsou souhrnně nazývány

Operační systémy pro práci v reálném čase (anglicky RTOS; Real-Time Operating Systems). Společnými prvky obou typů systémů (GPOS a RTOS) jsou např. [1]:

- podpora multitaskingu
- správa softwarových a hardwarových zdrojů
- zprostředkování systémových služeb běžícím aplikacím a samozřejmě
- oddělení hardwarové vrstvy od softwarové.

Na druhou stranu, RTOS by měl vykazovat určité klíčové vlastnosti, které nemusí být v GPOS vždy implementovány. Těmito vlastnostmi jsou:

- lepší spolehlivost
- škálovatelnost OS
- rychlost práce OS
- menší paměťové nároky
- schopnost přepínání procesů (přidělování procesorového času) splňující obecné požadavky na RTOS
- podpora bezdiskových embedded systémů s možností zavádění z ROM a RAM a
- lepší přenositelnost na různé hardwarové platformy.

Známými zástupci těchto RTOS jsou například systémy VxWorks [29], QNX [25], LinuxWorks [26], RT-Linux, Katix [28], Fusion [27] a další.

Bohužel, nasazení těchto RTOS není možné na všech typech embedded systémů a proto bývá někdy nezbytné, využít jiné možnosti zajištění (nebo náhrady) HAL. Touto možností je například použití některého vývojového nástroje vhodného pro inicializaci a ovládání HW periferií a zajištění funkcionality blízké klasickým RTOS (např. již zmiňovaný Processor Expert, či systém Quantum Leaps).

2 VYMEZENÍ CÍLŮ DISERTAČNÍ PRÁCE

Jak vyplývá z úvodu této práce a kapitol zabývajících se rozborem současného stavu problematiky využití formálních metod při vývoji SW aplikací pro embedded systémy, existují sice v současné době kvalitní nástroje pro „vizuální tvorbu“ zmiňovaného programového vybavení, ale všechny mají jednu společnou negativní vlastnost: programový kód jimi generovaný není možné považovat za tzv. **produkční kód**, tzn. kód plně funkční a připravený k okamžitému nasazení. To je způsobeno zejména již zmiňovaným faktem, že neumožňují vytvářet programový kód využívající hardwarových a softwarovým specifik cílové platformy. V některých případech je sice možné využít funkcionality RTOS, na kterém by byly tyto aplikace provozovány, není to však pravidlem.

Cílem této disertační práce je tedy vytvořit „chybějící článek“ takového aplikačního systému, který by vyplňoval současnou mezeru v nabídce vývojových nástrojů určených pro embedded systémy. Mělo by se jednat o aplikaci, která by umožňovala jednoduchou a intuitivní tvorbu vizuálního (formálního) popisu aplikační logiky vytvářené aplikace, a na základě tohoto popisu a s využitím současných dostupných nástrojů pro tvorbu (nahrazení) HAL u embedded systémů, by byla schopna generovat produkční, platformě nezávislý programový kód.

Dílčími cíly, vyplývajícími z postupných kroků nezbytných při vývoji tohoto systému budou:

- Výběr vhodné formální metody popisu aplikační logiky
- Modifikace / vytvoření algoritmů vhodných pro generování programového kódu
- Vytvoření aplikace umožňující grafický návrh formálního popisu aplikační logiky a následné generování programového kódu
- Integrace SW systému vhodného pro zajištění HAL s vytvořenými nástroji určenými pro vizuální návrh aplikace.

2.1 Přínosy pro vědu a praxi

Jak je patrné, hlavním přínosem této práce bude možnost rychlé, intuitivní a přehledné tvorby softwarového vybavení určeného pro embedded systémy, u nichž není z nějakého důvodu možné využít funkcionality RTOS a současných existujících vývojových nástrojů podporujících formální a RAD přístupy k tvorbě softwarových aplikací.

3 POSTUP ŘEŠENÍ

3.1 Formální popis aplikační logiky, stavové automaty

Jako nejvhodnější metoda pro popis aplikační logiky, tedy behaviorální struktury vytvářeného programu, se jeví tzv. **konečné stavové automaty**.

Konečný stavový automat (KSA) je, z našeho pohledu, model chování skládající se ze tří základních stavebních prvků: stavů systému, přechodů mezi těmito stavy a akcemi, které se provádějí, v době přechodu z jednoho stavu systému do stavu jiného. Stavy v podstatě uchovávají informaci o celé minulosti systému, tzn. reflektují jakými změnami systém prošel od počátku jeho činnosti do současnosti [5][12].

3.1.1 Základní terminologie a vlastnosti stavových automatů

Obecně rozlišujeme dva základní typy stavových automatů: **rozpoznávající** (**Acceptors**/**Recognizers**), jejichž výstupem je binární informace říkající, zda je daná posloupnost vstupních znaků přijata daným automatem a existuje na ni adekvátní odpověď a **převodové** (**Transducers**), schopné na základě postupného zpracování vstupních symbolů generovat posloupnost symbolů výstupních. Zde navíc rozlišujeme mezi tzv. **Mealyho** a **Mooreovými** stavovými automaty (výstup Mooreova stavového automatu závisí pouze na aktuálním stavu, kdežto výstup Mealyho automatu závisí na aktuálním stavu a aktuálním vstupu). Matematický model těchto automatů je následující:

Rozpoznávající stavový automat (stavový automat bez výstupu) je pětice $<Q, \Sigma, \delta, q_0, F>$, kde:

- Q je konečná neprázdná množina stavů (stavový prostor)
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- q_0 je počáteční – iniciální stav, $q_0 \in Q$
- δ je přechodová funkce, tedy zobrazení $Q \times \Sigma \rightarrow Q$
- F je množina koncových stavů (cílová množina), $F \subseteq Q$

Matematická definice převodového stavového automatu (stavového automatu s výstupem) je šestice $<Q, \Sigma, O, q_0, \delta, \lambda>$, kde:

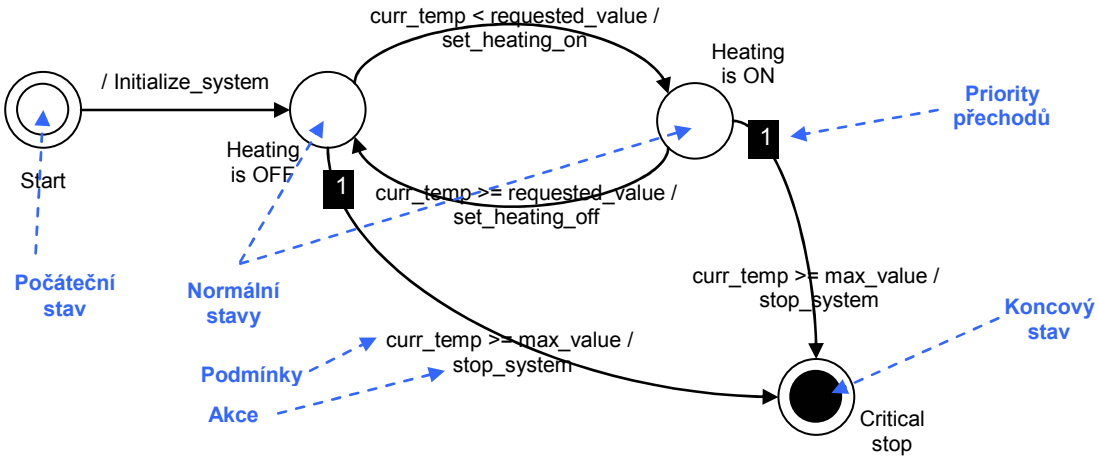
- Q je konečná neprázdná množina stavů (stavový prostor)
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- O je konečná neprázdná množina výstupních symbolů (výstupní abeceda)
- q_0 je počáteční – iniciální stav, $q_0 \in Q$
- δ je přechodová funkce, tedy zobrazení $Q \times \Sigma \rightarrow Q$
- λ je výstupní funkce, tedy zobrazení
    - $\lambda : Q \times \Sigma \rightarrow O$ Mealyho automat
    - $\lambda : Q \rightarrow O$ Mooreův automat

Mealyho i Mooreovy stavové automaty jsou ekvivalentní v tom smyslu, že mohou být bez obav použity pro popis stejných systémů, ale s tím rozdílem, že popis pomocí Mealyho automatu redukuje počet stavů nutných pro popis daného systému a je tudíž z našeho hlediska úspornější. Navíc obousměrný převod mezi Mealyho a Mooreovým stavovým automatem je triviální.

Dalším možným dělením stavových automatů by mohlo být členění na **deterministické** a **nedeterministické**. U deterministických stavových automatů existuje pro jednu uspořádanou dvojici stav-vstup vždy pouze jeden možný přechod, u nedeterministických automatů může existovat více možných přechodů z daného stavu za podmínky jednoho konkrétního vstupu. Rovněž obousměrný převod mezi deterministickým a nedeterministickým FSM popisujícím daný systém je možný [5].

V této práci se budeme dále věnovat již pouze **deterministickým Mealyho stavovým automatům**, protože právě tyto jsou schopny nejlépe popisovat chování většiny základních typů softwarových aplikací. Také počet jednotlivých stavů, nutných pro konstrukci takového automatu, je optimální s ohledem na potřebu přehledného zobrazení daného popisu. Problematika stavových automatů je blíže popsána v [5][6].

Obrázek 3 ilustruje jednu z možných grafických reprezentací stavového automatu, tzv. stavový diagram (state chart). Obrázek představuje popis aplikační logiky programu, který zajišťuje dvoupolohovou regulaci tepelného systému s ověřováním překročení kritické hodnoty teploty.



Obr. 3 Jedna z možných grafických reprezentací FSM

Význam použitých symbolů a celého diagramu je následující: Vstupní bod algoritmu (programu) je definován tzv. **počátečním stavem** (initial state) z něhož program bezprostředně po své spuštění přechází do **stavu** „*Heating is OFF*“. Přechod mezi těmito dvěma stavy není ničím podmíněn (proběhne bezprostředně po startu programu – tedy **pro přijetí** prázdného **symbolu vstupní abecedy**) a vyvolá akci s názvem „*initialize_system*“ (což lze chápat jak **zápis symbolu výstupní abecedy** automatu). Přechod mezi stavy „*Heating is OFF*“ a „*Heating is ON*“ je podmíněn poklesem aktuální teploty pod definovanou žádanou hodnotu (tedy přijmutím příslušného symbolu vstupní abecedy) a při jeho provedení je vyvolána akce s názvem „*set_heating_on*“, jejímž výsledkem je aktivace topného akčního členu. Analogicky, přechod mezi stavy „*Heating_is ON*“ a „*Heating_is_OFF*“ je strážen podmínkou testujících překročení žádané hodnoty řízené teploty a jeho akcí je vypnutí topného akčního členu. Kromě dvou výše zmiňovaných stavů realizujících vlastní dvoupolohový regulační algoritmus pak popisovaná aplikace obsahuje také speciální koncový stav „*Critical stop*“, který může být dosažen tehdy, překročí-li

aktuální hodnota řízené tepelné veličiny předem definovanou kritickou hranici. Vstup do tohoto koncového stavu je zajištěn dvěma přechody vedoucími z obou standardních stavů programu. U těchto přechodů si povšimněte zejména definovaných priorit které zajišťují, že test překročení kritické hodnoty je proveden přednostně před porovnáním žádané a aktuální hodnoty regulované veličiny.

Struktura stavového automatu může být popsána nejen pomocí stavového diagramu, ale také pomocí tzv. **přechodové tabulky**.

	Podmínky / Aktuální stav	N/A	Curr_Temp >= Max_Val	Curr_Temp < Requested_Val	Curr_Temp >= Requested_Val
→	**Start**	Heating_is_OFF	N/A	N/A	N/A
	Heating_is_OFF	N/A	Critical_Stop	Heating_is_ON	Heating_is_OFF
	Heating_is_ON	N/A	Critical_Stop	Heating_is_OFF	Heating_is_ON
←	**Critical_Stop**	N/A	N/A	N/A	N/A

Obr. 4 Přechodová tabulka

3.1.2 Vztah mezi stavovými automaty a aplikační logikou

V kapitole 3.1.1 jsme si popsali základní vlastnosti stavových automatů. Nyní vyvstává otázka, jaká je tedy souvislost mezi stavovými automaty a popisovanou aplikační logikou, respektive zdrojovým programovým kódem dané aplikace.

Jak již bylo zmíněno, alfou a omegou funkcionality stavových automatů je schopnost **postupně** zpracovávat symboly vstupní abecedy a na základě jejich posloupnosti pak přenášet popisovaný systém do jeho dílčích stavů, nebo/a generovat příslušné symboly výstupní. Je důležité si uvědomit, že definice těchto vstupních a výstupních symbolů, náležejících vstupní a výstupní abecedě rozpoznávané daným stavovým automatem, je plně v kompetenci uživatele (tvůrce) stavového automatu a mohou představovat „téměř“ cokoli. Jedinou podmínkou je, aby takto definovaná vstupní a výstupní abeceda splňovala kritéria vyplývající z Nerodovy věty [5], která říká, zda je daná abeceda a jazyk z ní složený rozpoznatelný stavovým automatem. Její obecné znění je následující:

Definice 1: Je-li Σ libovolná konečná množina (abeceda), pak Σ^+ označuje množinu všech konečných neprázdných posloupností utvořených z prvků množiny Σ, *e* označuje prázdnou posloupnost a definujeme $\Sigma^* = \Sigma^+ \cup \{e\}$.

Definice 2: Budiž Σ konečná abeceda a ~ relace ekvivalence na Σ*. Relace ~ se nazývá *pravou kongruencí*, jestliže pro všechna $u, v, w \in \Sigma^*$ ze vztahu $u \sim v$ plyne $uw \sim wv$.

Definice 3: O relaci ekvivalence ~ na množině M říkáme, že je konečného indexu, jestliže rozklad M/~ má konečný počet tříd.

Věta 1 (Nerodova): Nechť *L* je jazyk nad konečnou abecedou Σ. Pak *L* je rozpoznatelný konečným automatem, právě když existuje pravá kongruence konečného indexu taková, že L je sjednocením jistých tříd rozkladu Σ*/~.

Lze také říct, že jazyk rozpoznávaný konečným automatem je takový, pro který platí

$$L(A) = \{w; w \in \Sigma^* \wedge \delta^*(q_0, w) \in F\}. \qquad (1)$$

tzn. jazyk je rozpoznatelný stavovým automatem tehdy, existují-li taková slova daného jazyka, které přenesou stavový automat z jeho počátečního stavu do některého koncového stavu [5].

Lze říct, že každá aplikace, jejíž aplikační logiku budeme popisovat pomocí FSM, se může na určité úrovni programové logiky nalézat pouze v konečném počtu možných aplikačních stavů (a tudíž i počet možných tříd rozkladu je Σ*/~ je konečný). Ve většině případů tedy nebude obtížné nalézt takovou abecedu a jazyk, nad nímž bude možno zkonstruovat stavový automat rozpoznávající tuto abecedu.

Jednotlivými symboly vstupní abecedy mohou být v naší interpretaci například podmínkové výrazy použité v programových příkazech sloužících k řízení toku programu a symboly výstupní abecedy pak mohou být procedury nebo funkce volané při přechodech

mezi jednotlivými stavy (čili akce přechodů - produkty výstupní funkce λ - u Mealyho stavového automatu s výstupem [5]). Další vztahy mezi aplikační logikou a FSM pak mohou být následující:

Každý stav, ve kterém se může popisovaná aplikace nalézat, lze definovat jako jeden stav obsažený ve stavovém automatu. Přechody mezi těmito stavy lze realizovat pomocí klasického řízení toku programu s využitím vhodných programových příkazů (`if`, `switch`, `goto`, …), kde testované logické výrazy jsou implementacemi podmínek strážících jednotlivé přechody stavového automatu. Akce spojené s těmito přechody pak mohou být realizovány programovým kódem volaným za předpokladu splnění definované podmínky. Je zřejmé, že tímto způsobem lze popisovat aplikační logiku na většině úrovních vytvářeného programu – jak chování hlavní funkce programu, tak i dalších dílčích funkcí a procedur. Pro ilustraci zde uveďme zdrojový kód v programovacím jazyce ANSI C, který by odpovídal struktuře stavového automatu uvedené na obrázku 3.

```
TYPE_STATE Regulator(void)
{
  TYPE_STATE state=ID_Start;
  for(;;)
  {
    /* Main loop */
    switch( state )
    {
      /* State: Start */
      case ID_Start:
        initialize_system();
        state=ID_Heating_is_OFF;
      /* State: Heating is OFF */
      case ID_Heating_is_OFF:
        if( curr_temp >= MAX_VAL )
        {
          set_heating_off();
          state=ID_Critical_stop;
        }
        else if( curr_temp < requested_value )
        {
          set_heating_on();
          state=ID_Heating_is_ON;
        }
        break;
```

```
/* State: Heating is ON */
      case ID_Heating_is_ON:
        if( curr_temp >= MAX_VAL )
        {
          set_heating_off();
          state=ID_Critical_stop;
        }
        else if( curr_temp >= requested_value )
        {
            set_heating_off();
            state=ID_Heating_is_OFF;
        }
        break;
      /* State: Critical stop */
      case ID_Critical_stop:
        return ID_Critical_stop;
    }
  }
}
```

3.2 Generování programového kódu

Jedním z největších úskalí tvorby programových aplikací určených pro embedded systémy je fakt, že je téměř nemožné vytvářet aplikace, které by byly přenosné mezi různými zařízeními. Tento problém je dán skutečností, že ve většině případů jsou tyto aplikace vyvíjeny pro konkrétní použitý řídící procesor a jsou psány tak, aby přímo využívaly vnitřní řídící registry těchto procesorů. Výhoda tohoto způsobu programování spočívá v tom, že lze vytvářet výkonné a vysoce optimalizované aplikace efektivně využívající všech prostředků poskytovaných zvoleným systémem. Nevýhodou pak je, že jiný systém, při nutnosti přechodu na něj, nemusí obsahovat všechny periferie a funkcionalitu použité v původním systému, nebo se mohou lišit adresy různých, v programu použitých, registrů, portů a podobně.

Jakým způsobem by tedy bylo možné obejít toto omezení a zajistit, že programový kód generovaný na základě formálního (ve své podstatě univerzálního – platformně nezávislého) popisu aplikace bude použitelný pro překlad aplikací na různých cílových platformách?

Musíme si především uvědomit, že sestavení aplikace dané jejím formálním popisem se bude sestávat ze dvou základních kroků, mezi nimž může navíc existovat těsná vazba :

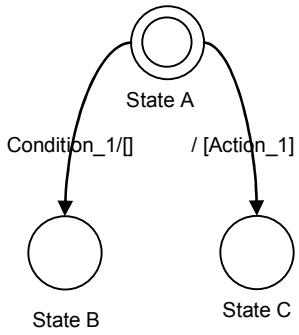
- Generování zdrojového kódu v požadované formě (volba programovacího jazyka a generujících a optimalizačních algoritmů)
- Přizpůsobení generovaného kódu specifikám cílové platformy.

V následujících kapitolách podrobněji rozebereme jak některé z možných typů generujících algoritmů, tak i možnosti automatického generování jazykově a hardwarově nezávislého programového kódu.

3.2.1 Algoritmy pro generování programového kódu

Výstupní programový kód generovaný ze struktury zdrojového stavového automatu může být výsledkem činnosti různých generujících algoritmů. Každý z těchto algoritmů má své výhody i nevýhody a je pouze na uživateli aby rozhodl, který z dostupných algoritmů bude nejlépe vyhovovat jeho požadavkům. Rozdíl výstupu těchto algoritmů spočívá zejména ve způsobu **řízení toku programu** a tudíž i v rozsahu a srozumitelnosti (rozumějme člověku) výsledného programového kódu. Mezi tyto algoritmy můžeme zařadit např. algoritmy založené na využití kódových návěští a příkazu `goto` (tzv. **GoTo** algoritmus), algoritmy využívající větvení programu pomocí příkazu `switch` (tzv. **Loop-Case** algoritmus), nebo algoritmy generující tabulky s ukazateli na funkce, které v podstatě odpovídají klasickým přechodovým tabulkám (viz. obr 4). Je zřejmé, že například posledně jmenovaný algoritmus bude produkovat programový kód s nejmenším rozsahem programových řádků, jeho čitelnost však bude pro člověka mnohem složitější, než čitelnost programu vytvořeného na základě algoritmu Loop-Case.

Následující obrázek ilustruje rozdíly ve výstupním programovém kódu produkovaném dvěma výše zmíněnými algoritmy: Loop-Case a GoTo.




```
/* Loop-Case algoritmus */

TYPE_STATE DoSomething(void)
{
  TYPE_STATE state=ID_A;
  for(;;)
  {
    /* Main loop */
    switch( state )
    {
      /* State: A */
      case ID_A:
        if(/*Condition 1*/)
        {
          state=ID_B;
        }
        else
        {
          /* Action 1 */
          state=ID_C;
        }
        break;
      /* State: B */
      case ID_B:
        return ID_B;
      /* State: C */
      case ID_C:
        return ID_C;
    }
  }
}
```

```
/* Go-To algoritmus */

TYPE_STATE DoSomething(void)
{
  /* State: A */
  if(!(/*Condition 1*/))
  {
    /* Action 1 */
    goto State_ID_C;
  }

  /* State: B */
  return ID_B;

  /* State: C */
  State_ID_C:
  return ID_C;
}
```

Obr. 5 Srovnání dvou různých přístupů ke generování programového kódu

Jak je patrné z obrázku 5, generující algoritmus může významně ovlivnit jak strukturu, tak i rozsah generovaného programového kódu a proto je vždy důležité zvolit takový algoritmus, který bude nejlépe vyhovovat všem požadavkům na vlastnosti generovaného výstupu. Samozřejmě, pro dosažení co nejvyšší efektivity při generování kódu lze použít také různé optimalizační metody, které mohou dále zkvalitňovat získaný výstupní kód. O těchto optimalizačních postupech bude pojednávat kapitola 3.2.2.

Nyní se podrobněji zaměříme na vlastnosti a možný způsob implementace dvou výše zmíněných algoritmů pro generování programového kódu, a to GOTO a Loop-Case algoritmů. Pro větší názornost předpokládejme, že dané algoritmy budou generovat výstup pouze v programovacím jazyce ANSI C.

3.2.1.1 Základní koncepty, stavy aplikace

Algoritmy GOTO i Loop-Case obsahují některé společné koncepty využívané při generování kódu. Jedná se zejména o způsob, jakým jsou generovány programové fragmenty odpovídající jednotlivým typům stavů a vlastní automaty.

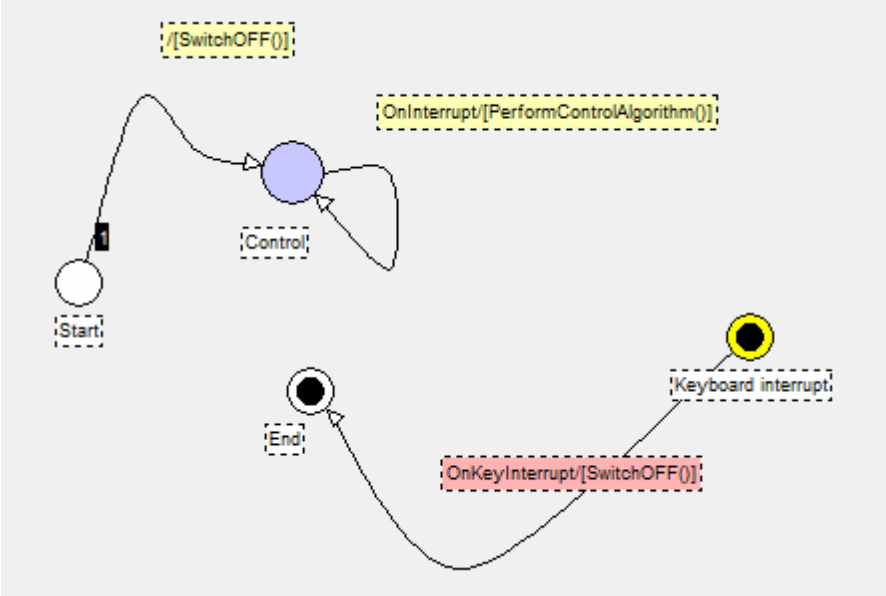
V teorii KSA jsou v zásadě rozlišovány 3 typy stavů: **počáteční** (iniciální) stav, **normální** (průběžný) stav, a **koncový** stav. Tyto základní typy bohatě postačují pro popis chování jakékoliv aplikace, jejíž programový tok je řízen čistě na základě vnitřního stavu aplikace. V generovaném programovém kódu jsou pak tyto typy stavů zohledňovány následovně:

- **Počáteční** stav je vždy generován jako první ihned po vstupu do funkce.
- **Normální** stavy mohou být generovány v libovolném okamžiku průběhu generování kódu.
- Na pořadí **koncových** stavů rovněž nezáleží, jsou ale terminačními body algoritmu, tzn. ukončují běh části programu implementující zpracovávaný KSA a (ve většině případů) vrací návratovou hodnotu definující daný stav (své ID).

Grafické znázornění těchto typů stavů může být následující:

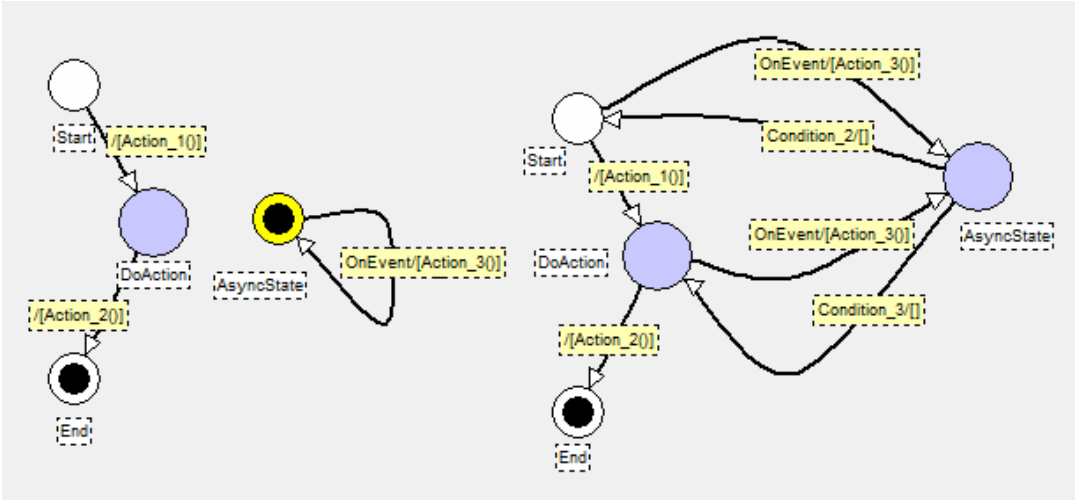
Obr. 6 Typy stavů

V některých typech úloh (a to zejména v oblasti embedded systémů) se setkáváme s aplikacemi, jejichž programový tok je řízen také na základě (vnějších) asynchronních událostí, na které je nutné promptně reagovat bez ohledu na to, v jakém stavu se aplikace zrovna nachází. Těmito událostmi mohou být např. požadavky na obsluhu přerušení interních, nebo externích periferií embedded zařízení. Aby bylo možné efektivně a hlavně srozumitelně popsat i takové případy, zavedeme ještě čtvrtý typ stavu, který budeme nazývat **asynchronní stav**. Do asynchronního stavu se může daná aplikace dostat pouze na základě splnění určité podmínky, typicky při příchodu požadavku na obsluhu přerušení. Použití asynchronní stavu v projektu může být následující:

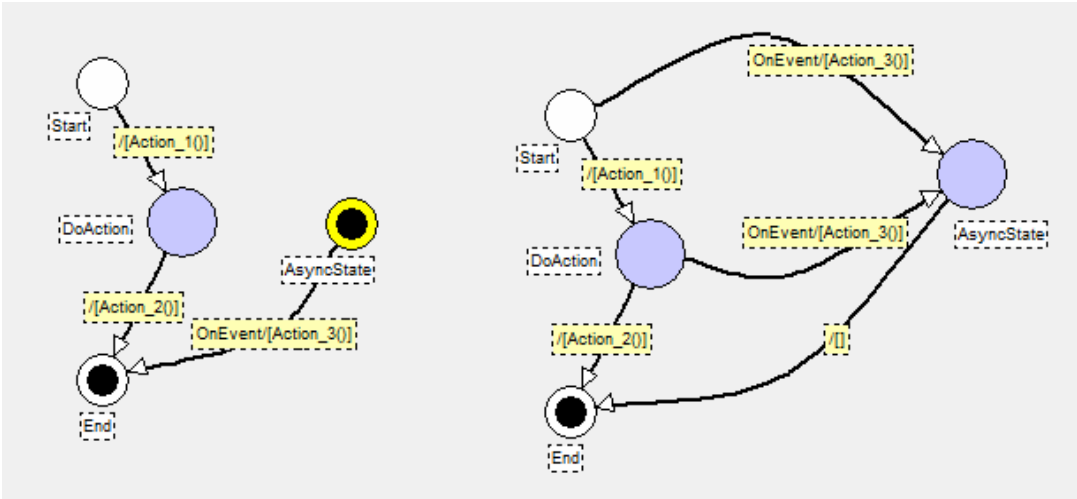


Obr. 7 Použití asynchronního stavu

Může se zdát, že takový stav odporuje pravidlům tvorby deterministických stavových automatů, ale opak je pravdou. Jedná se totiž o zjednodušený zápis grafu, kdy ze všech stavů vede cesta do jednoho specifického stavu indikujícího požadavek na zpracování asynchronní události, z něhož vedou návratové cesty do stavů, ve kterých byl daný požadavek registrován.



Obr. 8 Formální význam samostatného asynchronního stavu



Obr. 9 Formální význam asynchronního stavu s definovaným návratem

Modifikací pak je asynchronní stav s návratovou cestou (viz obr. 9); ten se od samostatného asynchronního stavu liší tím, že má definovanou návratovou cestu do konkrétního stavu automatu (typicky koncového).

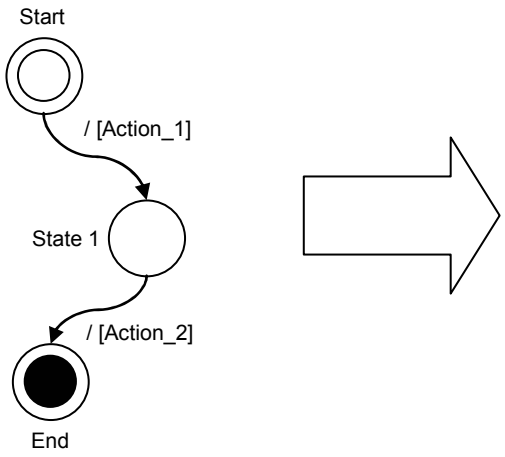
Programový kód samotného generovaného automatu bývá ve většině případů generován do těla libovolné funkce, jejíž jméno může odpovídat jménu generovaného

automatu (odpovídá-li zásadám ANSI C). Tato funkce by měla vracet hodnotu odpovídají např. identifikátoru koncového stavu.

3.2.1.2 GOTO algoritmus

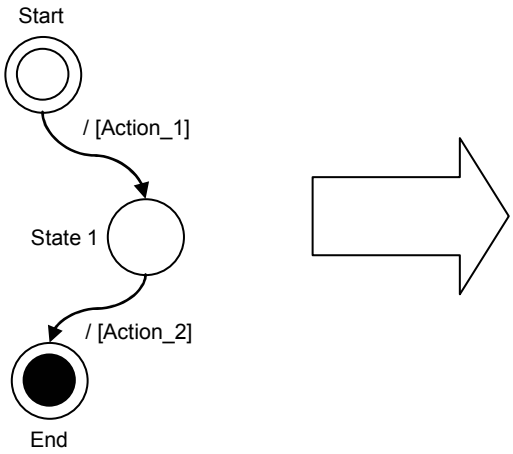
Základní myšlenkou GOTO algoritmu je, že každý stav KSA lze v analogii programových stavů definovat jako část programového kódu označenou návěštím, např. návěštím se jménem daného stavu (pokud jméno odpovídá normám formální správnosti programových identifikátorů).

K přechodu mezi těmito částmi programu pak lze použít klasický příkaz `goto`, zajišťující skok na definované návěští. Tyto skoky mohou být **přímé** (v případě přechodů bez strážící podmínky), nebo **podmíněné** splněním příslušného logického výrazu testovaného pomocí příkazu `if` (přechody automatu strážené podmínkou). V určitých případech, kdy pořadí generovaných stavů odpovídá také jejich návaznosti definované přechody mezi nimi a tyto přechody nejsou stráženy podmínkami, lze programový kód těchto stavů jednoduše generovat za sebe bez použití příkazů skoku. Všechny jmenované možnosti jsou znázorněny na následujících obrázcích:



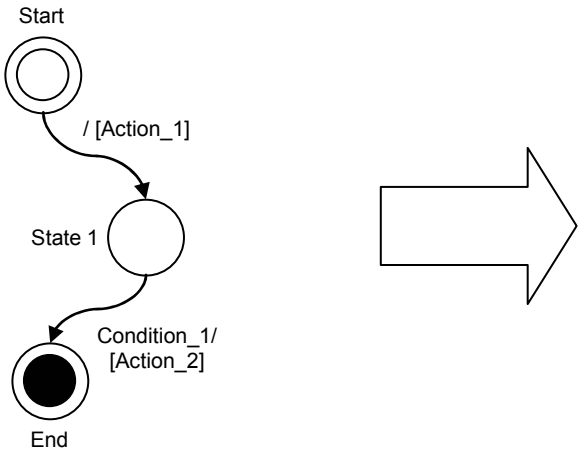

```
TYPE_STATE AUT1(void)
{
  /* State: Start */
  State_ID_Start:
  Action_1();
  goto State_ID_State_1;

  /* State: End */
  State_ID_End:
  return ID_End;

  /* State: State 1 */
  State_ID_State_1:
  Action_2();
  goto State_ID_End;
}
```

Obr. 10 Řízení toku programu bez podmínek přechodů a s neoptimálním pořadím generovaných stavů




```
TYPE_STATE AUT1(void)
{
  /* State: Start */
  State_ID_Start:
  Action_1();

  /* State: State 1 */
  State_ID_State_1:
  Action_2();

  /* State: End */
  State_ID_End:
  return ID_End;
}
```

Obr. 11 Řízení toku programu bez podmínek přechodů a s optimálním pořadím generovaných stavů



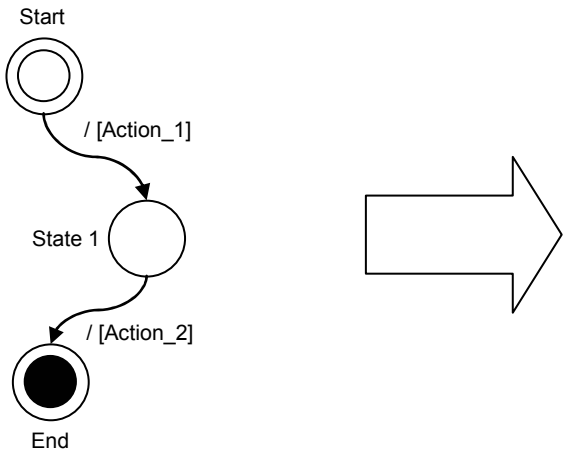

```
TYPE_STATE AUT1(void)
{
  /* State: Start */
  State_ID_Start:
  Action_1();

  /* State: State 1 */
  State_ID_State_1:
  if( Condition_1() ) {
    Action_2();
  }
  else
    goto State_ID_State_1;

  /* State: End */
  State_ID_End:
  return ID_End;
}
```

Obr. 12 Řízení toku programu s podmínkami přechodů

3.2.1.3 LOOP-CASE algoritmus

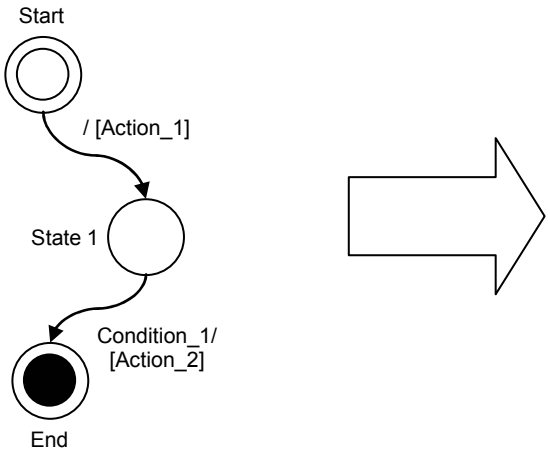
Na rozdíl od předchozího algoritmu, algoritmus Loop-Case nepoužívá pro větvení programu příkazy skoku, ale využívá vícenásobné větvení pomocí zřetězených, nebo strukturovaných podmínkových příkazů (např. příkazy `if`, `else if`, nebo `switch` v programovacím jazyce ANSI C/C++).

Základní myšlenkou algoritmu je, že máme k dispozici proměnnou, ve které udržujeme informaci o aktuálním zpracovávaném stavu automatu a tuto proměnnou průběžně testujeme v nekonečné smyčce, kterou může přerušit pouze koncový stav automatu. Tato smyčka může být implementována pomocí libovolného příkazu pro tvorbu cyklů (např. `for(;;)` či `while(1)` v jazyce ANCI C/C++). Obsah stavové proměnné může být měněn bezprostředně po vstupu do programové sekce odpovídající určitému stavu (to v případě že přechod do následujícího stavu není strážen podmínkou), nebo po kladném vyhodnocení určitého logického výrazu (u přechodů stážených podmínkou). Následující obrázky ilustrují způsob generování programového kódu pro přímé a podmíněné přechody mezi stavy.




```
TYPE_STATE AUT1(void)
{
  TYPE_STATE state=ID_Start;
  for(;;)
  {
    switch( state )
    {
      case ID_Start:
        Action_1();
        state=ID_State_1;
      case ID_State_1:
        Action_2();
        state=ID_End;
      case ID_End:
        return ID_End;
    }
  }
}
```

Obr. 13 Řízení toku programu bez podmínek přechodů




```
TYPE_STATE AUT1(void)
{
  TYPE_STATE state=ID_Start;
  for(;;)
  {
    switch( state )
    {
      case ID_Start:
        Action_1();
        state=ID_State_1;
      case ID_State_1:
        if( Condition_1() )
        {
          Action_2();
          state=ID_End;
        }
        break;
      case ID_End:
        return ID_End;
    }
  }
}
```

Obr. 14 Řízení toku programu s přechody sráženými podmínkami

3.2.1.4 Implementace asynchronních stavů

Implementace asynchronních stavů u obou generujících algoritmů je zřejmá. samostatných asynchronních stavů pouze vygenerujeme akce spojené s přechodem do tohoto stavu jako těla obslužných rutin příslušných událostí (ISR rutin pro obsluhy přerušení).

V případě asynchronních stavů s definovaným přechodem do jednoho ze stavů zpracovávaného automatu musíme navíc přidat programový kód zajišťující přechod do příslušného stavu. K tomu budeme potřebovat globální příznak nastavovaný v rámci obslužné rutiny události/přerušení, který bude nastaven po provedení příslušných operací a bude testován jako další přechodová podmínka ve všech stavech automatu.

U GOTO algoritmu lze takový dodatečný přechod implementovat jako další klasickou podmínku přechodu, u Loop-Case algoritmu lze tento test předřadit před vlastní strukturovaný podmínkový výraz testující přechody všech ostatních klasických stavů.

3.2.2 Verifikace a optimalizace stavových automatů a programového kódu

3.2.2.1 Verifikace a optimalizace struktury zdrojového FSM

Před započetím vlastního procesu generování programového kódu je nutné nejprve ověřit formální správnost zdrojového stavového automatu. Pokud bychom tak neučinili, mohli bychom získat nefunkční programový kód, programový kód neodpovídající požadované aplikační logice, nebo by proces generování kódu zcela selhal. Proto je nutné ověřit zejména:

- Existenci počátečního stavu
- Neočekávané koncové body (klasické stavy, z nichž nevedou žádné přechody)
- Dosažitelnost jednotlivých stavů
- Správnou definici hran přechodů automatu a jejich ohodnocení podmínkami a případně i prioritami

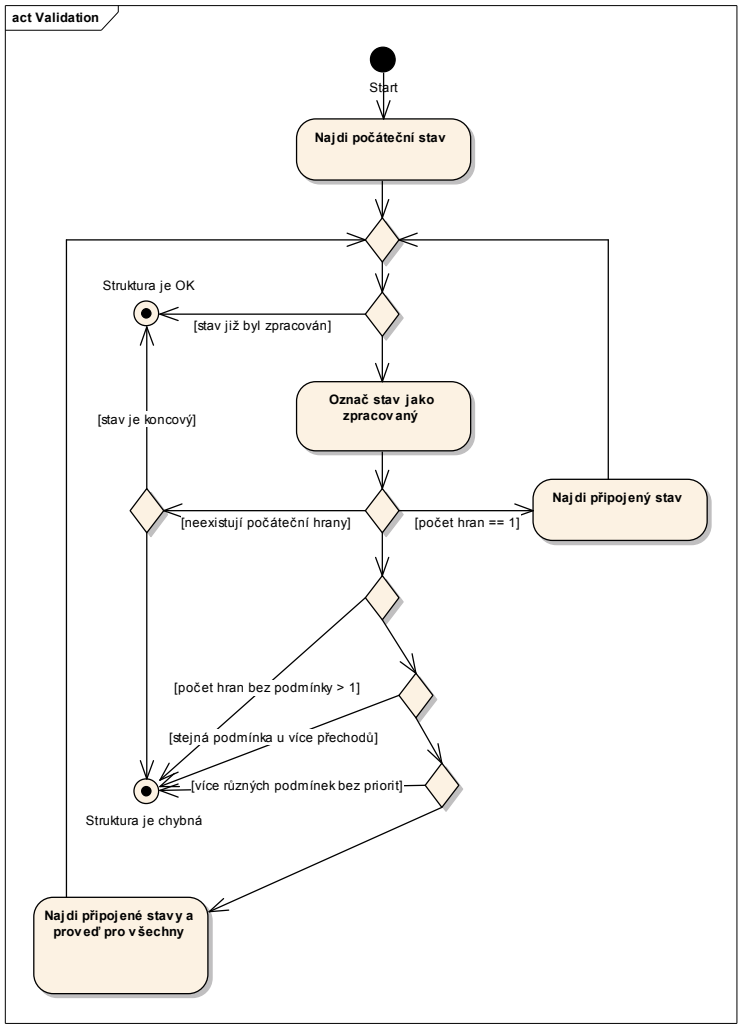
Algoritmy pro ověření dosažitelnosti stavů automatu jsou obecně známé [5] a ani jejich praktická implementace není složitá. Drobnými úpravami lze tyto algoritmy modifikovat tak, aby byly schopné ověřovat také správnost ohodnocení hran přechodů.

Po ověření formální správnosti stavového automatu bývá také vhodné aplikovat některý z algoritmů umožňujících redukovat jeho strukturu tak, abychom dosáhli co možná nejnižšího počtu použitých stavů při zachování jeho logiky. I pro tuto činnost existují standardní algoritmy, jakým je např. vyhledávání a redukce ekvivalentních stavů automatu [5]. Modifikacemi tohoto algoritmu lze vytvořit další algoritmy vhodné například pro

- slučování ekvivalentních paralelních větví,
- slučování přímých větví stavového automatu.

3.2.2.1.1 Algoritmus ověření dosažitelnosti stavů

Při ověření dosažitelnosti všech stavů vycházíme z předpokladu, že správně navržený stavový diagram musí být propojeným grafem (použití asynchronních stavů v našem popisu nehraje roli, jelikož mohou být formálně nahrazeny běžnými stavy s příslušnými dodatečnými přechody, jak bylo ukázáno v kapitole 3.2.1.1) a všechny přechodové hrany vycházející z jednotlivých stavů musí být správně ohodnoceny, tzn. musí umožňovat deterministicky rozhodnout o následujícím kroku programu.

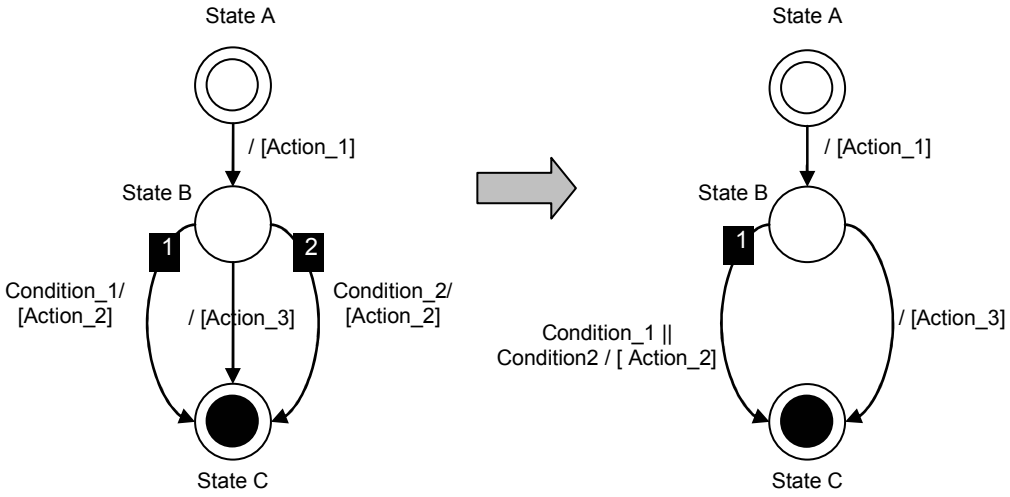


Obr. 15 Diagram aktivit algoritmu pro ověřování dosažitelnosti stavů

Takový ověřující algoritmus, jehož diagram aktivit je zobrazen na obrázku 15, lze relativně snadno navrhnout jako rekurzivní algoritmus postupně procházející jednotlivé stavy automatu po jeho definovaných přechodech. Každý takový stav by byl označen jako dosažený, byly by provedeny testy správného ohodnocení jeho hran (pokud by existovaly) a na stavy, do nichž vedou tyto hrany by byl znovu (rekurzivně) aplikován ověřující algoritmus.

3.2.2.1.2 Algoritmus slučování paralelních větví

Základní myšlenkou **slučování ekvivalentních paralelních větví** spočívá v tom, že paralelní hrany (tzn. hrany se stejnými výchozími i koncovými stavy) se stejnou podmínkou strážící přechody (tedy reagující na stejné symboly vstupní abecedy), ale různými akcemi vykonávanými v průběhu procesu přechodu, lze sloučit do jediné hrany s kombinovanou akcí (složenou posloupností symbolů výstupní abecedy), tak jak je vidět z obrázku 16.



Obr. 16 Slučování paralelních větví automatu

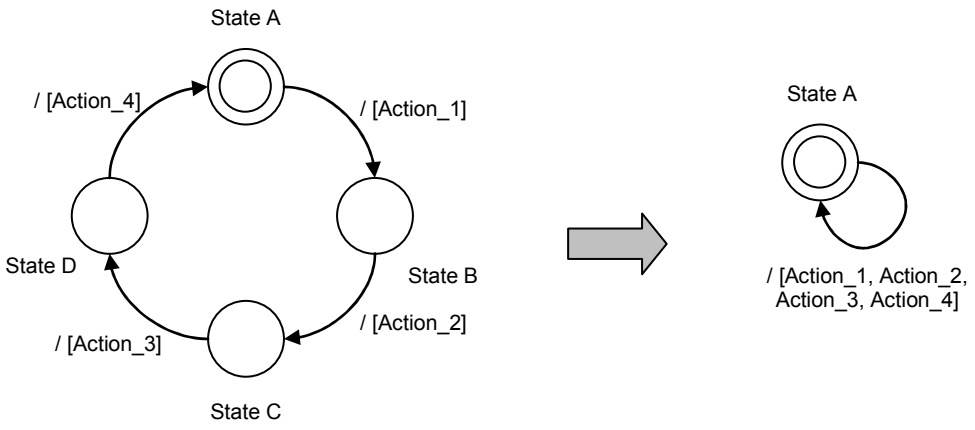
Tato modifikace struktury stavového automatu se ve výsledném programovém kódu promítne následovně:

```
/* Původní struktura */

TYPE_STATE DoSomething(void)
{
  /* State: A */
  /* Action 2 */

  /* State: B */
  if( /* Condition 1 */ )
  {
    /* Action 2 */
  }
  else if( /* Condition 2 */ )
  {
    /* Action 2 */
  }
  else
  {
    /* Action 3 */
  }

  /* State: C */
  return ID_C;
}
```

```
/* Optimalizovaná struktura */

TYPE_STATE DoSomething(void)
{
  /* State: A */
  /* Action 2 */

  /* State: B */
  if( (/* Condition 1 */)||(/* Condition 2 */) )
  {
    /* Action 2 */
  }
  else
  {
    /* Action 3 */
  }

  /* State: C */
  return ID_C;
}
```

3.2.2.1.3 Algoritmus slučování přímých větví

Algoritmus **slučování přímých větví** stavového automatu zase využívá fakt, že hrany přechodů stavů s pouze jedním vstupem a výstupem, které navíc neobsahují podmínky strážící dané přechody (čili akceptují prázdný symbol vstupní abecedy), lze opět sloučit do jednoho přechodu s kombinovanou akcí.



Obr. 17 Slučování přímých větví automatu

Tvar výstupního programového kódu bude po takové optimalizaci následující:

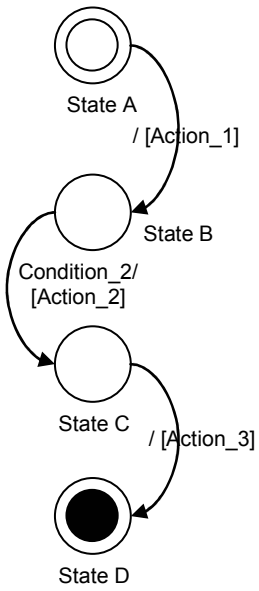
```
/* Původní struktura */

  TYPE_STATE state=ID_A;
  for(;;)
  {
    /* Main loop */
    switch( state )
    {
      /* State: A */
      case ID_A:
        /* Action 1 */
        state=ID_B;
      /* State: B */
      case ID_B:
        /* Action 2 */
        state=ID_C;
      /* State: C */
      case ID_C:
        /* Action 3 */
        state=ID_D;
      /* State: D */
      case ID_D:
        /* Action 4 */
        state=ID_A;
        break;
    }
  }
}
```

```
/* Optimalizovaná struktura */

void DoSomething(void)
{
  TYPE_STATE state=ID_A;
  for(;;)
  {
    /* Main loop */
    switch( state )
    {
      /* State: A + B + C + D */
      case ID_A:
        /* Action 1 */
        /* Action 2 */
        /* Action 3 */
        /* Action 4 */
        break;
    }
  }
}
```

3.2.2.2 Minimalizace generovaného programového kódu

Zdrojový programový kód aplikací určených pro embedded systémy by měl být vytvořen co možná nejefektivněji. Jelikož je pro generování zdrojového kódu aplikace z jejího formálního popisu možné použít několik různých algoritmů, je nutné správně rozhodnout, který je pro danou situaci nejvýhodnější. Různé generující algoritmy se liší především použitými příkazy pro řízení toku programu a tím pádem ovlivňují jak přehlednost, tak i velikost generovaného zdrojového kódu. To bylo ostatně dokázáno v kapitole pojednávají o typech generujících algoritmů.

Pokud bychom chtěli generovaný programový kód dále minimalizovat, bylo by například vhodné použít takové algoritmy, které odstraňují nepotřebné programové fragmenty a příkazy tak, jak je naznačeno v následujícím příkladu.



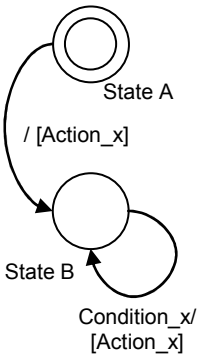

```
/* Loop-Case algoritmus */

TYPE_STATE DoSomething(void)
{
  TYPE_STATE state=ID_A;
  for(;;)
  {
    /* Main loop */
    switch( state )
    {
      /* State: A */
      case ID_A:
        /* Action 1 */
        state=ID_B;
      /* State: B */
      case ID_B:
        if( /* Condition 1 */ )
        {
          /* Action 2 */
          state=ID_C;
        }
        break;
      /* State: C */
      case ID_C:
        /* Action 3 */
        state=ID_D;
      /* State: D */
      case ID_D:
        return ID_D;
    }
  }
}
```

```
/* Go-To algoritmus */

TYPE_STATE DoSomething(void)
{
  /* State: A */
  /* Action 1 */

  /* State: B */
  State_ID_B:
  if( /* Condition 1 */ )
  {
    /* Action 2 */
  }
  else
    goto State_ID_B;

  /* State: C */
  /* Action 3 */

  /* State: D */
  return ID_D;
}
```

Obr. 18 Minimalizace pomocí vypouštění nepotřebných programových fragmentů

Na obrázku 18 je patrné, jakým způsobem by bylo možno snížit počet generovaných programových řádků u Loop-Case i GOTO algoritmu. V prvním případě byly ve zdrojovém kódu vynechány některé nepotřebné příkazy `break`, čímž byl nejen zmenšen výsledný program, ale jeho běh byl také urychlen, protože příkaz `switch` nemusel provádět další testování podmínky určené k řízení toku programu. Druhý příklad ilustruje možnosti vynechání některých programových návěští a příkazů pro skoky u GOTO algoritmu.

Další možností, jak optimalizovat generovaný programový kód, je určení způsobu, jakým budou využity uživatelem definované fragmenty programu zastávající úlohu akcí nebo podmínek přechodů stavového automatu (symbolů výstupní, nebo vstupní abecedy automatu).




```c
/* Akce volané jako funkce */

void Action_1(void)
{
  /* Action 1 */
}

unsigned char Condition_1(void)
{
  /* Condition 1 */
  return 1;
}

void DoSomething(void)
{
  /* State: A */
  Action_1();

  /* State: B */
  State_ID_B:
  if( Condition_1() )
  {
    Action_1();
  }
  goto State_ID_B;
}
```

```c
/* Akce vložené přímo do
prog. kódu */

void DoSomething(void)
{
  /* State: A */
  /* Action 2 */

  /* State: B */
  State_ID_B:
  if( /* Condition 2 */ )
  {
    /* Action 2 */
  }
  goto State_ID_B;
}
```

Obr. 19 Možné implementace kódu akcí a podmínek přechodů automatu

Tyto programové fragmenty lze do výstupního kódu importovat dvěma způsoby: buďto jako části programového kódu **vkládané přímo do místa jejich volaní**, či jako těla **procedur volaných z příslušných míst**. Způsob použití těchto programových fragmentů pak ovlivňuje nejen rozsah programového kódu, ale také rychlost běhu programu (režie systému spojené s voláním funkcí), ale také to, zda takový program bude vůbec funkční; na systémech s omezenou či fixní velikostí zásobníku může jeho přetečení z důvodu nadměrného použití vnořených funkcí způsobit pád, nebo „zamrznutí“ celé aplikace a potažmo celého zařízení. Oba dva možné způsoby implementace akcí automatu je uvedeno na následujícím obrázku.

3.2.3 Generování jazykově nezávislého programového kódu

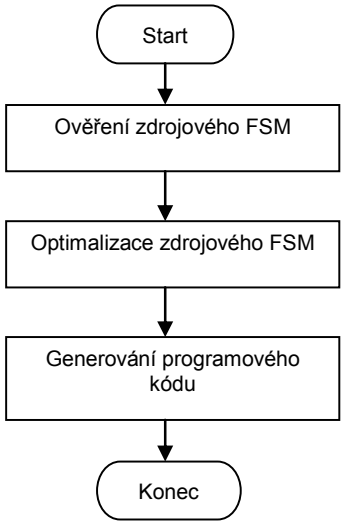
Jak již bylo řečeno, stavové automaty nám nabízejí možnost formálně popsat aplikační logiku vyvíjené aplikace. Budeme-li však chtít takovým automatem popsanou aplikaci používat na konkrétní platformě (embedded zařízení), musíme mít k dispozici její kód napsaný pomocí programovacího jazyka, pro který existuje na dané platformě překladač. Je tedy nutné takový kód vygenerovat.

Celý zjednodušený proces generování zdrojového kódu z FSM je ilustrován na obrázku 20.

Prvním krokem je **ověření integrity zdrojového stavového automatu** a nalezení a vyloučení všech formálních chyb v popisu chování generované aplikace.

Druhým krokem je **optimalizace struktury** formálně správného FSM. V tomto kroku lze snížit složitost zdrojového stavového automatu a redukovat tak množství výsledného programového kódu generovaného na jeho základě.

Třetím krokem pak bude vlastní **generování výsledného zdrojového kódu**. Tato činnost bude prováděna tzv. **generátorem kódu**.



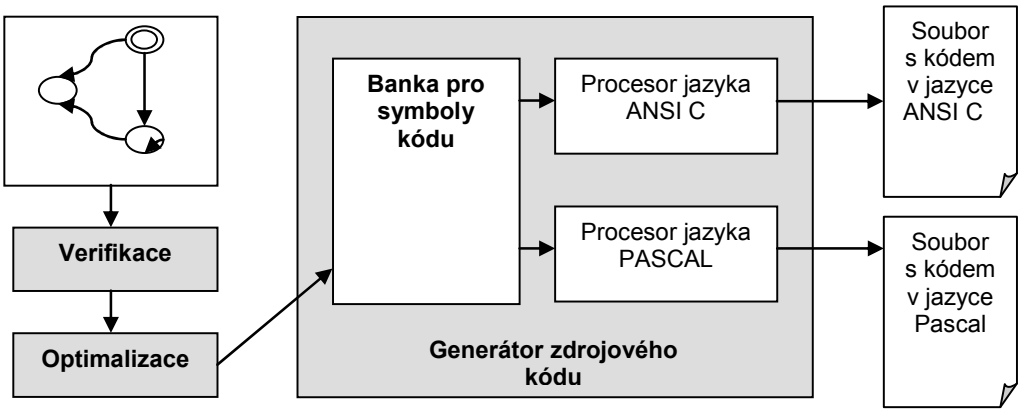
Obr. 20 Proces generování kódu

Generátor kódu čte strukturu FSM a za použití zvoleného algoritmu generuje **programové fragmenty** odpovídající jednotlivým částem zdrojového automatu. Tyto programové fragmenty mohou v zásadě obsahovat:

- Deklarace a definice metod a funkcí
- Deklarace proměnných a přiřazení hodnot
- Podmínkové příkazy

- Uživatelem definované části zdrojového kódu použité jako těla podmínek a akcí, které jsou přímo součástí stavového automatu.

Je zřejmé, že syntaxe těchto programových fragmentů závisí na zvoleném programovacím jazyce, tudíž budeme v průběhu generování kódu potřebovat entitu, která nám bude poskytovat pouze **symbolické popisy** jednotlivých fragmentů. Tato entita bude součástí generátoru kódu a budeme ji nazývat **banka pro symboly kódu** (anglicky též **code tokenizer**). Symbolické části generovaného kódu produkovaného touto bankou pak budou dále zpracovány částí nazývanou **procesor zdrojového kódu** (**code processor**). Ten již bude mít za úkol zapisovat symbolické fragmenty zdrojového kódu přímo v syntaxi zvoleného výstupního programovacího jazyka. Stejně jako banka, i procesor zdrojového kódu je součástí generátoru kódu, ovšem na rozdíl od banky symbolů může generátor obsahovat libovolné množství procesorů – pro každý podporovaný programovací jazyk jeden. V tom případě pak lze v průběhu procesu generování zdrojového kódu vytvořit jeden výstupní soubor se syntaxí odpovídající zvolenému programovacímu jazyku, nebo taky hned několik různých výstupních souborů se sémanticky shodným kódem, ovšem se syntaxí příslušných programovacích jazyků. Tyto výstupní soubory zdrojového kódu pak již mohou být jednoduše přeloženy pomocí vhodného překladače.



Obr. 21 Logická struktura generátoru zdrojového kódu programu

Tímto způsobem lze tedy zajistit, že programový kód vytvořený generátorem kódu pracujícím na výše uvedeném principu bude zapsán vždy v syntaxi, kterou vyžaduje námi vybraný překladač; formální popis je tedy jazykově nezávislý.

Jak tomu však bude v případě platformní nezávislosti generovaného kódu?

3.2.4 Generování platformě nezávislého programového kódu

Při psaní programů po embedded zařízení se velmi často využívá přímého použití řídících registrů integrovaných periferií. Pomocí těchto registrů lze provádět operace s různými komunikačními rozhraními, vstupně výstupními obvody, časovači apod. Kámen úrazu spočívá především v tom, že u různých typů mikroprocesorů použitých v embedded zařízeních se způsob práce s těmito periferiemi a jejich adresy mění. Za předpokladu, že zdrojový kód generovaný ze stavového automatu využívá některé z těchto periferií, měli bychom být schopni zajistit, aby byl generovaný kód přenosný mezi různými cílovými platformami.

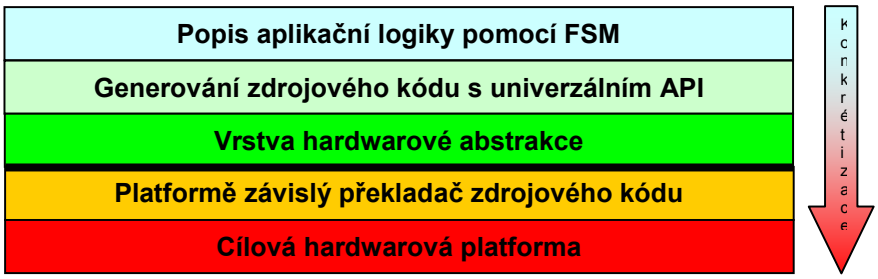
V oblasti klasických stolních počítačů je problém unifikovaného přístupu k různorodému hardware řešen použitím **operačního systému**, který poskytuje univerzální API (programovací rozhraní) a pomocí ovladačů sám přistupuje k řízeným periferiím. I ve světě embedded zařízení existují operační systémy, které nám mimo jiné umožňují využívat komfortu univerzálního přístupu k hardware; tyto systémy však není možné nasadit úplně všude. Důvodem jsou hardwarová omezení cílových platforem, kdy můžeme být silně limitování například velikostí volné operační paměti, zásobníku, nebo výkonu mikroprocesoru. Také v tomto případě však existují cesty, jak vytvořit (nebo nahradit) **softwarovou vrstvu hardwarové abstrakce** (dále jen **HAL**; Hardware Abstraction Layer) a zajistit tak, aby byl generovaný kód použitelný i na více cílových platformách.

Na výběr máme ze dvou možností:

1. Využití nástrojů, které by poskytovaly **univerzální API** pro přístup k řízeným periferiím a tyto funkce by byly **v průběhu překladu** (nebo **preprocesingu** zdrojového kódu) nahrazovány přímým voláním řídících registrů cílové platformy.

2. Využití **multiplatformní knihovny** obsahující univerzální API, která by byla implementována pro každou cílovou platformu zvlášť a byla připojena (linkována) k vlastnímu programu.

Výhoda první možností spočívá zejména v menším objemu generovaného kódu. Nevýhodou pak může být poněkud pracnější způsob překladu aplikace. Naproti tomu, při použití multiplatformních knihoven odpadá nutnost použití více vývojových nástrojů, nevýhodou pak bývá méně optimalizovaný a objemnější zdrojový kód.



Obr. 22 Platformě nezávislý vývojový proces využívající FSM pro popis aplikační logiky

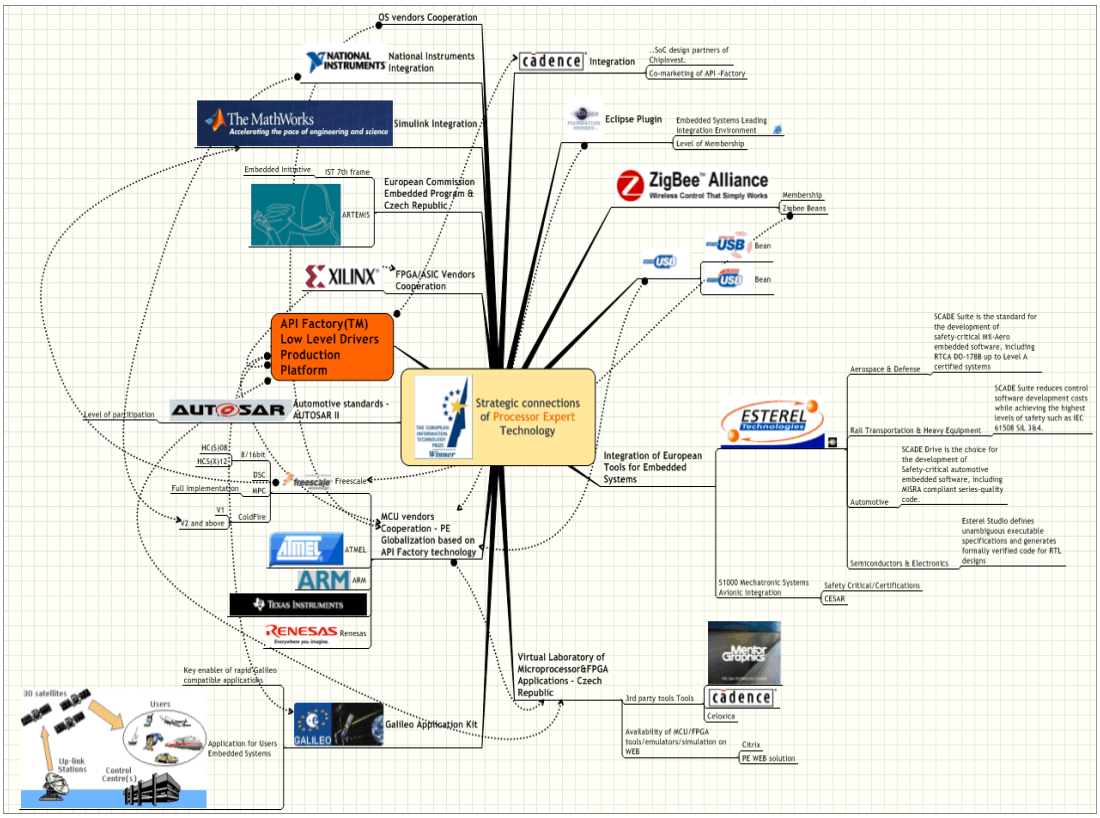
3.3 Implementace HAL

Vzhledem k využité technologii a možnostem integrace se pro implementaci HAL jako nejvýhodnější pro naše účely jeví již zmiňovaná aplikace Processor Expert [20]. Aplikace využívá první způsob náhrady HAL uvedený v předchozí kapitole, tedy nabízí univerzální API, jehož procedury jsou modifikovány v průběhu preprocesingu zdrojového kódu tak, aby odpovídaly specifikaci cílové platformy. Výsledkem je efektivní produkční kód, který lze bez problémů nasadit k ostrému použití.

Aplikace Processor Expert podporuje širokou paletu MCU a je distribuována komerční formou samostatně, nebo jako rozšíření IDE CodeWarrior. Navíc, strategické

propojení technologie Processor Experta s dalšími výrobci a dodavateli embedded systémů je rozsáhlé (jak dokazuje schéma na obrázku 23), a poskytuje nám tak určitou záruku, že vývoj a podpora této technologie bude pokračovat i v budoucnu.

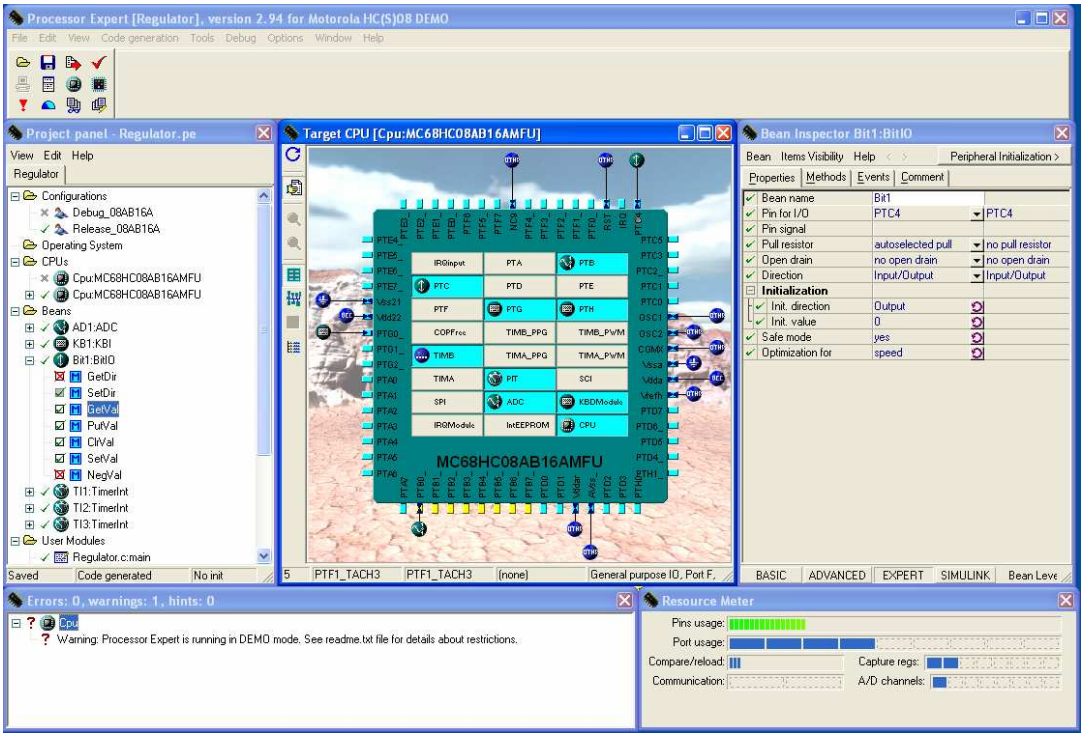
Další velkou výhodou je možnost integrace nástrojů třetích stran s Processor Expertem (dále jen PE). Aplikace nabízí programové rozhraní založené na technologii COM, které umožňuje detailní využití kompletní funkcionality PE v hostovaných, nebo hostujících aplikacích. Díky tomuto rozhraní mohou být tyto aplikace řízeny z prostředí PE, nebo naopak, samy využívat, a do jisté míry ovlivňovat, činnost samotného PE.



Obr. 23 Schéma strategického propojení technologie ProcessorExpert (zdroj UNIS)

Jak již bylo zmíněno v kapitole 1.3, PE zapouzdřuje jednotlivé interní i externí periferie MCU do komponent nazývaných **Beany**, jejichž **vlastnosti**, **metody** a **události**

mohou být použity v uživatelem vytvořeném programovém kódu. Tyto objekty PE interně využívají vlastního makrojazyka (uživateli je dokonce umožněno vytvářet nové komponenty zapouzdřující další HW zařízení/periferie, či provádějící jinou činnost), který je v průběhu preprocesingu zdrojového kódu převáděn na použití nativních funkcí a registrů cílového MCU. Každý projekt PE může obsahovat libovolný počet takových MCU a jejich jednoduchým aktivováním (přepínáním) vznikne po přegenerování a překladu zdrojového kódu výstup vhodný k nasazení na konkrétní cílové platformě.



Obr. 24 Prostředí aplikace Processor Expert

3.4 Praktická implementace

Aplikace, o níž pojednává tato práce a která umožňuje interaktivně vytvářet stavové diagramy a za pomoci aplikace PE generovat platformě nezávislý programový kód pro embedded systémy, byla nazvána **State Builder** (dále jen **SB**). Jednotlivé technologie a postupy použité při tvorbě této aplikace, jakož i popis její struktury a způsobu použití jsou uvedeny v následujících kapitolách.

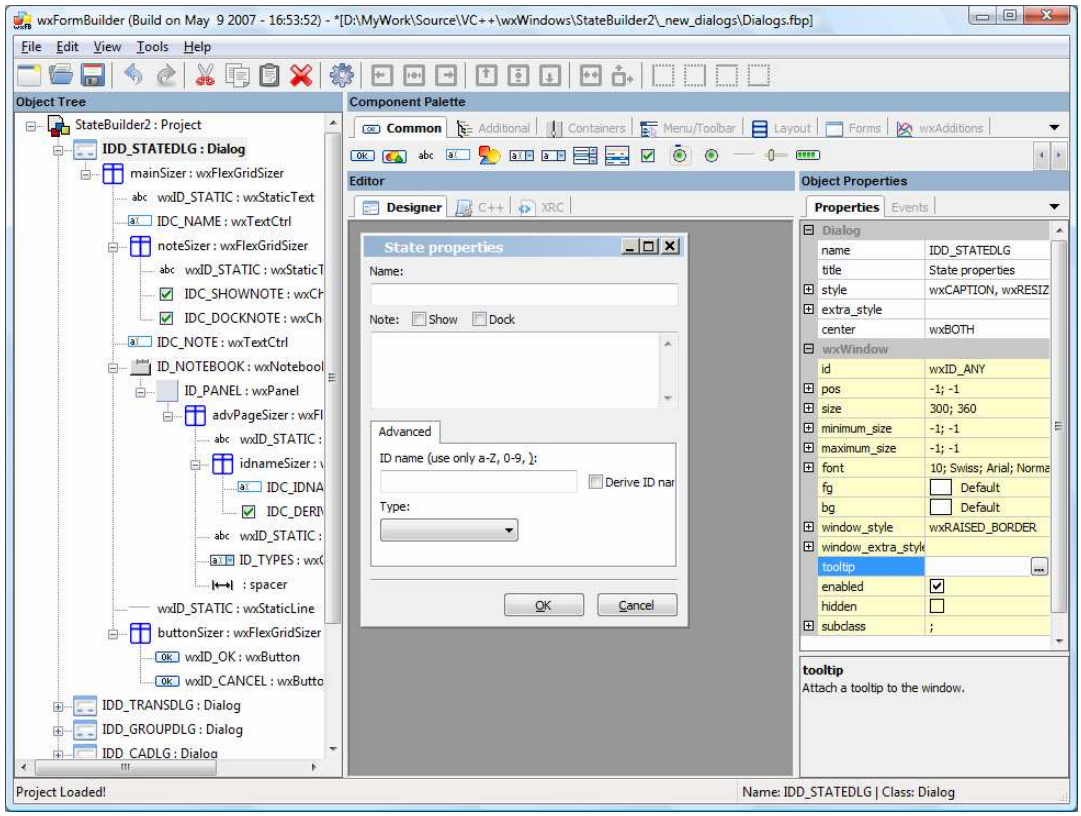
3.4.1 Použité vývojové prostředky a nástroje

Při vývoji a vlastní implementaci algoritmů i celé aplikace byly využity převážně open source technologie a nástroje, umožňující vývoj platformě nezávislých aplikací. Zejména se jedná o programovací jazyk C/C++ a multiplatformní softwarovou knihovnu wxWidgets [31], která je volně šířitelná a použitelná i pro komerční projekty. Tato knihovna pokrývá kompletní funkcionalitu hostujícího operačního systému; GUI, síťové technologie, řízení procesů a vláken, atd. Navíc se nejedná o interpretovanou technologii, a tudíž aplikace vytvořené pomocí této knihovny jsou téměř tak rychlé a výkonné, jako aplikace vytvořené pomocí nativního API cílového OS [8], [31]. Knihovnu wxWidgets lze použít s širokou řadou překladačů různých programovacích jazyků (C++, Python, C#, Lua, …) a vývojových prostředí, a proto ani v tomto ohledu nebudeme při vývoji ničím limitování.

Další hojně užitou technologií bylo XML, použité jako formát konfiguračních souborů SB i jako prostředek pro předávání strukturovaných informací mezi PE a SB. Knihovna wxWidgets sice obsahuje implementaci DOM modelu XML, ale ve verzi knihovny použité v době tvorby aplikace byly třídy zapouzdřující XML parser i generátor považovány za nestabilní z hlediska specifikace aplikačního rozhraní, a proto bylo nutné zvolit jinou implementaci. Volba padla na open source implementaci XML s názvem Apache XML Project (Xerces-C), což je výkonná, hojně využívaná softwarová knihovna zapouzdřující danou technologii.

3.4.2 Návrh a tvorba grafického uživatelského rozhraní

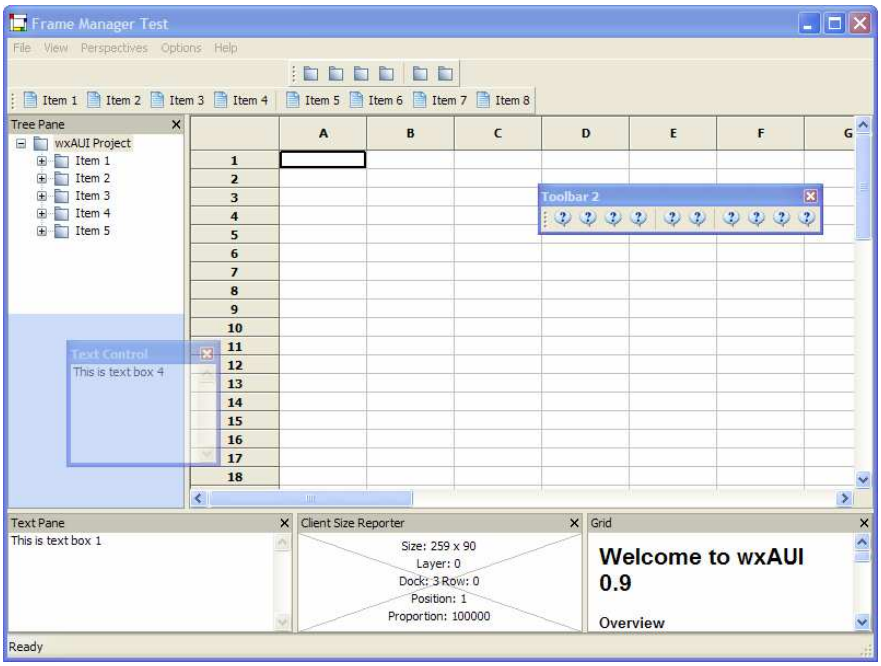
Tvorba uživatelského rozhraní aplikace pomocí knihovny wxWidgets může probíhat různými způsoby. Ať už čistě programově, kdy programátor sám vytváří instance tříd zapouzdřujících jednotlivé ovládací prvky a navzájem je provazuje, nebo vizuálním způsobem, za použití některého (ať už komerčního, či volně šiřitelného) grafického WYSIWYG editoru GUI pro wxWidgets. Knihovna wxWidgets rovněž umožňuje použití tzv. XML souborů zdrojů aplikace (XRC soubory, XML Ressource), které slouží k oddělení vizuální stránky aplikací od samotné programové logiky; veškeré vizuální ovládací prvky aplikace jsou popsány XML strukturami a lze je z tohoto popisu dynamicky vytvářet (a dokonce i modifikovat) za chodu aplikace.



Obr. 25 Prostředí aplikace wxFormBuilder

Mezi nejznámější editory GUI pro wxWidgets patří například komerční aplikace wxDesigner [32] či DialogBlocks [33], nebo volně šiřitelé alternativy wxGlade [34], VisualWx [35], wxDev-Cpp [36], wxFormBuilder a mnohé další.

Při tvorbě tříd dialogových oken aplikace SB byla použita právě aplikace wxFormBuilder (v době tvorby aplikace SB jeden z nejpropracovanějších vizuálních editorů zdrojů pro wxWidgets; navíc volně šířitelný a použitelný). Grafické rozhraní samotného hlavního rámcového okna aplikace SB bylo navrženo a vytvořeno pomocí technologie umožňující uživatelskou konfiguraci jednotlivých ovládacích prvků přímo za běhu programu. Jednalo se o rozšiřující technologii knihovny třídy wxWidgets s názvem wxAUI (Advanced User Interface), která poskytuje třídy vhodné pro tvorbu ovládacích prvků, oken a panelů nástrojů, u nichž lze přímo za běhu programu interaktivně měnit jejich umístění a velikost.



Obr. 26 Vzorová aplikace demonstrující možnosti wxAUI [37]

Výše uvedený obrázek 26 ilustruje na příkladu vzorové aplikace dodávané společně se zdrojovými kódy knihovny wxAUI možnosti tohoto volitelného rozšíření. (pozn.: V době tvorby aplikace SB bylo wxAUI samostatným, volitelným rozšířením knihovny wxWidgets. V současné době (verze 2.8.x) je již wxAUI distribuováno jako integrovaná součást knihovny wxWidgets.)

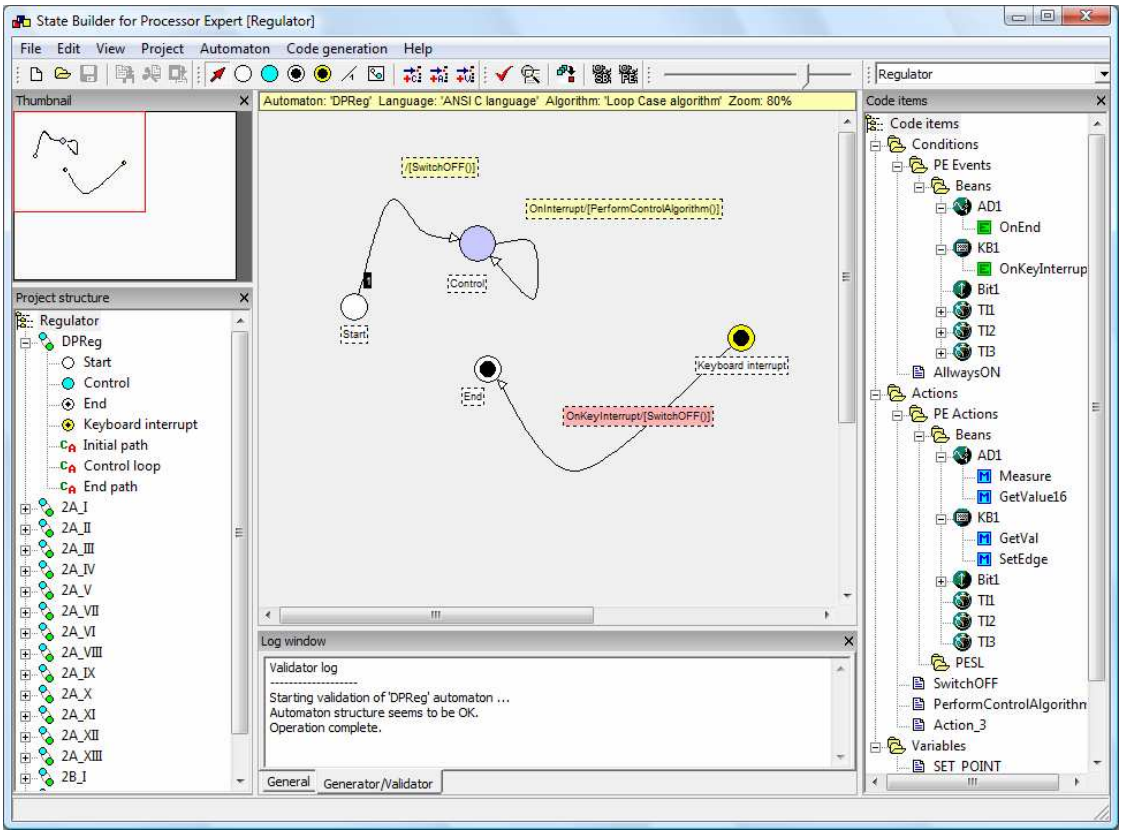
3.4.2.1 Uživatelské rozhraní hlavního rámcového okna

Jak již bylo řečeno, hlavní rámcové okno aplikace SB bylo vytvořeno pomocí technologie umožňující uživatelsky definovat umístění a velikost jednotlivých panelů ovládacích prvků. Vzhledem k hostitelské aplikaci PE je hlavní okno SB samostatným oknem, provázaným pouze prostřednictvím aplikačního rozhraní. Všechny prvky poskytované aplikací PE mající z hlediska uživatele význam v prostředí aplikace SB, jsou importovány a zobrazeny v jednom z hlavních panelů aplikace SB. Jedná se především o metody a události beanů zapouzdřujících funkcionalitu jednotlivých HW periferií cílového embedded zařízení (viz panel programových komponent).

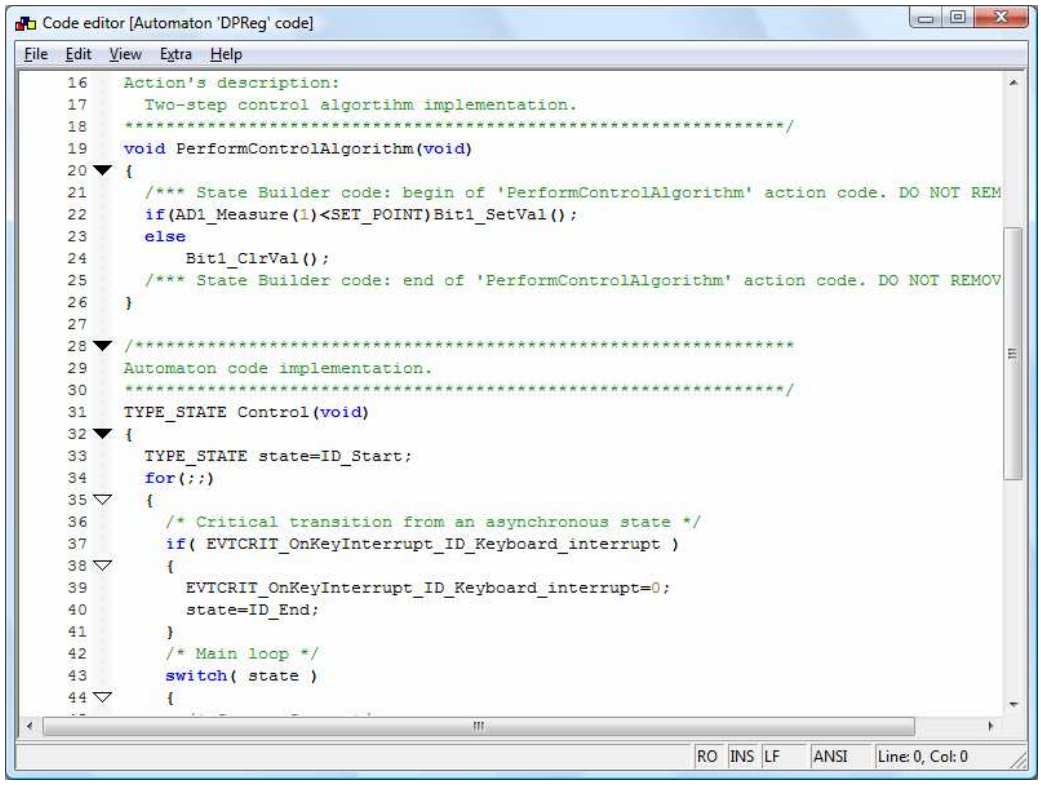
Hlavní rámcové okno aplikace SB se skládá z hlavního menu, panelů nástrojů, stavového řádku a několika panelů ovládacích prvků (náhled pracovní plochy, komponenty SB projektu/automatu, pracovní plocha, okno zpráv, komponenty PE projektu). Účel jednotlivých ovládacích prvků bude podrobně vysvětlen v kapitolách zabývajících se popisem a ovládáním aplikace SB.

Dalším z hlavních, vícenásobně použitých, prvků aplikace SB, je také integrovaný editor zdrojových kódů. Ten je využit jak pro prohlížení generovaného zdrojového kódu, tak pro editaci uživatelem definovaných kódových fragmentů (uživatelských datových typů, zdrojového kódu uživatelských akcí a podmínek přechodů).

Jak je vidět z obrázku 28, integrovaný textový editor poskytuje ovládací prvky a vlastnosti používané v moderních vývojových prostředích. Jedná se zejména o možnost zvýrazňování syntaxe zdrojového kódu, zobrazování/skrývání částí textu, číslování řádků, změnu kódování dokumentu, vyhledávání a nahrazování textu a mnohé další.



Obr. 27 Hlavní rámcové okno aplikace State Builder



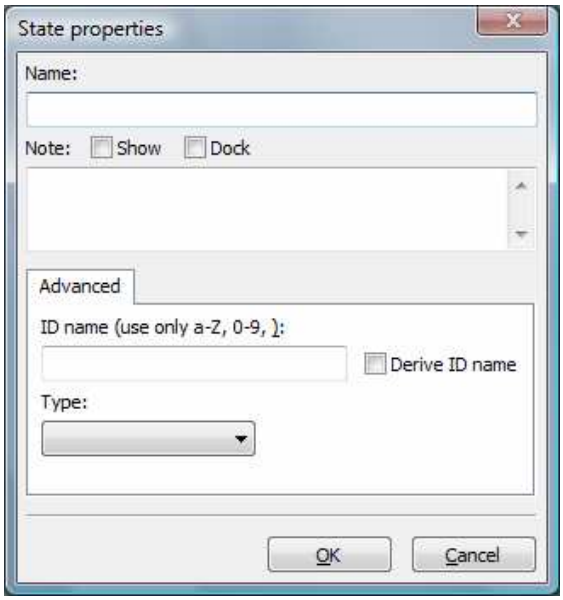
Obr. 28 Integrovaný editor zdrojových kódů

3.4.2.2 Uživatelské rozhraní dialogových oken

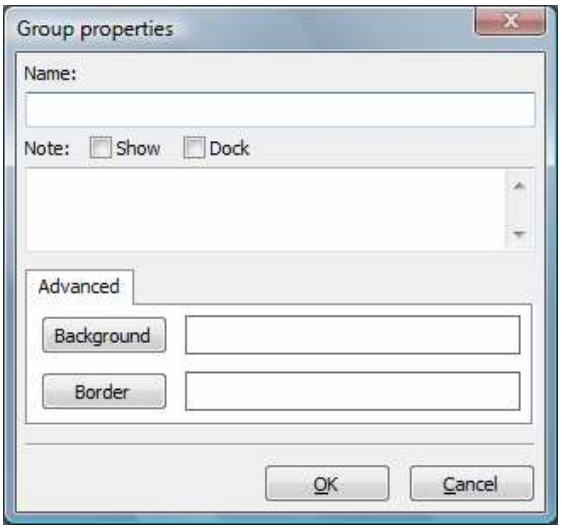
Kromě hlavního rámcového okna je rozhraní mezi aplikací SB a jejím uživatelem tvořeno sadou dialogových oken umožňujících jejich vzájemnou interakci.

Všechna dialogová okna byla nejprve vizuálně navržena v prostředí aplikace wxFormBuilder, poté byl vygenerován XML soubor zdrojů pro knihovnu wxWidgets a ten byl použit pro dynamickou tvorbu dialogových oken přímo za běhu aplikace. Tento způsob realizace uživatelského rozhraní má tu výhodu, že je jednoznačně oddělena logika aplikace od její prezentační vrstvy, čímž se lze vyvarovat některým problémům vznikajícím při tvorbě a pozdější údržbě zdrojového kódu aplikace. Navíc není nutné při změnách rozvržení a vlastností ovládacích prvků umístěných na daných dialogových oknech znovu překládat celou aplikaci; změní se pouze obsah XML dokumentu souboru zdrojů, což se

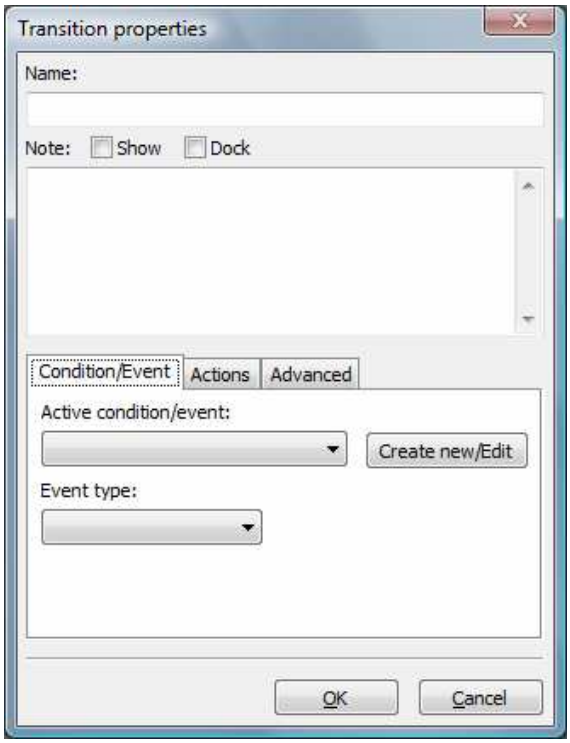
projeví okamžitě při dalším načtení dialogu ze souboru zdrojů. Použitá dialogová okna aplikace SB jsou následující:



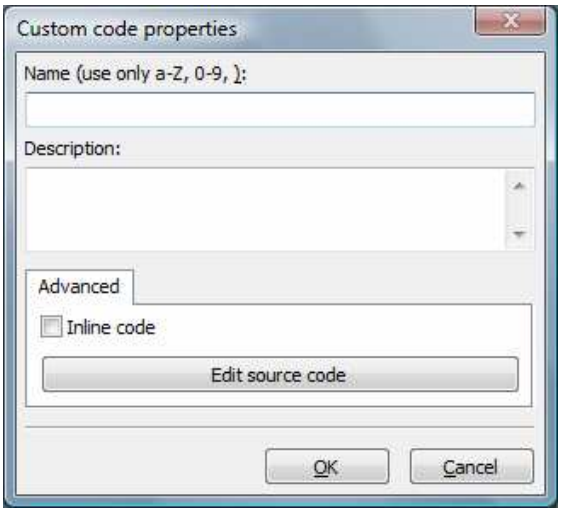
Obr. 29 Návrh dialogového okna „Vlastnosti stavu“



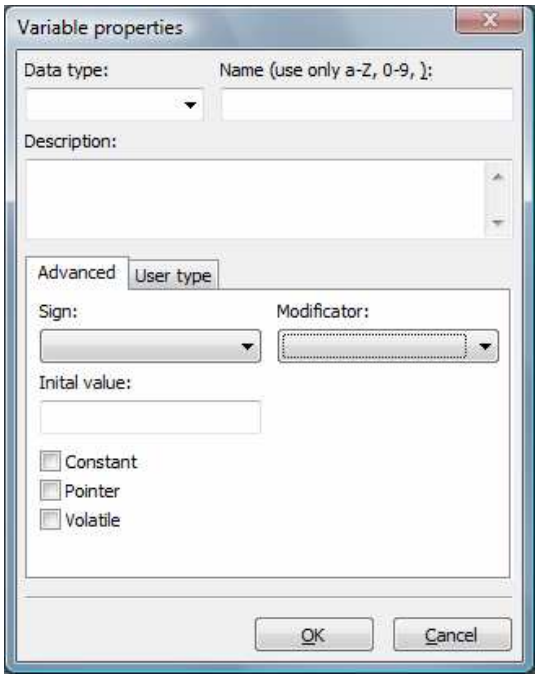
Obr. 30 Návrh dialogového okna „Vlastnosti skupiny stavů“



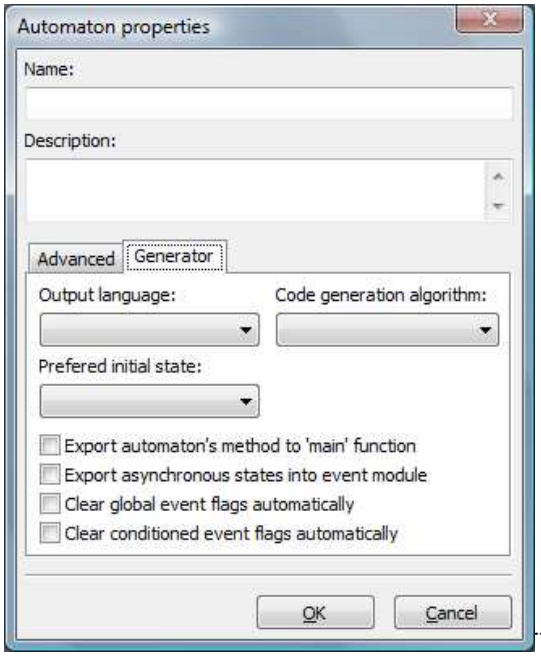
Obr. 31 Návrh dialogového okna „Vlastnosti hrany přechodu“



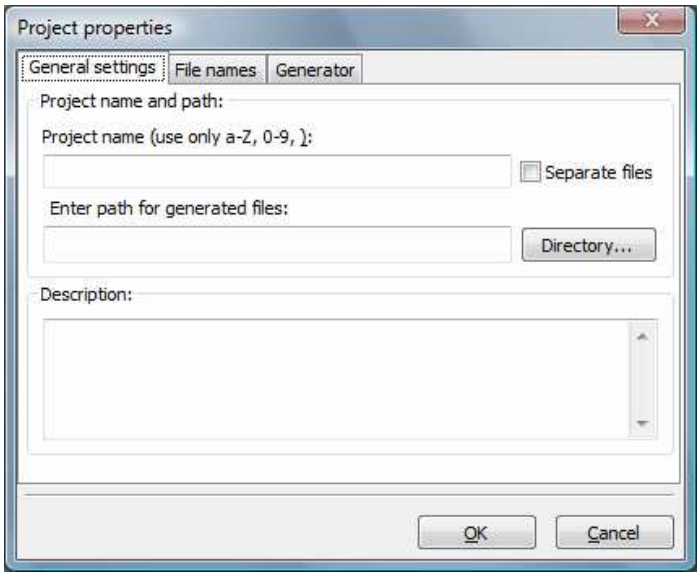
Obr. 32 Návrh dialogového okna „Vlastnosti fragmentu zdrojového kódu“



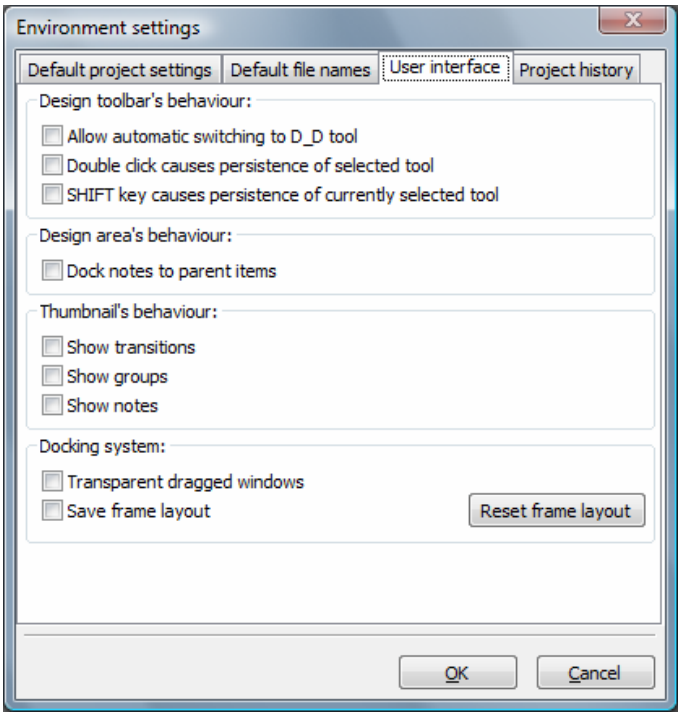
Obr. 33 Návrh dialogového okna „Vlastnosti globální proměnné“



Obr. 34 Návrh dialogového okna „Vlastnosti automatu“



Obr. 35 Návrh dialogového okna „Vlastnosti projektu“



Obr. 36 Návrh dialogového okna „Vlastnosti aplikace“

3.4.3 Implementace programového rozhraní API

Komunikace mezi hostující aplikací PE a nově vytvořenými nástroji je zajištěna prostřednictvím COM rozhraní aplikace PE. To umožňuje použití pluginů (DLL souborů), které jsou načítány při startu prostředí PE a mohou být spouštěny automaticky, nebo prostřednictvím konfigurovatelného menu aplikace PE (toto chování lze jednoduše ovlivňovat pomocí nastavení uložené v konfiguračním XML souboru pluginu).

```
<PE_Plugin>
  <Plugin name="State Builder" version="1.00" info="" author="MB"/>
  <Module name="StateBuilder\StateBuilder.dll"/>
  <WEB URL="http://www.processorexpert.com/support"/>
  <Menu Item="State Builder for Processor Expert"/>
</PE_Plugin>
```

Obr. 37 Obsah konfiguračního soubor pluginu aplikace PE

API prostředí PE je tvořeno COM rozhraním na těchto úrovních:

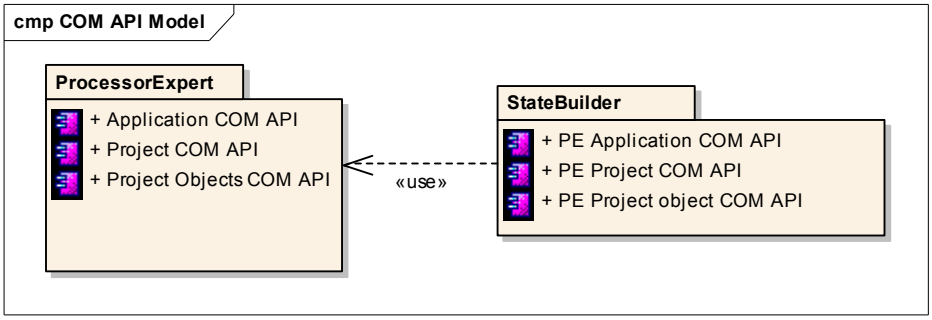
- Aplikace PE (funkce pro celé PE)
- Projektu (funkce pro projekt)
- Objektu v projektu (funkce pro objekty v projektu)

Druhé dvě úrovně mohou existovat ve více instancích (více otevřených projektů resp. více objektů v projektu). Pro každou úroveň jsou definovány:

- Statické funkce, které mohou být volány kdykoliv během práce
- Události, kterými je externí aplikace informována o změnách
- Seznamy uchovávající informace o projektech/objektech.

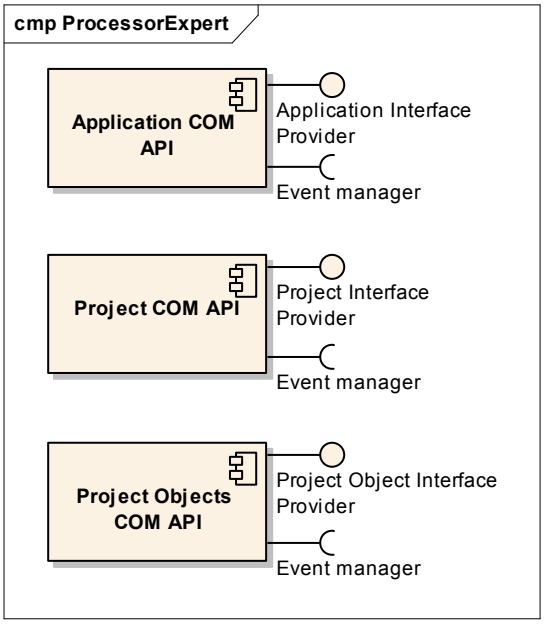
Data jsou přenášena (mimo atomických hodnot) prostřednictvím XML, které je jednoduše rozšiřitelné.

Následující obrázek představuje zjednodušený model komunikačního rozhraní mezi aplikacemi PE a SB.



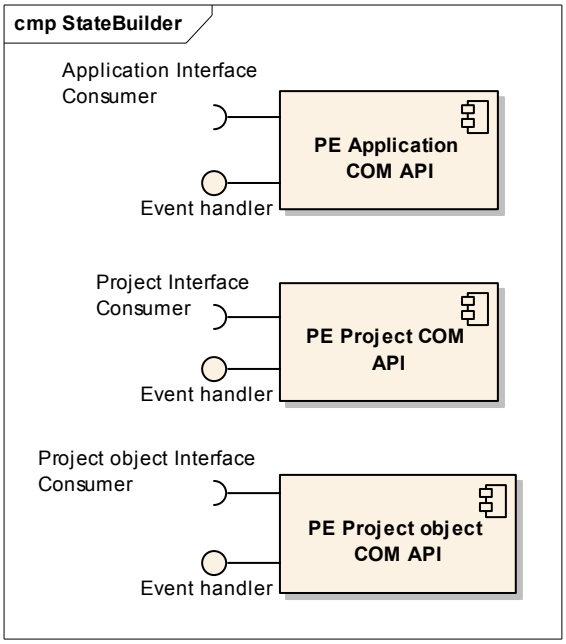
Obr. 38 Model komunikačního rozhraní

Všechny 3 úrovně komunikačního rozhraní poskytují na straně PE jak objekty schopné řídit funkcionalitu PE, tak také abstraktní třídy rozhraní, na základě nichž lze vytvořit třídy obsahující obsluhy událostí vznikajících v prostředí PE. Komponenty komunikačního rozhraní implementované na straně aplikace PE jsou zobrazeny v následujícím schématu.



Obr. 39 Komponenty COM rozhraní na straně aplikace PE

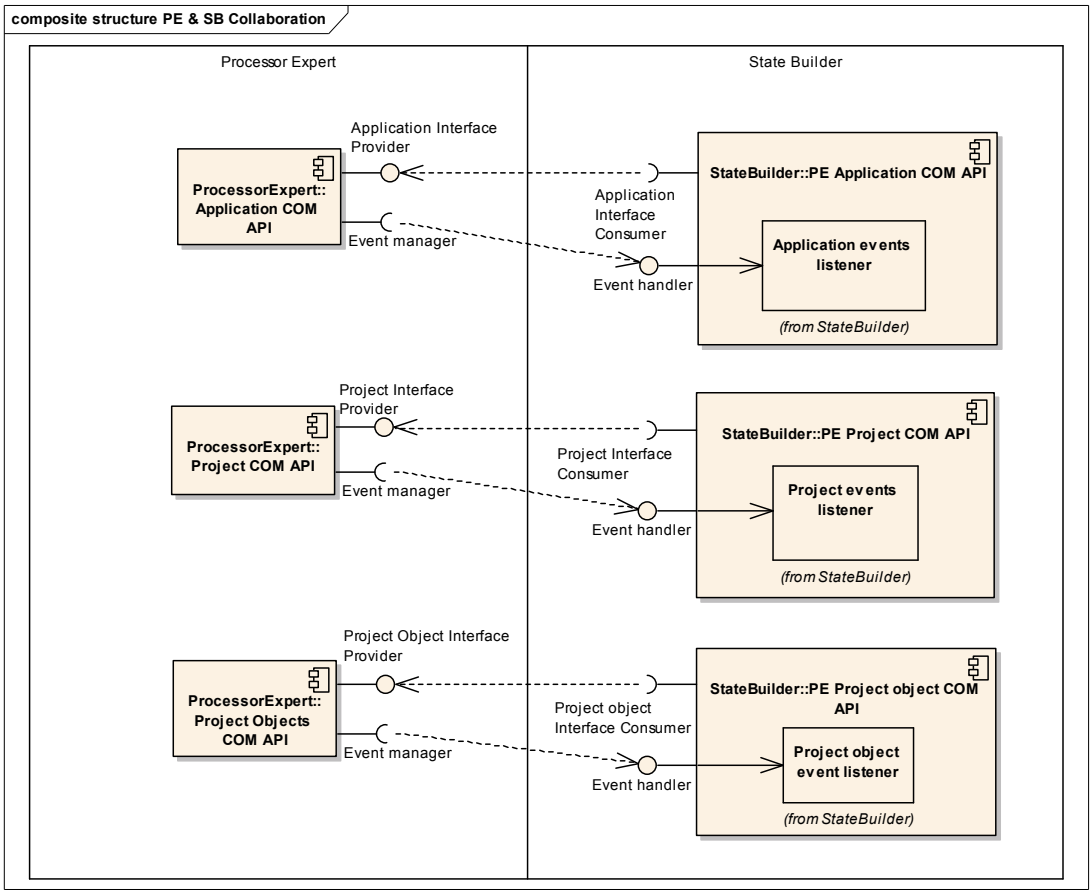
Na straně pluginu pak může uživatel libovolně využívat kterékoliv z poskytovaných rozhraní a je pouze na něm, do jaké míry bude jejich funkcionalitu ve své aplikaci využívat. Implementace aplikačního rozhraní na straně SB využívá všech tří definovaných rozhraní a to jak pro účely řízení aplikace PE, tak i obsluhy jejích událostí. Implementované komponenty COM rozhraní na straně aplikace SB jsou uvedeny na obrázku 40.



Obr. 40 Komponenty COM rozhraní na straně aplikace SB

K provázání jednotlivých komunikačních rozhraní nedochází v jeden specifický okamžik; tento proces závisí na stavu aplikací a jejich projektů. Komunikační rozhraní na úrovni aplikací je provázáno bezprostředně po zavedení pluginu do prostředí aplikace PE. K provázání projektů aplikace PE a SB dochází po vytvoření nového PE projektu, popř. po otevření již existujícího; v tomto případě nezáleží na tom, v které aplikaci byl daný projekt otevřen. Rozhraní objektů PE projektu je pak inicializováno v okamžiku inicializace projektu aplikace SB. Podrobněji je způsob inicializace komunikačních rozhraní vysvětlen v následujících kapitolách.

Vzájemné vazby všech komunikačních rozhraní jsou zobrazeny na následujícím schématu.



Obr. 41 Schéma plně inicializovaného aplikačního rozhraní mezi PE a SB

V následující kapitole si přiblížíme důležité aspekty implementace komunikačního rozhraní mezi aplikacemi PE a SB.

3.4.3.1 Inicializace komunikačního rozhraní

Každý rozšiřující modul (plugin) aplikace PE musí zveřejňovat tři speciální programové funkce, které budou ve vhodnou chvíli volány z prostředí PE a pomocí nichž bude zajištěna **identifikace**, **inicializace** a **deinicializace** pluginu.

Funkcí zajišťující **identifikaci** pluginu je funkce

```
char* initPlugin(void);
```

Tato funkce je volána při pokusu o zavedení pluginu do prostředí PE (což se děje ihned po startu aplikace PE) a jejím úkolem je poskytnou aplikaci PE informace potřebné pro korektní načtení pluginu. Informace je předávána ve formě textového řetězce obsahující XML strukturu popisující daný plugin. Obsah identifikačního řetězce může být např. následující:

```
<PE_Plugin>
  <Plugin name="State Builder" version="1.23" info="for Motorola
HCS12" author="UNIS"/>
  <PE_API version="1.00"/>
</PE_Plugin>
```

Obr. 42 Identifikační řetězec pluginu

Po zavedení pluginu aplikace SB do prostředí PE je možné daný plugin spustit (např. prostřednictvím přiřazené položky menu aplikace PE). Při jeho spouštění je volána exportovaná funkce zajišťující jeho **inicializaci**. Deklarace této funkce je:

```
void runPlugin(IPEApplication* AppInterface, HWND
AppHandle);
```

Předávané parametry přestavují ukazatel na **objekt komunikačního rozhraní aplikace PE** a její handle. Objekt aplikace je již součástí vlastního COM rozhraní PE a obsahuje metody, které mohou být volány z prostředí pluginu. Jednou z těchto metod je i funkce určená pro registraci obslužných rutin událostí aplikace PE. V rámci funkce `runPlugin()` tedy dochází k provázání komunikačního rozhraní na úrovni aplikací. Jednotlivé metody a události publikované tímto rozhraním budou popsány dále.

V tomto místě je potřeba si uvědomit jednu zásadní věc, která bude dále ovlivňovat způsob práce k COM rozhraním aplikace PE. Funkce `runPlugin()` je volána

v separátním vlákně aplikace PE a prakticky představuje vstupní bod pluginu (obdobu hlavní funkce `main()` v programu vytvořeném pomocí jazyce C/C++). Daný plugin je tedy aktivní po dobu života této funkce; po opuštění těla funkce dochází také k ukončení činnosti pluginu na straně PE. Taky je potřeba brát zřetel na fakt, že veškeré volání funkcí COM rozhraní z aplikace PE (tedy i vyvolání událostí, které mohou být na straně SB obslouženy přetížením patřičných virtuálních funkcí příslušné abstraktní třídy aplikačního rozhraní) je prováděno z kontextu vlákna aplikace PE, kdežto samotný plugin (v našem případě aplikace SB) běží v kontextu svého vlastního vlákna. Z tohoto důvodu je potřeba dodržovat přísná pravidla při práci se sdílenými datovými zdroji, nebo se systémovými voláními, která nejsou vláknově bezpečná (např. některé nativní funkce WIN32API určené pro práci s vizuálními ovládacími prvky aplikace). V takovém případě je potřeba zajistit, aby kritické operace **nebyly** prováděny přímo v těle přetížených virtuálních funkcí objektů aplikačního rozhraní, ale až v kontextu vlákna pluginu. To lze zajistit například tak, že v daných obslužných funkcích událostí PE budou pouze generovány zprávy popisující dané události, a tyto zprávy pak budou standardním způsobem zpracovávány v rámci systému zpráv pluginu.

Poslední zmiňovanou funkcí umožňující korektní **deinicializaci** pluginu je funkce

```
HRESULT dllCanUnload(void);
```

Tato funkce je volána aplikací PE v okamžiku, kdy PE požaduje ukončení práce pluginu a jeho fyzické uvolnění z paměti. Návratová hodnota potom PE informuje o tom, zda je bezpečný daný plugin uvolnit.

3.4.3.2 COM rozhraní aplikace PE

COM rozhraní na úrovni samotné aplikace PE je tvořeno sadou metod, které mohou být volány nad objektem rozhraní předaným funkcí `runPlugin()` a třídou obsluhy událostí s názvem `IPEApplicationEvents`. Tato čistě abstraktní třída slouží jako základ pro vytvoření třídy určené pro obsluhu událostí aplikace PE. Obsahuje jednak deklarace standardních funkcí pro počítání referencí COM objektů a také sadu

přetížitelných virtuálních funkcí určených pro obsluhu jednotlivých událostí aplikace PE. Objekt třídy obsluh událostí je předáván aplikaci PE prostřednictvím speciální funkce `setEventHandler()` a jeho odebrání ze správce událostí lze provést voláním funkce `releaseEventHandler()`. Obě tyto funkce jsou volány nad objektem COM rozhraní aplikace.

Metody publikované COM rozhraním aplikace PE jsou následující:

`void close(void)`

Ukončí činnost IDE

`IPEProject* openProject(LPSTR FilePath);`

Otevře project s názvem FilePath v PE. Vrací instanci projektu IPEProject

`VARIANT* ID(void)`

Vrací identifikaci PE.

`IPEProjectList* projects(void)`

Vrací seznam projektů otevřených v PE.

`void setEventHandler(IPEApplicationEvents* EventHandler)`

Nastaví handler pro správu událostí z aplikace PE.

`HRESULT releaseEventHandler(IPEApplicationEvents* EventHandler)`

Vyjme tuto událost ze správy událostí z aplikace PE. Vrací úspěch operace.

`IPEProject getActiveProject(VARIANT* FileName)`

Získá aktivní projekt a vrátí cestu a název .pe. Pokud není otevřený žádný projekt vrací null a FileName je rovněž null. Nenulový FileName je nutné uvolnit z paměti přes releaseBuffer. Pokud projekt není uložený, je FileName null.

`void setIDEvisible(byte Visible);`

Pokud je parameter Visible nenulový, tak se zobrazí celé PE IDE, v opačném případě se IDE skryje.

Události deklarované třídou `IPEApplicationEvents` jsou tyto:

`void projectOpened(IPEProject* Project);`

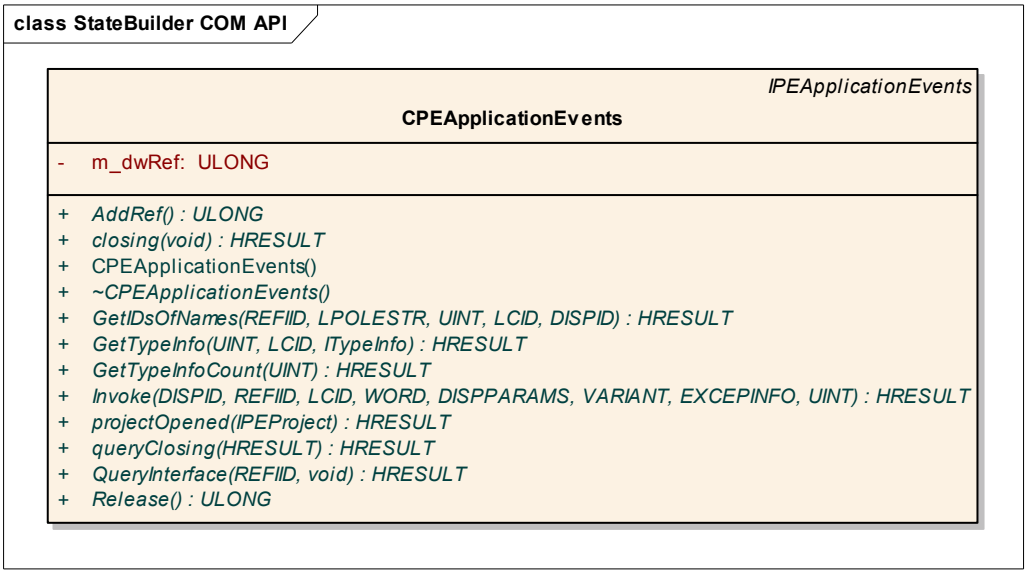
Informace o otevření projektu (po té, co byl projekt načten).

`HRESULT queryClosing(IPEProject* Project);`

Informace před zavřením IDE (návratová hodnota E_FAIL může zavírání IDE zastavit).

`void closing(IPEProject* Project);`

Informace o zavření IDE (těsně před zavřením).



Obr. 43 Diagram implementace třídy událostí aplikace PE v pluginu SB

Aplikace SB využívá pouze některé deklarované události aplikace PE. Rozsah implementace (třída `CPEApplicationEvents`) je zobrazen na diagramu v obrázku 43.

3.4.3.3 COM rozhraní projektu PE

Aplikace PE umožňuje, aby bylo v rámci prostředí PE v jednom okamžiku otevřeno více projektů. Plugin aplikace PE může prostřednicím funkcí COM rozhraní aplikace získat objekty COM rozhraní těchto projektů, nad nimiž pak lze, stejně jako tomu bylo v případě objektu COM rozhraní aplikace, volat sadu funkcí určených pro řízení projektu, či obsluhovat události v projektu vygenerované (události projektu jsou zapouzdřeny abstraktní třídou `IPEProjectEvents`). Stejně jako tomu bylo v případě rozhraní aplikace, i rozhraní projektu obsahuje funkce `setEventHandler()` a `releaseEventHandler()` určené v registraci třídy obslužných rutin událostí vznikajících v PE projetku. Metody publikované COM rozhraním projektu jsou následující:

`HRESULT close(byte SaveProject);`

Uzavře projekt v PE, parametr určuje, zda-li s má projekt uložit. Vrací úspěch operace.

`HRESULT save(void);`

Uloží projekt PE. Vrací úspěch operace.

`HRESULT saveAs(LPSTR* FileName);`

Uloží projekt PE pod názvem *FileName*. Vrací úspěch operace.

`void setEventHandler(IPEProjectEvents* EventHandler);`

Nastaví handler pro správu událostí z PE projetku.

`HRESULT releaseEventHandler(IPEProjectEvents* EventHandler);`

Odstraní handler pro správu událostí z PE projetku. Vrací úspěch operace.

`VARIANT* ID(void);`

Vrátí ID projektu.

HRESULT generate(VARIANT_BOOL ForceGen, VARIANT_BOOL WaitForEnd);

Vyvolá generování všech zdrojových souborů projektu PE. Úspěch generovaní je možné získat událostí **generated**. Vrací úspěch operace – neúspěch, když je frozen, nebo již generování právě probíhá. Je-li nastaveno ForceGen, spustí generování přestože projekt je již vygenerovaný a nedošlo k žádné změně. Když je nastaveno WaitForEnd, tak se čeká na konec generování, jinak generování proběhne na pozadí. Pokud je možné provést generování, tak se výsledek vrací v události generated.

VARIANT* content(TProjectObjectType ObjectType)

Vrací seznam objektů PE projektu. Parametr ObjectType umožňuje filtrovat množství přenesených dat; přenáší se jen základní informace o objektech zadaného typu.

void activate(void);

Nastaví daný project jako aktivní projekt v PE.

IPEProjectObject getProjectObject(TProjectObjectType ObjectType, LPSTR ID);

Vrátí objekt projetu daný typem a ID.

VARIANT* addProjectObject(TProjectObjectType ObjectType, LPSTR content);

Přidá do projektu objekt a vrátí jeho ID. Pokud je ID rovno "-1", nelze objekt přidat. Formát obsahu je následující

HRESULT removeProjectObject(TProjectObjectType ObjectType, LPSTR ID);

Odstraní z projektu objekt s ID. (ID je řetězec obsahující číslo - textově). Typ objektu ObjectType musí být specifikováno. Vrací úspěch operace.

`void setBIvisible(LPSTR BeanName, byte Visible);`

Pokud je parameter *Visible* nenulový, tak se zobrazí Bean Inspector pro bean *BeanName*. Bean musí v daném projektu existovat, jinak se nezobrazí nic. Pokud je Bean Inspector již zobrazen, nastaví se v něm daný bean. Pokud je *Visible* = 0 tak se parametr *BeanName* ignoruje a Bean Inspector se skryje.

`TProjectStatus checkProject();`

Explicitně zjistí stav projektu před generováním.

`void activate(void);`

Nastaví daný project jako aktivní projekt v PE.

Události deklarované třídou `IPEProjectEvents` jsou následující:

`void activated(IPEProject* Project);`

Informace o aktivaci projektu.

`void saved(IPEProject* Project);`

Informace o ukládání projektu na disk (před tím, než je uložen).

`HRESULT queryClosing(IPEProject* Project);`

Informace před zavřením projektu (návratová hodnota E_FAIL může zavírání projektu zastavit).

`HRESULT queryGenerate(IPEProject* Project);`

Informace o zahájení generování projektu (před tím, než se generuje).

`void closing(IPEProject* Project);`

Informace o zavření projektu (před tím, než je uzavřen).

`HRESULT generated(IPEProject* Project);`

Informace o vygenerování projektu (po tom, co se vygeneruje). Vrací úspěšnost generování.

```
void objectAdded(IPEProject* Project,
TProjectObjectType ObjectType, VARIANT ID);
```

Informace o přídání objektu ID do projektu (po té, co byl přidán).

```
void objectDeleted(IPEProject* Project,
TProjectObjectType ObjectType, VARIANT ID);
```

Informace o vymazání objektu ID z projektu (po té, co byl smazán).

```
void changed(IPEProject* Project);
```

Informace o změně projektu.

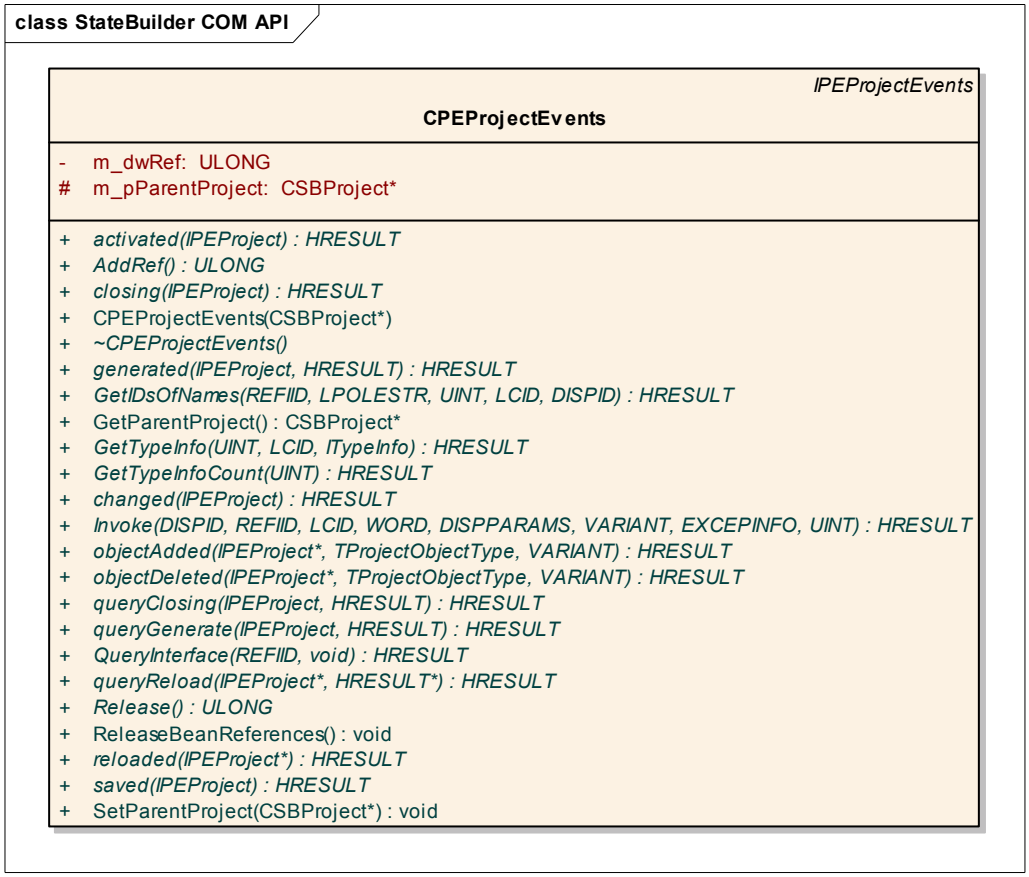
```
HRESULT queryReload(IPEProject* Project);
```

Informace před znovunačtením projektu (návratová hodnota E_FAIL může znovunačtení projektu zastavit).

```
void reloaded(IPEProject* Project);
```

Projekt byt znovu načten.

Implementace tohoto rozhraní (třída `CPEProjectEvents`) v aplikaci SB je znázorněna na obrázku 44.



Obr. 44 Diagram implementace třídy událostí projektu PE v pluginu SB

3.4.3.4 COM rozhraní objektu projektu PE

Posledním aplikačním rozhraním implementovaným v SB je rozhraní jednotlivých objektů v PE projektu. Toto rozhraní má význam tehdy, chceme-li bezprostředně reagovat na změnu vlastností objektů PE projektu provedenou uživatelem v prostředí aplikace PE.

V tom případě máme opět k dispozici sadu funkcí vhodných pro práci s objektem projektu a třídu obslužných rutin událostí. Pravidla pro práci s nimi jsou stejná, jako u obou výše uvedených rozhraní. Publikované metody rozhraní objektu PE projektu jsou:

VARIANT* *info*(LPSTR *Filter*);

Vrací zjednodušené informace o daném objektu v projektu. Parametr Filter umožňuje filtrovat množství přenesených dat.

void setEventHandler(IPEProjectObjectEvents* EventHandler);

Nastaví handler pro správu událostí z objektu PE projetku.

HRESULT releaseEventHandler(IPEProjectObjectEvents* EventHandler);

Odstraní handler pro správu událostí z objektu PE projetku. Vrací úspěch operace.

TProjectObjectType* ID(void);

Vrátí typ objektu.

VARIANT* content(LPSTR FilterExpression);

Vrací obsah Objektu projektu. Parametr FilterExpression umožňuje specifikovat filtr ovlivňující množství přenesených dat. Formát výrazu pro filtr viz kapitola **Chyba! Nenalezen zdroj odkazů.**.

VARIANT* setValue(VARIANT Item, VARIANT Value);

Nastaví položku Item na hodnotu Value. Item musí odpovídát názvu symbolu položky. Vrací skutečne nastavenou hodnotu, případně ###ITEM NOT FOUND#### v případě, že položka s daným symbolem nebyla nalezena. Pokud se vrácená hodnota liší od hodnoty Value, tak položku buď nelze nastavit (např. je read-only) a nebo PE neakceptoval novou hodnotu (např. díky typové kontrole – znaky v typu integer a podobně).

HRESULT removeItemFromList(VARIANT ItemSymbol);

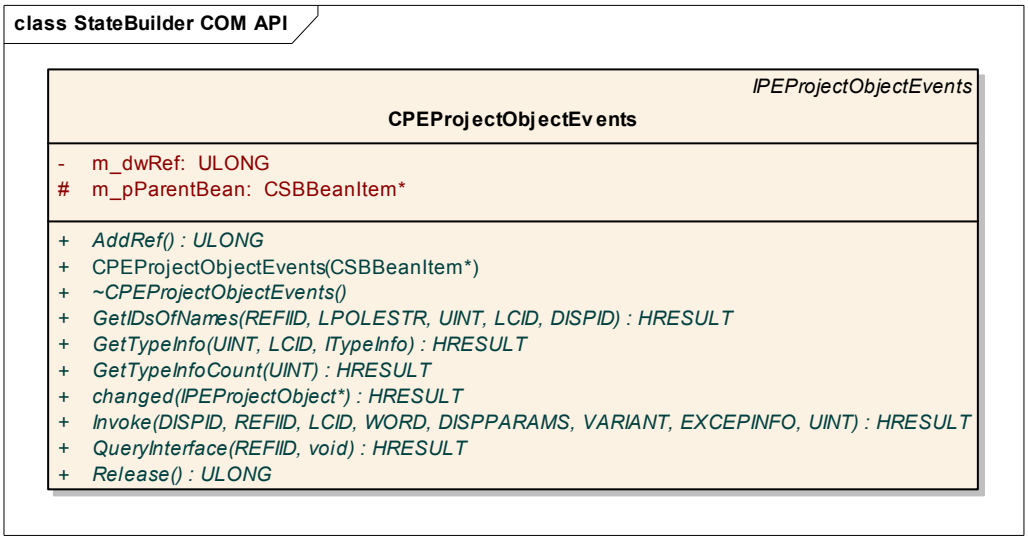
Vymaže položku z listu. Položka je identifikována symbolem ItemSymbol. Vrací S_OK pokud byla položka odstraněna, případně E_FAIL, pokud položka neexistuje, nebo pokud ji nelze odstranit.

Třída obsluhy událostí objektu v projektu poskytuje jednu přetížitelnou virtuální funkci:

```
void changed(IPEProjectObject* ProjectObject);
```

Informace o změně nastavení některé položky uživatelem.

Implementace této třídy na straně SB je tedy velmi jednoduchá:



Obr. 45 Diagram implementace třídy událostí objektu projektu PE v pluginu SB

3.4.4 Struktura aplikace State Builder

Aplikace State Builder je tvořena více než 80 třídami zapouzdřujícími její funkcionalitu. Obecně lze tyto jednotlivé třídy z hlediska jejich určení rozdělit do pěti kategorií:

- Třídy jádra aplikace
- Třídy grafického uživatelského prostředí (GUI)
- Třídy stavových diagramů

- Třídy aplikačního COM rozhraní
- Třídy generátoru programového kódu

Vzájemné vztahy mezi těmito kategoriemi jsou zobrazeny na obrázku 46.

3.4.4.1 Třídy jádra aplikace

Třídy jádra aplikace SB zobrazené na obrázku 47 mají za úkol zapouzdřit základní funkcionalitu aplikace a provázat její ostatní funkční celky. Mezi tyto třídy patří hlavní třída aplikace obsahující smyčku zpráv a další funkce pro inicializaci a deinicializaci aplikace a komunikačního rozhraní, pomocná třída zapouzdřující XML parser a generátor a třídy zapouzdřující projekt a jeho serializační a deserializační schopnosti.

Hlavní třída aplikace SB je vytvářena staticky ihned po načtení pluginu do paměti a je inicializována z veřejné funkce pluginu s názvem `runPlugin()` (viz kapitola 3.4.3.1). V rámci inicializace aplikace je načítáno její nastavení a poté je vytvořeno a inicializováno GUI aplikace.

Po úspěšné inicializaci GUI je řízení programu předáno hlavní smyčce zpráv aplikace SB, čím dojde k jejímu oživení.

3.4.4.2 Třídy grafického uživatelského prostředí (GUI)

Třídy grafického rozhraní zapouzdřují vizuální stránku aplikace a jsou zodpovědné za vytvoření a zajištění funkcionality hlavního rámcového okna programu a všech dialogových oken určených pro vzájemnou interakci aplikace a jejího uživatele. Jedná se o nejrozsáhlejší skupinu tříd aplikace a proto zde není z prostorových důvodů uveden digram jejich závislostí.

3.4.4.3 Třídy stavových diagramů

Jak už napovídá samotný název této skupiny, třídy stavových automatů zapouzdřují jak samotné komponenty stavového automatu (různé typy stavů, hrany přechodů, skupiny

stavů), tak také samotnou strukturu automatu. Do této skupiny spadají také třídy umožňující interaktivně konstruovat stavové diagramy.

Nejdůležitějšími třídami této skupiny je základní třída komponenty automatu (`CAutItem`), od níž jsou odvozeny třídy stavů (`CStateItem`), přechodů (`TransItem`) a dalších pomocných objektů (`CGroupItem`, `CTreeItem`), třída zapouzdřující strukturu automatu (`CAutomaton`) a třída pohledu na automat (`CAutomatonView`).

Vlastnosti a vzájemné vztahy těchto tříd jsou zobrazeny na obrázku 48.

3.4.4.4 Třídy aplikačního COM rozhraní

O třídách aplikačního komunikačního rozhraní mezi PE a SB již pojednávala kapitola 3.4.3, takže si zde pouze připomeneme, že se jednalo o třídy zapouzdřující obslužné rutiny událostí vznikajících v PE na úrovni aplikace (`CPEApplicationEvents`), PE projektu (`CPEProjectEvents`) a objektů PE projektu (`CPEProjectObjectEvents`).

Struktura těchto tříd je zobrazena v diagramu na obrázku 49.

3.4.4.5 Třídy generátoru programového kódu

Poslední hlavní skupinou tříd jsou třídy implementující samotný generátor programového kódu a třídy zajišťující, aby mohl proces generování vůbec proběhnout (tzn. validační a optimalizační třídy).

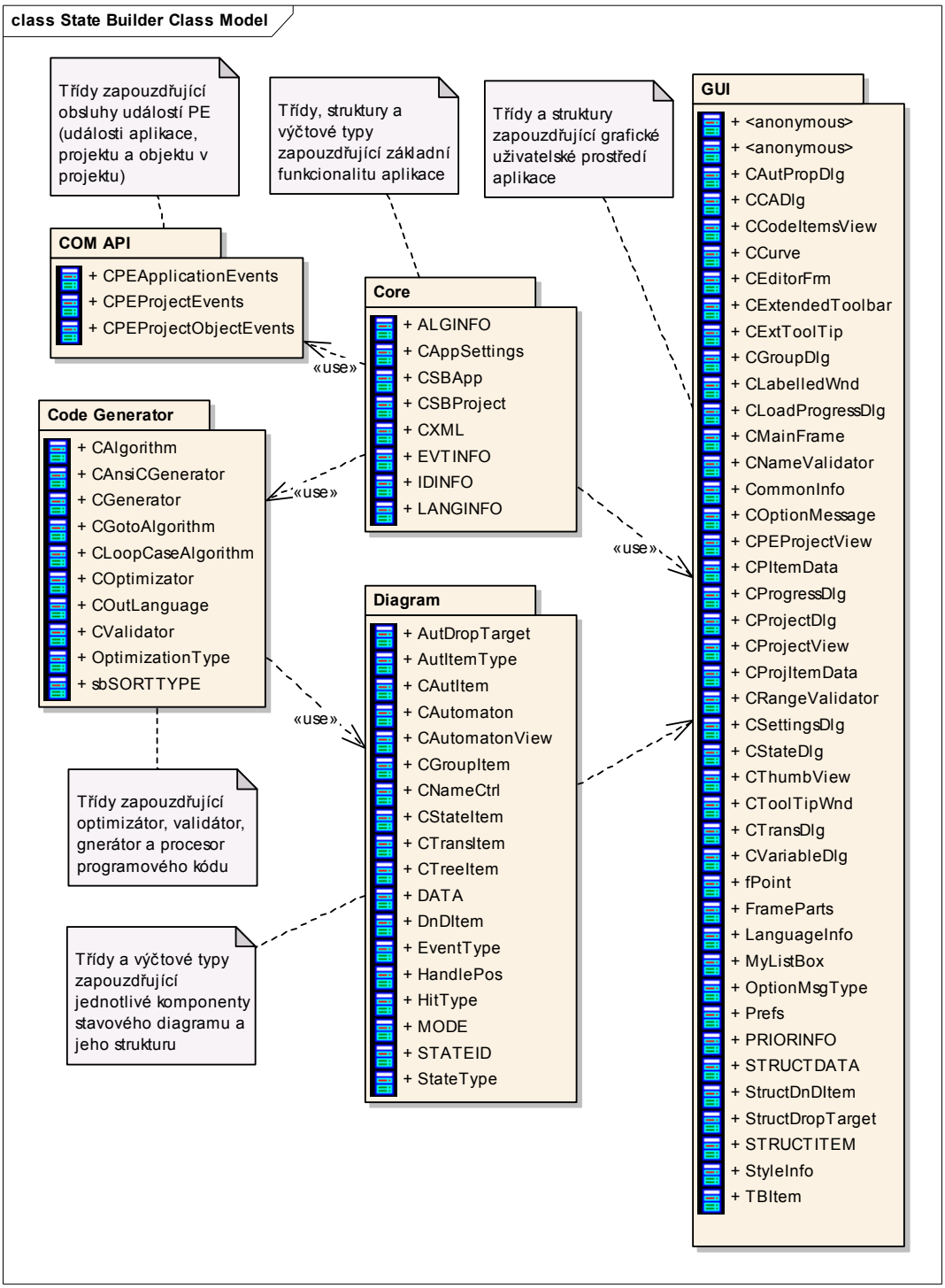
Před započetím vlastního generování výstupního programového kódu je nejprve ověřena formální správnost struktury stavového automatu. Tuto činnost provádí validátor automatu zapouzdřený třídou `CValidator`. Poté je (v závislosti na nastavení SB projektu) na struktuře automatu provedena sada minimalizačních operací (tyto operace jsou zapouzdřeny třídou `COptimizator`) a ověřený, minimalizovaný stavový automat pak slouží jako vstup do generátoru kódu zapouzdřeném třídou `CGenerator`. Je nutno

podotknout, že třída `CGenerator` je pouze základní, univerzální třídou generátoru, od níž jsou pak odvozeny jednotlivé generátory pro podporované programovací jazyky využívající příslušné procesory výstupního kódu. V současné době je implementován pouze generátor programovacího jazyka ANSI C zapouzdřený třídou `CAnsiCGenerator`. Univerzální procesor výstupního kódu je zapouzdřen třídou `COutLanguage` a využívá vlastního konfiguračního XML souboru pro definici konkrétního textového výstupu pro jednotlivé fragmenty programového kódu.

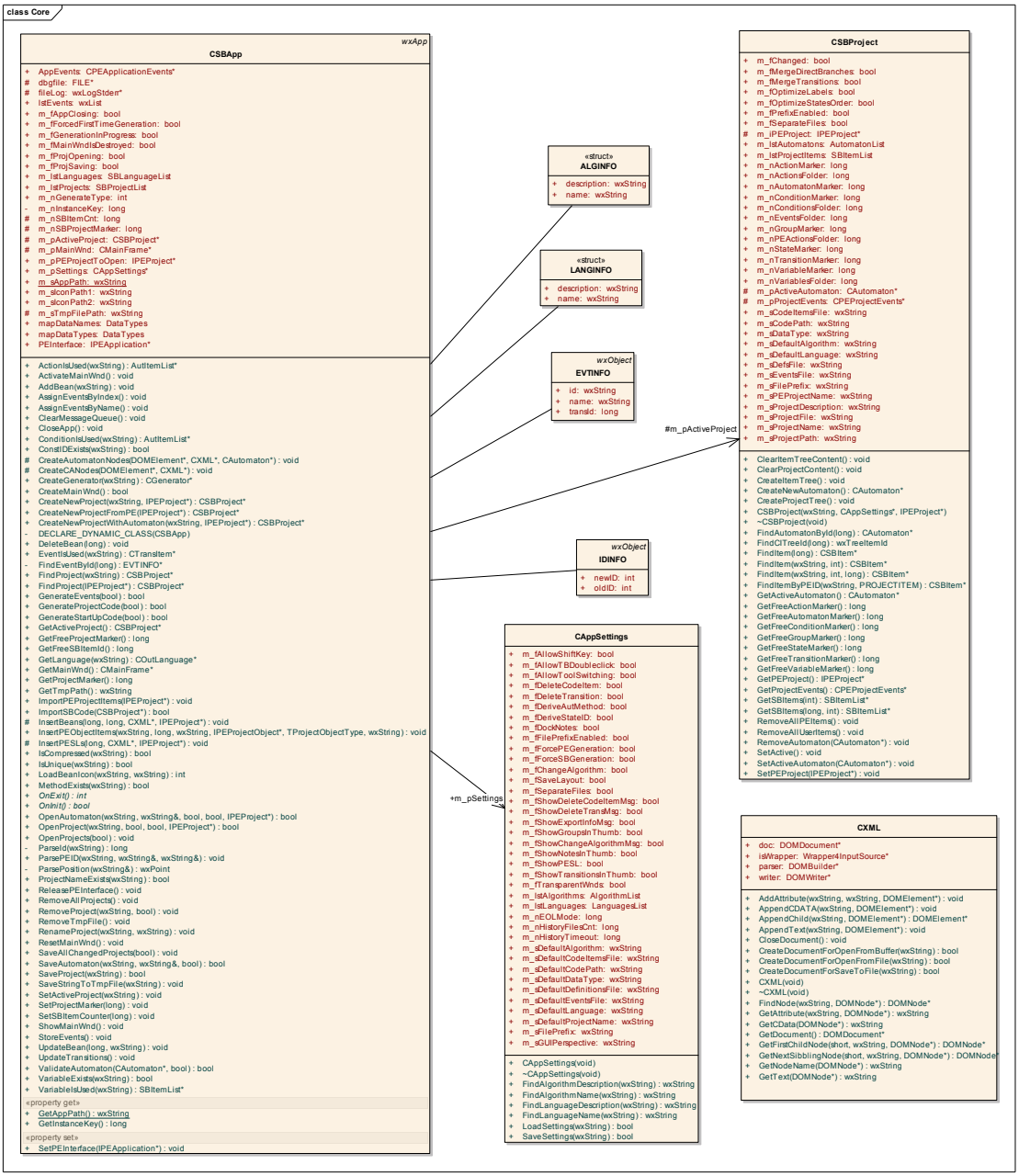
Programový výstup generátoru kódu závisí také na použitém generujícím algoritmu. Základní třídou pro takové algoritmy je třída `CAlgorithm`, od níž jsou dále odvozeny (a reálně používány) třídy algoritmů GOTO (`CGotoAlgorithm`) a Loop-Case (`CLoopCaseAlgorithm`). Vlastnosti obou generujících algoritmů jsou popsány v kapitolách 3.1.1.2 a 3.2.1.3.

Struktura a spolupráce tříd generátoru programového kódu je znázorněna na obrázku 50.

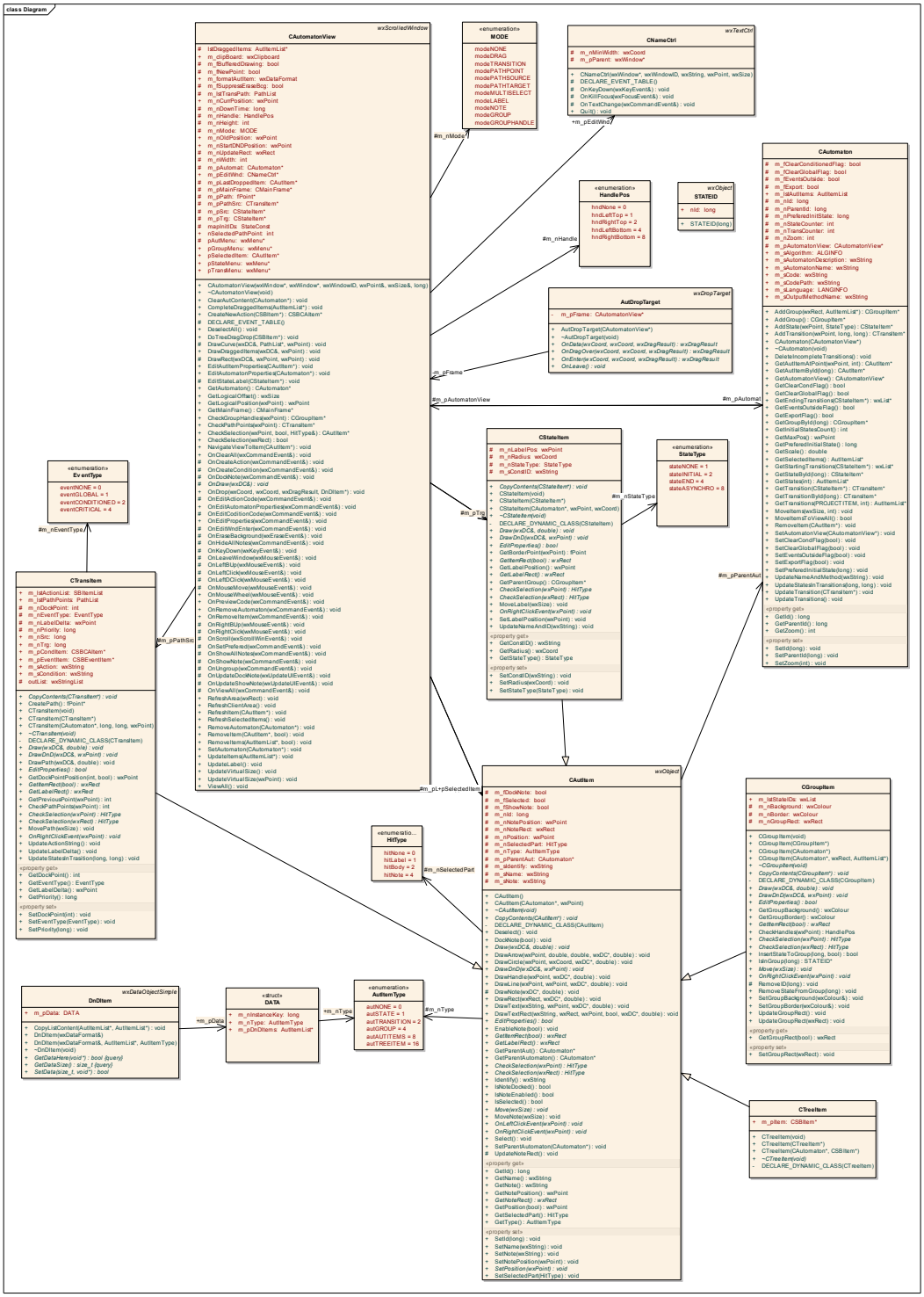
Nyní již následují všechny výše zmiňované obrázky a diagramy.



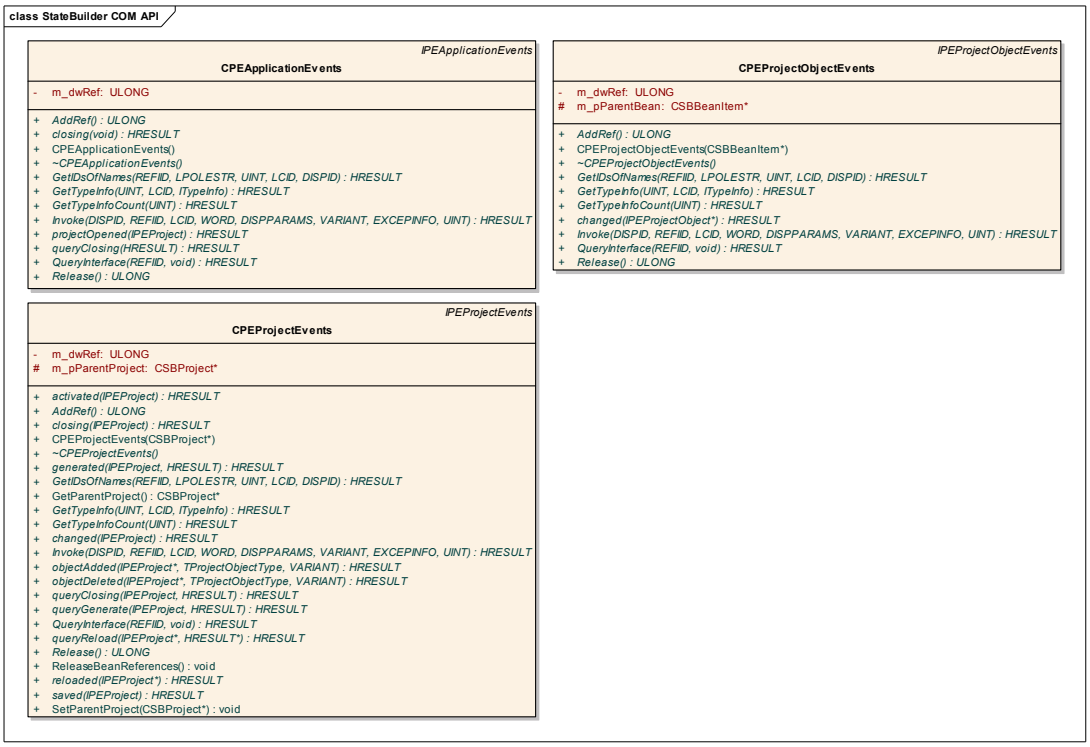
Obr. 46 Diagram seskupení tříd aplikace StateBuider



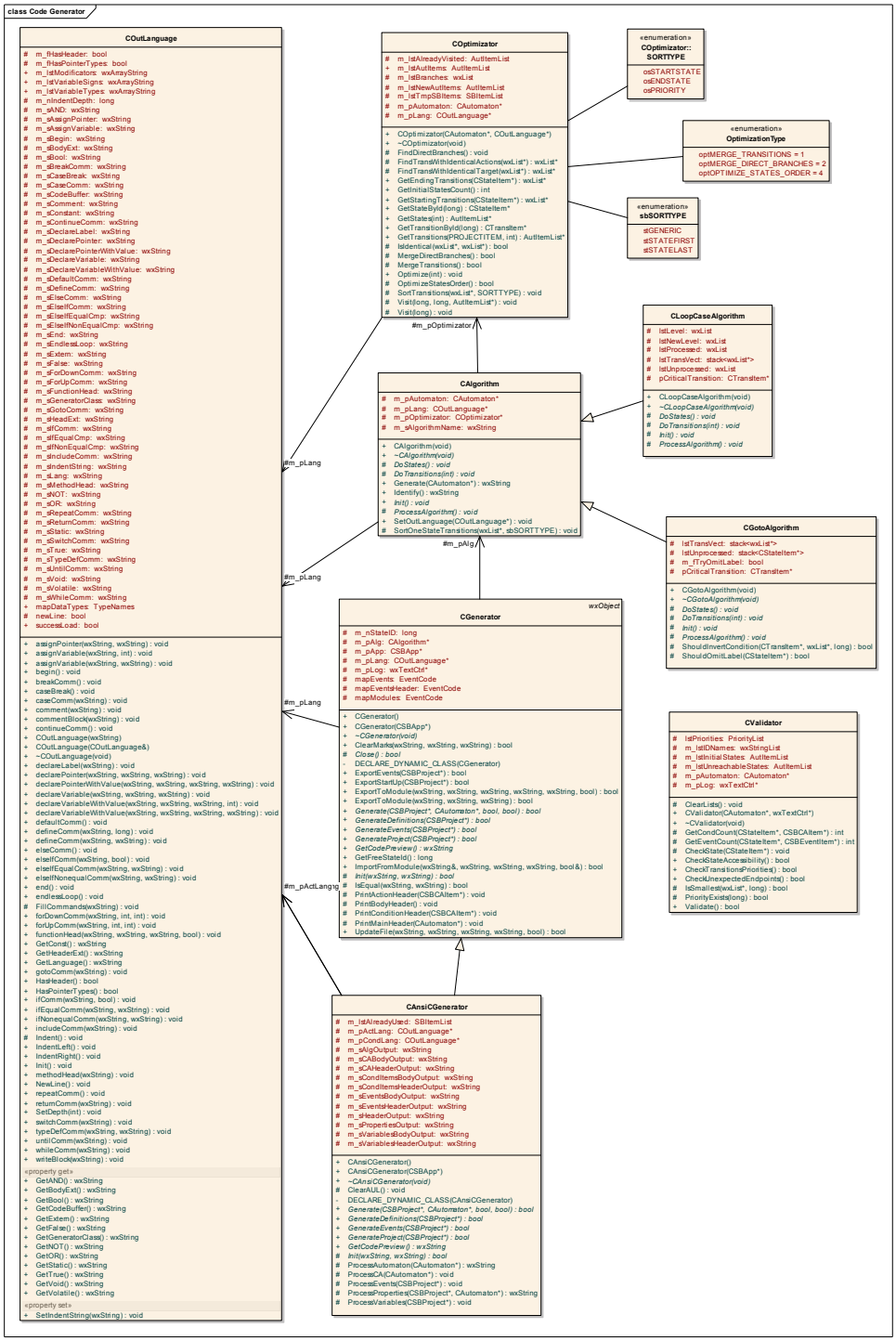
Obr. 47 Diagram tříd jádra aplikace State Builder



Obr. 48 Diagram tříd stavových automatů



Obr. 49 Diagram tříd COM API



Obr. 50 Diagram tříd generátoru programového kódu

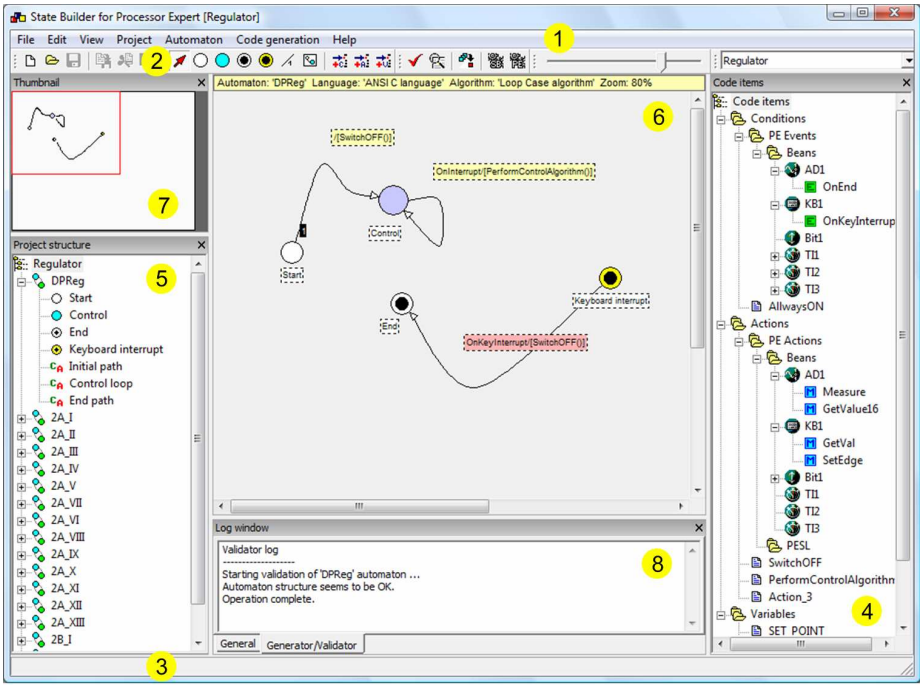
3.5 *Popis a ovládání aplikace State Builder*

Následující kapitoly pojednávají o vlastnostech a způsobu použití samotné aplikace State Builder. Je zde popsáno grafické uživatelské rozhraní aplikace a jeho jednotlivé ovládací prvky a na jednoduchém příkladu demonstrován kompletní proces vývoje embedded aplikace pomocí prostředí Processor Expert a jeho rozšíření State Builder.

3.5.1 Uživatelské rozhraní a ovládací prvky aplikace

3.5.1.1 Rámcové okno aplikace

Hlavní rámcové okno aplikace State Builder sestává z několika standardních a několika speciálních ovládacích prvků. Mezi standardní patří hlaví menu (1), panel nástrojů (2), a stavový řádek (3). Speciálními ovládacími prvky pak jsou panely programových komponent (4) a SB projektu (5), návrhová plocha automatu (6), navigační okno náhledu automatu (7) a okna zpráv (8).



Obr. 51 Hlavní rámcové okno aplikace State Builder

3.5.1.2 Hlavní menu

Hlavní menu aplikace zprostředkovává přístup k jednotlivým funkcím aplikace seřazeným dle jejich významu. Hlavní panel menu obsahuje 7 základních kategorií:

- **File** – funkce pro práci se soubory projektu aplikace State Builder, soubory historie a nastavení vlastností aplikace
- **Edit** – funkce pro práci se schránkou a objekty v návrhové ploše aplikace
- **View** – funkce pro přizpůsobení vizuálního vzhledu aplikace a rozložení jednotlivých ovládacích prvků
- **Project** – funkce pro práci s projektem aplikace SB, strukturou stavového automatu a synchronizaci generovaného programového kódu mezi aplikacemi PE a SB
- **Automaton** – funkce pro ověření struktury stavových automatů a nastavení jeho vlastností
- **Code generation** – funkce pro generování zdrojového kódu popsané embedded aplikace
- **Help** – funkce systému nápovědy aplikace State Builder

Význam jednotlivých položek menu zobrazených ve výše uvedených kategoriích je následující:

Menu **File**:

Funkce	Popis
New project	Vytvoř nový projekt. Projekt vytvořený z aplikace SB nebude přiřazen PE projektu a proto nebudou některé funkce aplikace k dispozici.
Open project	Otevři projekt ze souboru. Otevřený projekt bude automaticky propojen s příslušným projektem otevřeným v PE.
Save project	Ulož změny v aktivním projektu do jeho souboru.
Save project as...	Ulož projekt do nového souboru. Funkce je k dispozici pouze pro samostatné SB projekty bez návaznosti na projekty v PE.
Merge automaton	Importuj automat a uživatelsky definované uživatelské akce a

	podmínky přechodů.
Save automaton as...	Ulož aktivní automat a uživatelsky definované akce a podmínky do souboru.
History...	Podnabídka se soubory historie projektu.
Settings...	Otevři dialogové okno s volbami vlastností aplikace.

Menu **Edit**:

Funkce	Popis
Copy	Kopíruj aktuálně vybrané komponenty automatu do schránky.
Cut	Vyjmi aktuálně vybrané komponenty automatu a vlož je do schránky.
Paste	Vlož obsah schránky do aktivního automatu.
Delete	Smaž aktuálně vybrané komponenty automatu.

Menu **View**:

Funkce	Popis
Maximize workspace	Zobraz/skryj všechny ovládací panely.
File toolbar	Zobraz/skryj panel nástrojů pro práci se soubory.
Design toolbar	Zobraz/skryj panel nástrojů pro práci se strukturou stavového automatu.
Generator toolbar	Zobraz/skryj panel nástrojů pro generování programového kódu.
Zoom toolbar	Zobraz/skryj panel nástrojů pro změnu měřítka stavového diagramu.
Project toolbar	Zobraz/skryj panel nástrojů pro výběr aktivního projektu.
Layout toolbar	Zobraz/skryj panel nástrojů vzhledu aplikace.
Structure view	Zobraz/skryj panel komponent SB projektu.
Code items view	Zobraz/skryj panel programových komponent.
Log window	Zobraz/skryj panel oken zpráv.
Thumbnail	Zobraz/skryj panel náhledu stavového diagramu.

Menu **Project**:

Funkce:	Popis
Conditions/Actions/Variables	Podnabídka funkcí uživatelsky definovaných komponent stavového diagramu (akcí, podmínek, proměnných).

Add automaton	Vytvoř nový stavový automat.
Remove active automaton	Odstraň aktivní automat z projektu.
Remove all items	Vymaž strukturu aktivního stavového diagramu. Uživatelsky vytvořené komponenty (akce a podmínky) nebudou smazány.
Close project	Zavři aktivní projekt.
Synchronize code	Synchronizuj změny v generovaném kódu mezi PE a SB. Do SB budou importovány změny provedené v kódových fragmentech odpovídajících uživatelsky vytvářeným komponentám stavových diagramů aktivního projektu.
Re-activate warnings	Znovu aktivuj všechny uživatelem deaktivované varovné hlášení.
Properties...	Zobraz dialogové okno s vlastnostmi aktivního projektu.

Podnabídka **Conditions/Actions/Variables**:

Funkce	Popis
Add condition...	Vytvoř novou uživatelsky definovanou podmínku přechodu.
Add action...	Vytvoř novou uživatelsky definovanou akci přechodu.
Add variable...	Vytvoř novou uživatelsky definovanou globální proměnnou.
Remove all user items...	Odstraň všechny uživatelsky definované komponenty z aktivního projektu.

Menu **Automaton**:

Funkce	Popis
Validate	Ověř formální správnost struktury stavového diagramu.
Generate code preview	Zobraz náhled programového kódu generovaného z aktivního automatu.
Clear automatom	Odstraň obsah aktivního stavového automatu.
Properties...	Zobraz dialogové okno s vlastnostmi aktivního stavového automatu.

Menu **Generator**:

Funkce	Popis
Generate project code	Generuj programový kód všech automatů v aktivním projektu a importuj ho do PE.
Force generation of SB code	Programový kód stavových automatů bude automaticky generován před každým generováním kódu PE projektu (PE

	automaticky spouští proces generování kódu v SB).
Force generation of PE code	Programový kód projektu PE bude automaticky generován po každým generováním kódu z SB projektu (SB automaticky spouští proces generování kódu v PE).

Menu **Help**:

Funkce	Popis
Contents	Zobraz obsah témat nápovědy.
Context help	Aktivuj funkci kontextové nápovědy.
About...	Zobraz informace o verzi programu.

3.5.1.3 Panel nástrojů

Panely nástrojů aplikace State Builder umožňují rychlý přístup k nejčastěji používaným funkcím programu a usnadňují tak práci v prostředí aplikace. Uživatelské rozhraní State Builderu obsahuje šest různých panelů nástrojů, na nichž jsou umístěna tlačítka (zkratky) funkcí seskupená podle svého významu. Jedná se o panely nástrojů určených pro

- práci se soubory,
- návrh struktury stavového diagramu,
- ověření stavového automatu a generování programového kódu,
- změnu měřítka stavového diagramu,
- výběr aktivního projektu a
- úpravu vzhledu aplikace.

Funkce zpřístupněné na panelech nástrojů jsou tyto:

Panel nástrojů pro **práci se soubory**:

 Vytvoř nový SB projekt.

 Otevři SB projekt ze souboru.

 Ulož změny aktivního SB projektu.

Kopíruj vybrané komponenty aktivního automatu do schránky.

Vyjmi vybrané komponenty aktivního automatu a vlož je do schránky.

Vlož do aktivního automatu komponenty uložené ve schránce.

Obr. 52 Panel nástrojů pro práci se soubory

Panel nástrojů pro **návrh struktury stavového automatu**:

Nástroj pro změnu polohy a vlastností komponent stavového diagramu.

Vytvoř počáteční stav.

Vytvoř normální stav.

Vytvoř koncový stav.

Vytvoř asynchronní stav.

Vytvoř hranu přechodu.

Vytvoř skupinu stavů.

Vytvoř novou Podmínku přechodu.

Vytvoř novou Akci přechodu.

Vytvoř novou globální proměnnou.

Obr. 53 Panel nástrojů pro úpravu struktury stavového diagramu

Panel nástrojů pro **ověření stavového automatu a generování programového kódu**:

Ověř strukturu aktivního stavového automatu.

Zobraz náhled generovaného programového kódu.

Generuj programový kód pro aktivní projekt.

 Vynuť automatické generování pro SB projekt.

 Vynuť automatické generování pro PE projekt.

Obr. 54 Panel nástrojů pro ověření automatu a generování kódu

Panel nástrojů pro **změnu měřítka stavového diagramu** umožňuje plynou změnu velikosti prvků stavového diagramu v pracovní ploše aplikace.

Obr. 55 Panel nástrojů pro změnu měřítka automatu

Panel nástrojů pro **výběr aktivního projektu** umožňuje přepínat mezi více otevřenými projekty. Změnou aktivního projektu v SB se změní také aktivní PE projekt (pokud existuje).

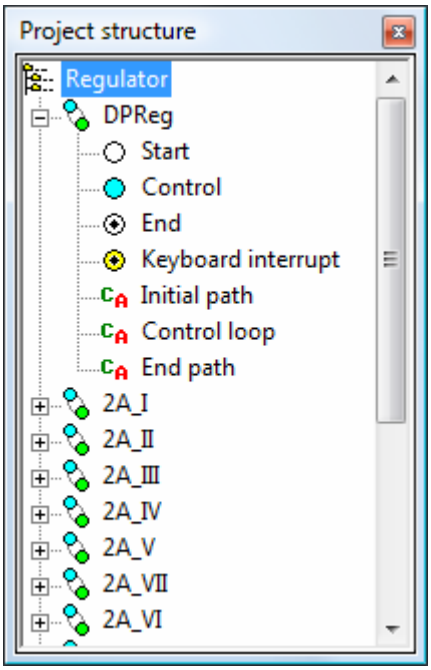
Obr. 56 Panel nástrojů pro změnu aktivního SB projektu

Panel nástrojů pro **změnu vzhledu aplikace**:

 Zobraz/skryj všechny ovládací panely.

Obr. 57 Panel nástrojů pro změnu vzhledu aplikace

3.5.1.4 Panel komponent SB projektu



Obr. 58 Panel komponent SB projektu

Panel komponent SB projektu zobrazuje pomocí přehledné stromové struktury všechny komponenty všech stavových automatů definovaných v aktivním projektu. Jednotlivé typy komponent jsou označeny speciálními ikonami tohoto významu:

- Stavový automat
- Normální stav
- Počáteční stav
- Koncový stav
- Asynchronní stav
- Hrana přechodu

Pořadí jednotlivých komponent ve stavovém automatu lze libovolně měnit přesouváním pomocí tažení myší. Tato operace nemá jen kosmetický význam; v případě,

že je pro generování programového kódu použit GOTO algoritmus a v nastavení aktivního SB projektu je vypnuta volba optimalizace pořadí generovaných stavů, pořadí jednotlivých komponent ovlivňuje také výstup z generátoru kódu (blíže viz kapitola 3.2.1.2).

Obecným pravidlem panelu komponenty SB projektu je, že při dvojkliku levým tlačítkem myši na položku stromu bude otevřeno dialogové okno s vlastnostmi příslušné komponenty. Zároveň bude pracovní plocha aplikace nastavena tak, aby zobrazovala aktuálně vybranou komponentu. Další funkcí společnou pro automaty a jejich komponenty je, že po stisku klávesy **F2** je přímo ve stromové struktuře umožněno přejmenování dané komponenty (což se projeví i v generovaném programovém kódu).

Každá komponenta má také svou vlastní kontextovou nabídku zobrazovanou po kliknutí pravým tlačítkem myši. Pomocí funkcí zobrazených v těchto nabídkách pak můžeme provádět další operace specifické pro daný typ komponenty. Kontextové nabídky dostupné z tohoto panelu jsou obecně trojího typu:

Kontextová nabídka projektu:

Funkce	Popis
Add automaton	Přidej nový automat do aktivního projektu.
Remove all...	Odstraň všechny automaty a jejich komponenty z projektu.
Properties...	Zobraz dialogové okno vlastností projektu.

Kontextová nabídka stavového automatu:

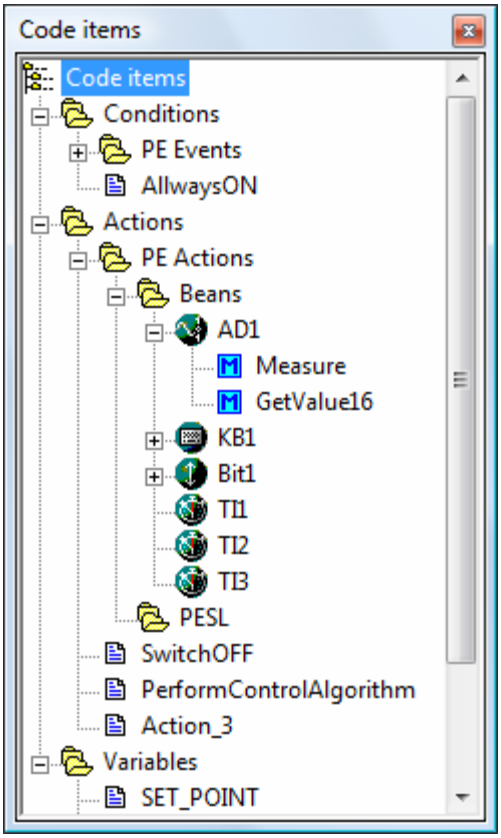
Funkce	Popis
Remove all items	Odstraň všechny komponenty z aktivního automatu.
Remove automaton	Odstraň automat z projektu.
Rename	Přejmenuj automat.
Properties...	Zobraz dialogové okno vlastností automatu.

Kontextová nabídka komponenty stavového automatu:

Funkce	Popis
Remove state/transition	Odstraň vybranou komponentu z aktivního automatu.
Rename	Přejmenuj vybranou komponentu.

Properties...	Zobraz dialogové okno s vlastnostmi vybrané komponenty.

3.5.1.5 Panel programových komponent



Obr. 59 Panel programových komponent

Panel programových komponent zobrazuje jak uživatelem vytvořené, tak importované komponenty programového kódu. Ty lze použít jak pro interaktivní definici struktury stavového diagramu, tak i pro definici jiných programových komponent.

V případě **uživatelsky definovaných programových komponent** se jedná o těla funkcí představujících podmínky, nebo akce přechodů mezi stavy automatu. Také je zde možnost definovat globální proměnné dále využívané v těchto komponentách. Komponenty akcí a přechodů lze vkládat jak do stavového diagramu, tak do jiných

programových komponent (to lze provést v okně integrovaného editoru programového kódu, který také obsahuje panel programových komponent). Uživatelsky vytvořené proměnné lze vkládat pouze do komponent akcí a přechodů.

V **případě importovaných programových komponent** se jedná o metody a události beanů importovaných z prostředí PE. Události beanů lze použít jako podmínky strážící přechody stavového automatu, metody pro ovládání periferií MCU zapouzdřené beany pak lze použít buď jako samostatné akce přechodů, nebo jako součásti jiných, uživatelsky vytvořených, programových komponent.

Vkládání komponent do stavového automatu, nebo přímo do editoru programového kódu se děje pomocí tažení příslušné komponenty myší. Dvojklikem levého tlačítka myši na položku programové komponenty lze buď otevřít okno integrovaného editoru programového kódu a v něm nadefinovat tělo komponenty (to v případě komponent uživatelsky vytvořených akcí a přechodů), nebo vyvolat dialogové okno s volbami vlastností komponenty (u komponent uživatelsky vytvořených akcí, přechodů, nebo proměnných). U importovaných programových komponent nemá tato akce žádný význam.

Stejně jako v případě panelu komponent SB projektu, i v panelu programových komponent existují kontextové nabídky vyvolávané stiskem pravého tlačítka myši. Jsou to kontextové nabídky pro:

Uživatelem definované podmínky:

Funkce	Popis
Add condition...	Vytvoř novou uživatelsky definovanou podmínku.
Remove condition	Odstraň podmínku z projektu.
Edit code...	Uprav programový kód podmínky.
Properties...	Zobraz dialogové okno vlastností podmínky.

Uživatelem definované akce:

Funkce	Popis
Add action...	Vytvoř novou uživatelsky definovanou akci.

Remove action	Odstraň akci z projektu.
Edit code...	Uprav programový kód akce.
Properties...	Zobraz dialogové okno vlastností akce.

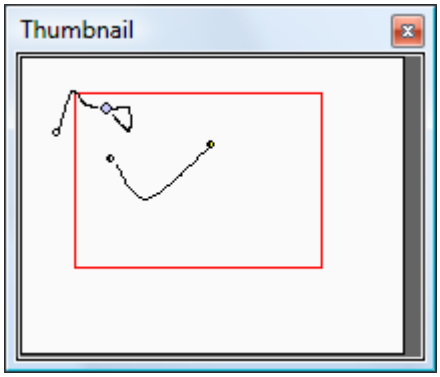
Globální proměnné:

Funkce	Popis
Add variable...	Vytvoře novou globální proměnnou.
Remove variable	Odstraň globální proměnnou z projektu.
Properties...	Zobraz dialogové okno s vlastnostmi proměnné.

V případě, že uživatel klikne pravým tlačítkem myši mimo oblast jakékoliv programové komponenty, bude zobrazena tato nabídka:

Funkce	Popis
Add condition...	Vytvoř novou uživatelsky definovanou podmínku.
Add action...	Vytvoř novou uživatelsky definovanou akci.
Add variable...	Vytvoř novou globální proměnnou.
Remove all user defined items...	Odstraň všechny uživatelské programové komponenty z projektu.

3.5.1.6 Náhled automatu



Obr. 60 Panel náhledu stavového diagramu

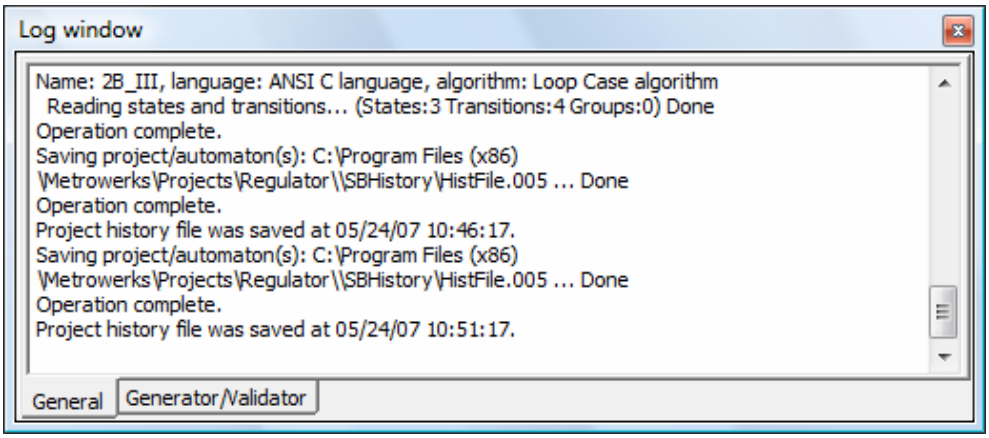
Stavové digramy vytvářené na pracovní ploše aplikace SB mohou být velmi rozsáhlé. Aby byla zachována možnost přehledné tvorby a orientace se v návrhu, byl v uživatelském rozhraní SB implementován speciální ovládací prvek umožňující zobrazovat zmenšený náhled struktury aktivního stavového diagramu.

Další vlastností tohoto ovládacího panelu je možnost zrychlené navigace napříč celým stavovým automatem; uživateli je umožněno pomocí tažení myší přesouvat červeně ohraničený obdélník v náhledu automatu představující aktuálně zobrazenou část pracovní plochy a tím měnit pozici zobrazované oblasti.

Náhledové okno umožňuje rovněž nastavit, které typy komponent stavového diagramu v něm budou zobrazovány a které ne, čímž lze zpřehlednit jeho obsah. Volbu zobrazovaných komponent lze provést prostřednictvím kontextové nabídky toho ovládacího prvku, která obsahuje následující položky:

Funkce	Popis
Show transitions	Zobraz / Skryj hrany přechodů.
Show groups	Zobraz / Skryj skupiny stavů.
Show notes	Zobraz / Skryj poznámky komponent stavového diagramu.

3.5.1.7 Okna zpráv

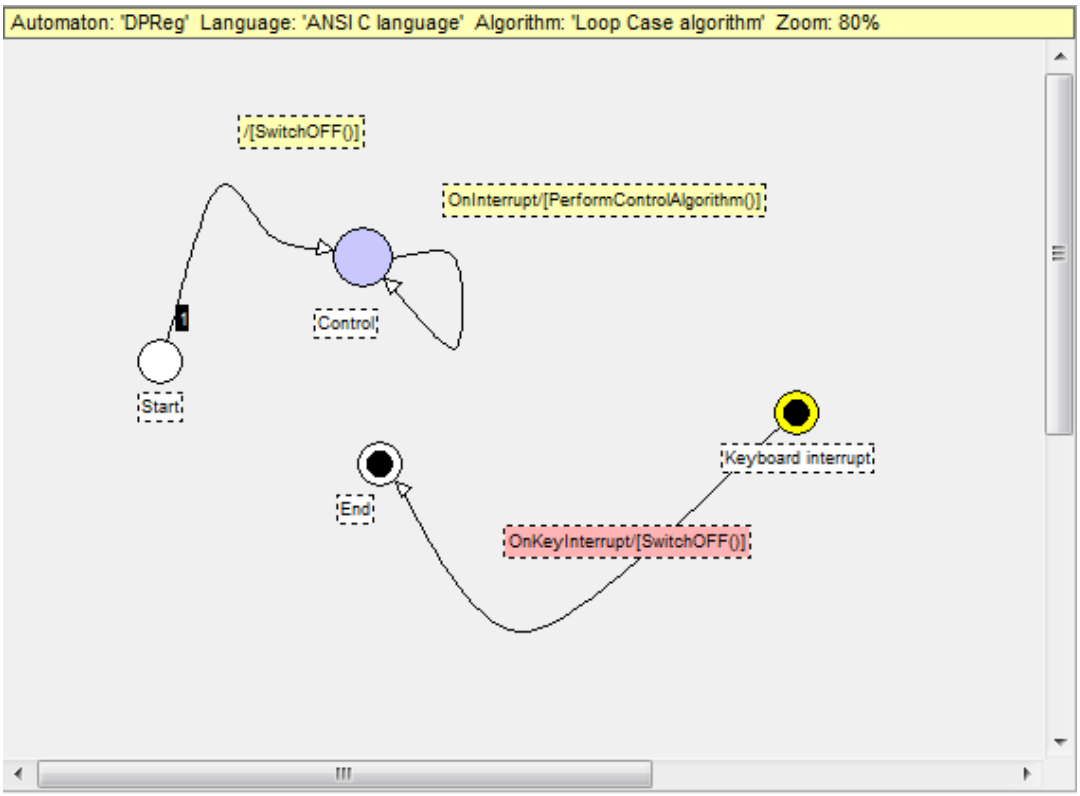


Obr. 61 Panel oken zpráv

Panel oken zpráv slouží k zobrazování informací o aktuální stavu/činnosti aplikace SB a obsahuje tři samostatná okna umístěná v záložkách. První z nich, označené záložkou *General*, zobrazuje obecné informace o průběhu načítání projektů, archivaci souborů historie a dalších obecných operacích aplikace SB. Druhé, označené jako *Generátor/Validator*, obsahuje zprávy validátoru struktury stavového automatu a generátoru programového kódu.

3.5.1.8 Pracovní plocha aplikace

Pracovní plocha aplikace umožňuje vlastní tvorbu stavového diagramu. Veškeré aktivity jsou prováděny pomocí operací s myší.



Obr. 62 Pracovní plocha aplikace State Builder

Jednotlivé komponenty diagramu lze vybírat z příslušného panelu nástrojů či programových komponent (viz kapitoly 3.5.1.3 a 3.5.1.5) a pomocí drag&drop operací je přesouvat po pracovní ploše, či s nimi standardně pracovat pomocí schránky. Komponenty vkládané do pracovní plochy jsou zobrazovány také v panelu komponent SB projektu (viz kapitola 3.5.1.4).

Každá komponenta diagramu, stejně tak jako samotná pracovní plocha, umožňuje zobrazení kontextové nabídky s funkcemi relevantními zvolené komponentě. Komponenty diagramu navíc umožňují rychlý přístup k dialogovému oknu s jejich vlastnostmi pomocí dvojkliku levého tlačítka myši na danou komponentu. Kontextové nabídky pracovní plochy a jednotlivých komponent diagramu jsou následující:

Nabídka **Stavu**:

Funkce	Popis
Remove state	Odstraň stav z automatu.
Ungroup	Odstraň strav ze skupiny (stav nebude smazán).
Show note	Zobraz / skryj poznámku ke stavu.
Dock note	Ukotvi / uvolni poznámku stavu (položka menu je viditelná pouze tehdy, je-li poznámka zobrazena).
Properties...	Zobraz dialogové okno s vlastnostmi stavu.

Nabídka **Skupiny stavů**:

Funkce	Popis
Remove group	Zruš skupinu stavů (stavy nebudou smazány).
Show note	Zobraz / skryj poznámku ke skupině stavů.
Dock note	Ukotvi / uvolni poznámku skupiny stavů (položka menu je viditelná pouze tehdy, je-li poznámka zobrazena).
Properties...	Zobraz dialogové okno s vlastnostmi skupiny stavů.

Nabídka **Hrany přechodů**:

Funkce	Popis
Create action	Vytvoř novou programovou komponentu Akce a přiřaď ji k přechodu.
Create condition	Vytvoř novou programovou komponentu Podmínky a přiřaď ji

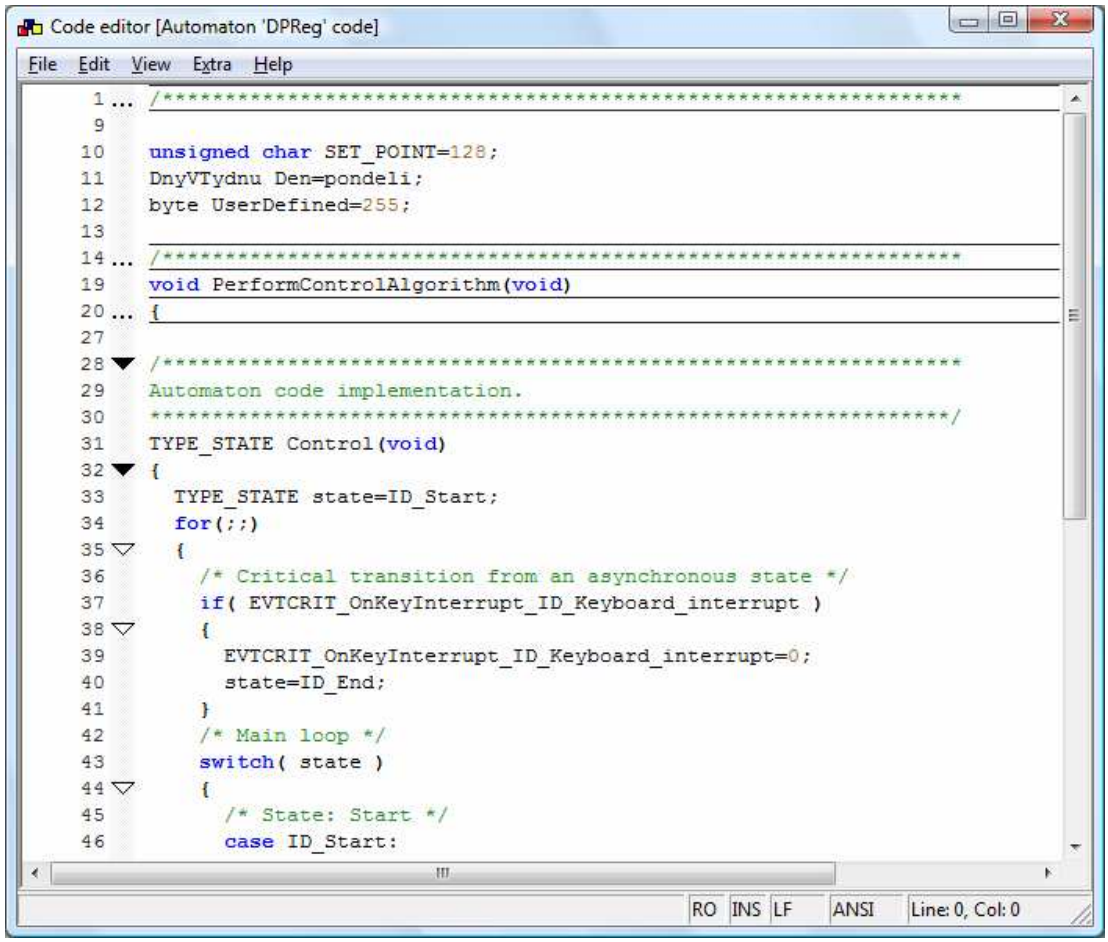
	k přechodu.
Edit action code	Podnabídka obsahuje položky představující všechny programové komponenty akcí přiřazené k danému přechodu. Po zvolení položky akce bude její programový kód zobrazen v integrovaném textovém editoru.
Edit condition code	Otevři programový kód podmínky přechodu v integrovaném textovém editoru.
Remove transition	Odstraň hranu přechodu; přiřazené stavy budou rovněž odstraněny. Funkce neovlivní případné přiřazené akce a podmínku přechodu.
Show note	Zobraz / skryj poznámku ke hraně přechodu.
Dock note	Ukotvi / uvolni poznámku hrany přechodu (položka menu je viditelná pouze tehdy, je-li poznámka zobrazena).
Properties...	Zobraz dialogové okno s vlastnostmi hrany přechodu.

Nabídka pracovní plochy:

Funkce	Popis
View all	Nastav měřítko pracovní plochy tak, aby byl viditelný celý stavový diagram.
Preview code	Zobraz náhled programového kódu generovaného z aktivního stavového diagramu.
Show all non-empty notes	Zobraz všechny neprázdné poznámky komponent stavového diagramu.
Hide all notes	Skryj všechny zobrazené poznámky.
Properties...	Zobraz dialogové okno s vlastnostmi aktivního automatu.
Remove this automaton	Odstraň aktivní automat z SB projektu.
Clear automaton	Odstraň všechny komponenty diagramu aktuálního automatu.

3.5.1.9 Integrovaný editor zdrojového kódu

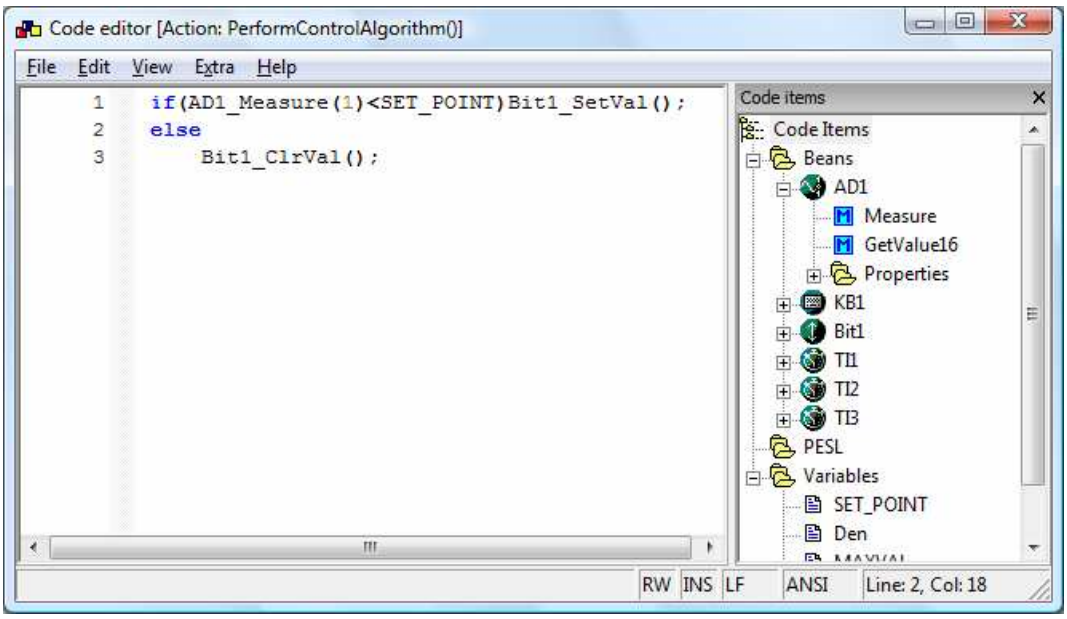
Integrovaný textový editor programového kódu je v prostředí aplikace SB využíván k editaci uživatelem vytvářených komponent programového kódu (programový kód podmínek a akcí přechodu, globálních proměnných s uživatelskými datovými typy), nebo k prohlížení generovaného programového kódu.



Obr. 63 Integrovaný textový editor v módu náhledu generovaného programového kódu

Pokud je editor použit k editaci programových komponent, obsahuje jeho okno také redukovanou variantu panelu programových komponent zobrazujícího uživatelsky definované i importované komponenty, které mohou být vkládány do textu pomocí tažení myší.

Textový editor disponuje vlastnostmi, které se v současné době objevují ve většině moderních integrovaných vývojových prostředích. Jedná se zejména o možnosti zvýraznění syntaxe použitého programovacího jazyka, číslování řádků či skrývání určitých bloků oblastí textu (tzv. *code folding*).



Obr. 64 Integrovaný textový editor v módu editace programového kódu uživatelských programových komponent

Samotné okno editoru obsahuje hlavní nabídku umožňující efektivně pracovat i s rozsáhlým zdrojovým textem. Jednotlivé funkce jsou tematicky rozděleny do několika podnabídek obsahujících tyto položky:

Nabídka **File**:

Funkce	Popis
Open	Otevři textový soubor v editoru.
Save as...	Ulož obsah editoru do textového souboru.
Close	Zavři okno editoru. Změny provedené v obsahu budou zohledněny.

Nabídka **Edit**:

Funkce	Popis
Undo	Operace „Zpět“
Redo	Operace „Znovu“
Copy	Kopíruj vybraný text do schránky.
Cut	Vyjmi vybraný text do schránky.

Paste	Text ze schránky vlož do editoru.
Delete	Smaž vybraný text.
Find...	Najdi první výskyt daného řetězce.
Find next	Najdi další výskyt řetězce.
Replace...	Nahraď jeden podřetězec jiným.
Replace again	Nahraď další instanci podřetězce jiným.
Go to...	Jdi na řádek…
Match brace	Vyber text uvnitř závorek.
Toggle bookmark	Zobraz/skryj záložku na aktuálním řádku.
Next bookmark	Nastav kurzor na další záložku.
Previous bookmark	Nastav kurzor na předchozí záložku.
Clear all bookmarks	Odstraň všechny záložky.
Indent increase	Zvětši odsazení bloku textu.
Indent reduce	Zmenši odsazení bloku textu.
Select all	Označ kompletní obsah editoru.
Select line	Označ text aktuálního řádku.

Menu **View**:

Funkce	Popis
Highlight language	Vyber programovací jazyk pro něhož bude zvýrazňována syntaxe.
Toggle current fold	Zobraz / skryj blok kódu.
Overwrite mode	Přepis / vkládání textu.
Wrap mode	Automatické zalamování řádků.
Show line endings	Zobraz / skryj konce řádků.
Show line guides	Zobraz / skryj pomocné vodítka řádků.
Show line numbers	Zobraz / skryj čísla řádků.
Show long line marker	Zobraz / skryj ukazatele dlouhých řádků.
Show whitespace	Zvýrazni mezery.
Use code page of...	Nastav stránku znakové sady.
Show code items	Zobraz / skryj panel programových komponent.

Nabídka **Extra**:

Funkce	Popis

Readonly mode	Povol / zakaž mód „Pouze pro čtení“.
Change case to...	Převeď na VELKÁ/malá písmena.
Change whitespace to...	Převeď mezery na tabulátory a naopak.
Convert line endings to...	Změň typ konců řádků.
Remove trailing whitespaces	Odstraň mezery s konců řádků.

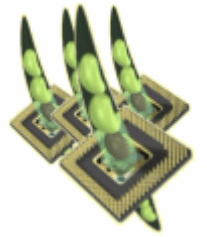
Menu **Help**:

Funkce	Popis
Content	Zobraz témata nápovědy.

3.5.2 Práce s aplikací State Builder

3.5.2.1 Základní koncepty

Dříve než se zaměříme na samotnou práci s aplikací State Buidler, je třeba ozřejmit také některé základní koncepty prostředí PE a vzájemnou vazbu aplikací PE a SB.



Integrované vývojové prostředí Processor Expert™ slouží k vývoji softwarového vybavení pro embedded systémy. Prostředí PE je založena na unikátní technologii tzv. **Embedded Beanů**. Jedná se o speciální softwarové komponenty zapouzdřující jak hlavní vlastnosti a funkcionalitu MCU, tak i jeho interní a externí HW periferie.

Každý bean obsahuje sadu metod, které lze použít pro inicializaci a nastavení vlastností dané periferie/MCU jak v době návrhu aplikace, tak i za jejího chodu, a navíc obsahuje speciální funkce sloužící jako obslužné rutiny událostí/přerušení HW periferií. Rozhodne-li se uživatel využívat funkcionality dané periferie, jednoduše ji přidá do projektu aplikace PE a pomocí speciálních ovládacích prvků tohoto prostředí nastaví její vlastnosti, či použije metody a události definované beanem ve svém zdrojovém kódu.

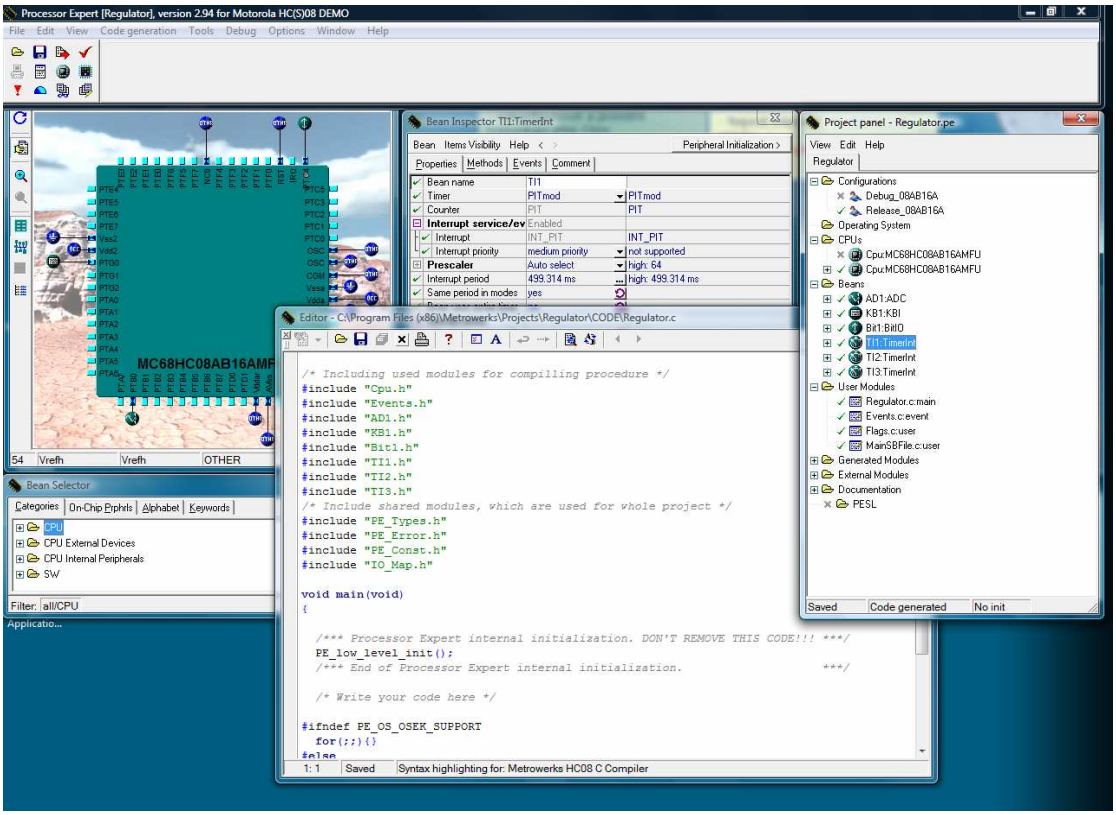
Způsob práce v prostředí PE spočívá v tom, že uživatel nejprve vloží do projektu bean zapouzdřující cílový MCU, a poté beany zapouzdřující jednotlivé HW periferie, které

míní ve vyvíjeném embedded zařízení používat. Poté může z prostředí PE nastavit vlastnosti těchto periferií a v integrovaném textovém editoru začít psát zdrojový kód aplikace s využitím metod a funkcí událostí publikovaných vloženými beany. Publikované funkce beanů tvoří univerzální API, takže názvy i parametry funkcí zůstávají stejné u všech beanů zapouzdřujících HW stejného typu. Poté, co uživatel vytvoří kompletní zdrojový kód embedded aplikace, nechá Processor Experta™ znovu vygenerovat její kód. Tímto procesem jsou všechny univerzální funkce beanů převedeny na volání nativních funkcí (registrů) použitého MCU a jeho periferií, čímž uživatel získá hotový produkční kód vhodný k přímému překladu pomocí překladače určeného pro danou cílovou HW platformu.

Je-li pak nutné nasadit stejnou aplikaci na jiném MCU, jednoduše se v projektu PE změní bean zapouzdřující původní MCU za bean zapouzdřující nové MCU, přegeneruje se výstupní kód, a tím uživatel získá novou verzi zdrojových kódů aplikace přeložitelnou pro nový MCU.

Všechna výše uvedená funkcionalita je zapouzdřena aplikací PE. Jak ale přispívá k tvorbě embedded aplikace rozšíření State Builder?

Aplikace State Builder tvoří nadřazenou vrstvu nad aplikací PE a umožňuje popisovat aplikační logiku vytvářené embedded aplikace pomocí stavových diagramů a z těch generovat zdrojový programový kód této aplikace. SB importuje API funkce beanů vložených do PE projektu a umožňuje jejich použití v uživatelem definovaných programových komponentách, které jsou použity jako podmínky a akce přechodů stavových automatů. Kód generovaný aplikací SB je ukládán do nových souborů automaticky vkládaných do PE projektu, nebo přímo modifikuje původní programové soubory vytvořené v prostředí PE. Tím je zajištěno, že po přegenerování PE projektu je ve výsledném programovém kódu zahrnut jak původní kód vytvořený autorem přímo v prostředí PE, tak i kód vzniklý generováním z prostředí aplikace SB.



Obr. 65 Prostředí aplikace Processor Expert™

Nyní si již tedy na jednoduchém příkladu demonstrujme celý vývojový proces embedded aplikace vytvořené pomocí IDE Processor Expert™ s rozšířením State Builder.

3.5.2.2 Tvorba PE projektu

Způsob práce s aplikací State Builder bude demonstrován na jednoduchém příkladu aplikace nazvané **Blinker** představující generátor obdélníkového pulsu s periodou 1 s. Výstup bude realizován pomocí jednoho pinu výstupního portu MCU připijeného na LED diodu a operace zastavení generátoru a ukončení programu budou realizovány prostřednicím externího modulu klávesnice.

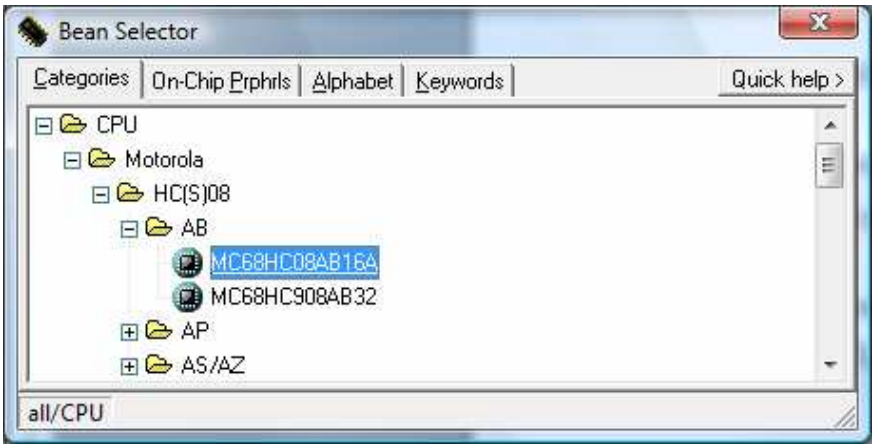
Postup tvorby a nastavení PE a SB projektu by měl být následující:

Prvním krokem při tvorbě embedded aplikace v prostředí Procesor Expert™ s rozšířením State Builder **je příprava PE projektu**. Nastavení PE projektu spočívá především ve výběru vhodných beanů zapouzdřujících MCU a periferie, které budeme v naší embedded aplikaci využívat. Pokud v průběhu počáteční konfigurace PE projektu opomeneme vložit některý z potřebných beanů, lze tak učinit kdykoliv v průběhu další práce. Změny PE projektu budou automaticky zohledněny i v prostředí SB.

Postup konfigurace PE projektu provedeme v těchto krocích:

1. V prostředí PE vytvoříme nový projekt prostřednictvím položky menu *File->New project…*

2. V prostředí PE vybereme v okně označeném jako *Bean Selector* na záložce *Categories* ze složky *CPU* bean cílového MCU a dvojklikem levého tlačítka myši ho přidáme do projektu. V našem případě jsme jako cílový MCU zvolili chip MC68HC08AB16A.

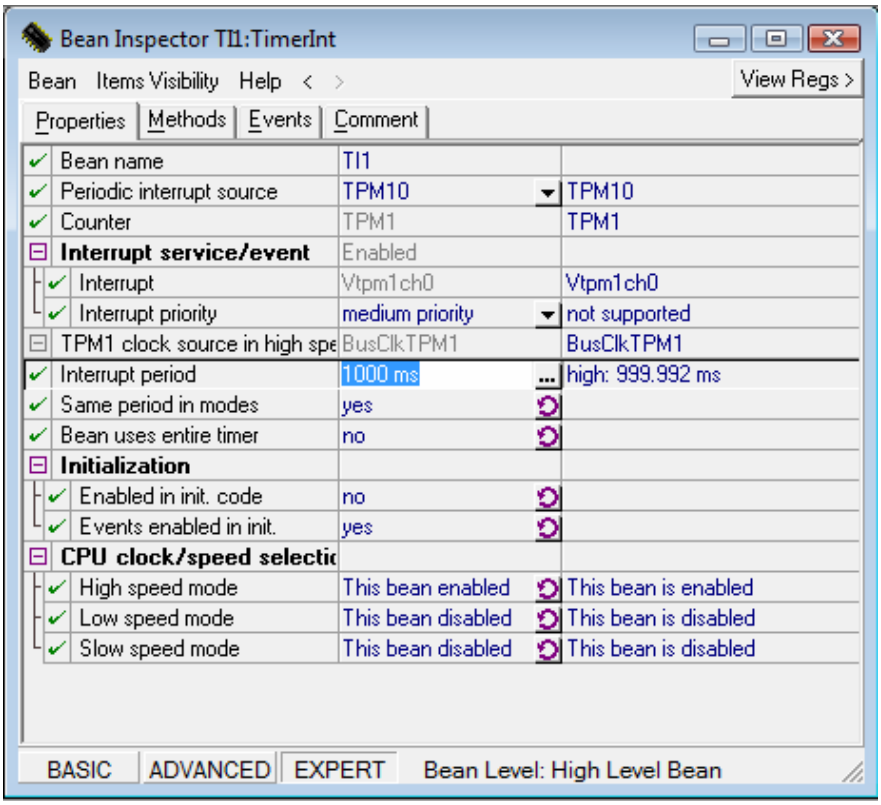


Obr. 66 Okno *Bean Selector* prostředí Procesor Expert™

3. Stejným způsobem vybereme z dalších složek, popř. záložky *On-Chip Prphrls*, beany potřebných periferií a vložíme je do projektu. Pro naši zamýšlenou aplikaci budeme potřebovat beany zapouzdřující 1

vstupně/výstupní bit výstupního portu (*BitIO*), periodické přerušení (*TimerInt*) a externí klávesnici připojenou na jeden z portů MCU (*KBI*).

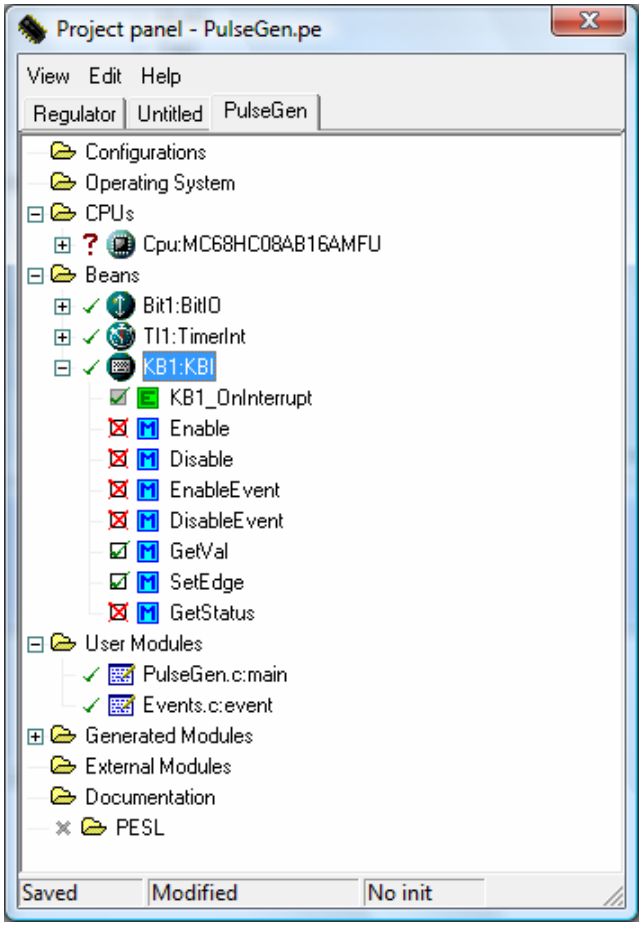
4. Pomocí okna *Bean Inspector* (opět v prostředí PE) nastavíme potřebné vlastnosti všech vložených beanů (provedeme počáteční inicializaci všech periferií). Aplikace PE sama kontroluje, zda jsou jednotlivá nastavení beanů v povolených mezích a v případě jakéhokoliv rozporu (např. při použití již obsazených pinů, nebo při nekorektním nastavení určitých hodnot) nás na to upozorní prostřednictvím příslušných informačních ikon a zpráv v oknech Bean Inspectoru a okně zpráv.



Obr. 67 Okno *Bean Inspector* v prostředí Procesor Expert™

5. V okně *Project panel* v prostředí PE si pak můžeme zkontrolovat obsah PE projektu a seznam publikovaných metod a událostí zapouzdřených

jednotlivými beany. Okno panelu projektu obsahuje stromovou strukturu, ve které jsou zobrazeny všechny použité beany a po rozbalení jednotlivých listů stromu znázorňujících daný bean se zobrazí položky všech publikovaných metod (položky jsou označeny pomocí modré ikony **M**) a událostí (zelená ikona **E**). Při zvolení položky stromu odpovídající beanu se aktualizuje okno *Bean Inspectoru* a umožní tak editaci vlastností daného beanu.



Obr. 68 Okno *Project panel* v prostředí Procesor Expert™

6. Nyní můžeme zkusit vygenerovat prvotní zdrojový kód aplikace (její „kostru“) pomocí položky menu *Code generation -> Generate Code* z okna

Procesor Experta, nebo také stisknutím kombinace kláves *Ctrl+G* (pokud bychom tak neučinili, programové soubory zobrazené v okně *Project panel* ve složce *User Modules* by nebyly dostupné). V tomto okamžiku není sice nutné mít vygenerovaný zdrojový kód kostry aplikace, můžeme si tím však ověřit správnost nastavení beanů v projektu; v případě jakékoliv chyby v nastavení nám totiž aplikace PE toto generování neumožní.

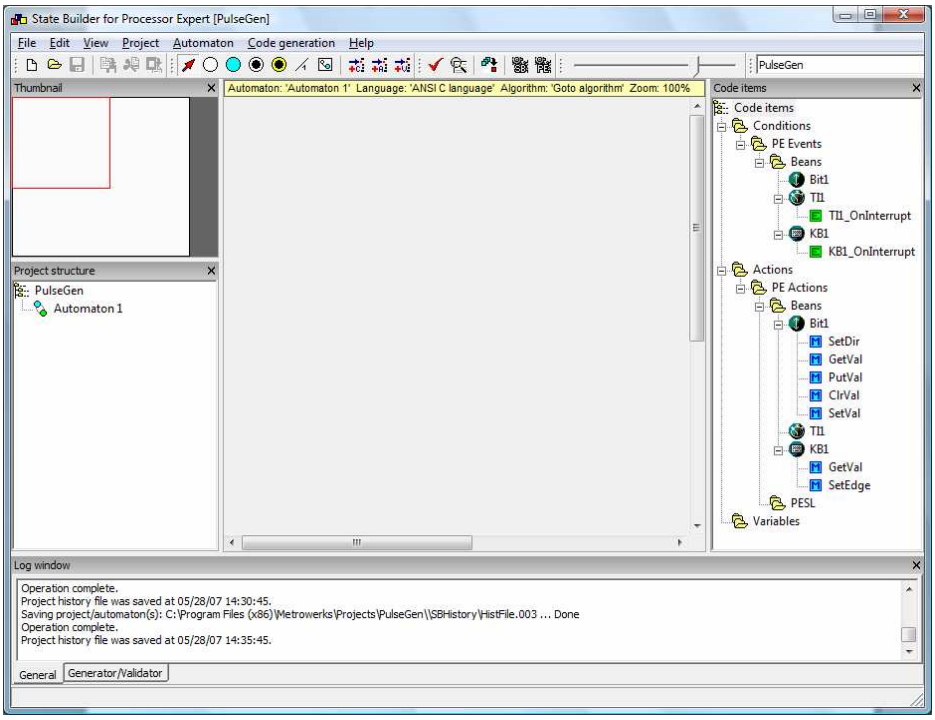
Po úspěšném vygenerování kódu projektu již můžeme začít tvořit vlastní aplikaci v prostředí rozšíření State Builder.

3.5.2.3 Návrh struktury stavového automatu v prostředí State Builder

Druhým krokem při tvorbě naší aplikace **bude definice stavového diagramu** popisujícího její aplikační logiku. V této kapitole si ukážeme, jak v prostředí aplikace State Builder interaktivně vytvořit grafickou reprezentaci stavového diagramu popisujícího naši aplikaci.

Postup konfigurace SB projektu a tvorby stavového diagramu je následující:

1. Z prostředí PE spustíme aplikaci State Builder prostřednictvím položky menu *Tools -> State Builder for Procesor Expert*. Po krátké inicializační proceduře indikované malým oknem s ukazatelem průběhu se zobrazí okno aplikace State Builder. Okamžitě po zobrazení okna aplikace SB je vytvořen nový SB projekt a ten je pomocí programového API provázán s jeho „protějškem“ v prostředí PE. Tím je umožněna vzájemná komunikace mezi PE a SB a synchronizace jejich projektů (konfigurační soubor SB projektu je automaticky ukládán vedle souboru PE projektu a toto nastavení je doporučeno zachovat). Povšimněte si zejména okna programových komponent (viz kapitola 3.5.1.5), jehož obsah by měl přesně reflektovat stav beanů vložených do PE projektu.



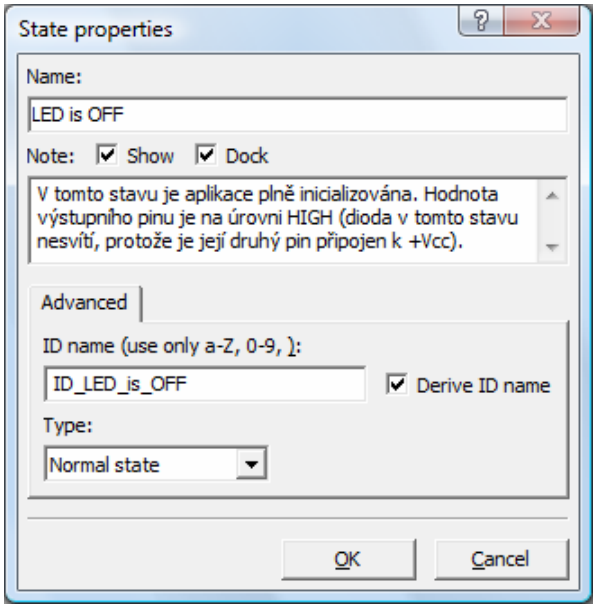
Obr. 69 Okno aplikace State Builder s novým projektem

2. Nyní můžeme pomocí panelů nástrojů (viz kapitola 3.5.1.3) vytvořit stavový diagram aplikace. To provedeme výběrem vhodných komponent z panelu nástrojů pro úpravu struktury stavového diagramu a jejich vkládáním do pracovní plochy aplikace (kapitola 3.5.1.8) pomocí myši. Stejným způsobem také vytvoříme propojení jednotlivých stavů. Postup tvorby diagramu je následující:

   a. Do pracovní plochy vložíme jeden **počáteční** a jeden **normální stav**. Stavy postupně označíme pomocí levého tlačítka myši a pomocí klávesy **F2** změníme jejich výchozí názvy. Počáteční stav přejmenujeme na „*Start*“ a normální stav na „*LED is OFF*.“

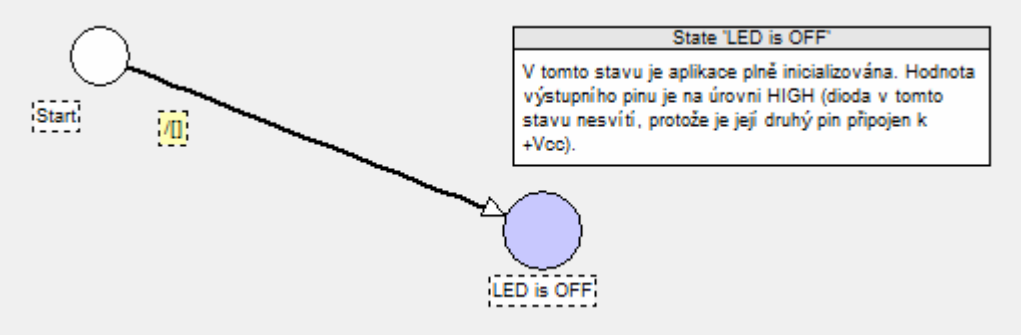
   b. Dvojklikem levého tlačítka myši na symbol stavu v návrhové ploše otevřeme dialogové okno s vlastnostmi normálního stavu a ty

upravíme podle následujícího obrázku. Tím zajistíme, že v blízkosti stavu se bude zobrazovat také malé okno s komentářem k danému stavu.

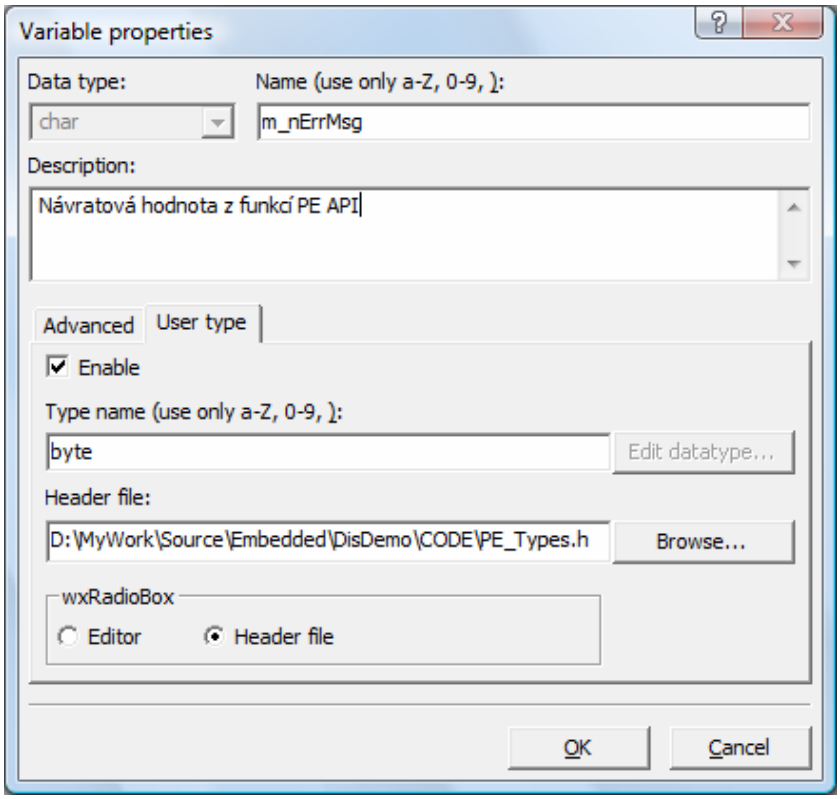


Obr. 70 Vlastnosti inicializačního stavu aplikace

c. Dialogové okno zavřeme pomocí tlačítka „OK“ a poté propojíme oba stavy pomocí přechodu tak, jak je patrné z následujícího obrázku.



Obr. 71 Inicializační část vytvářené embedded aplikace PulseGenerator

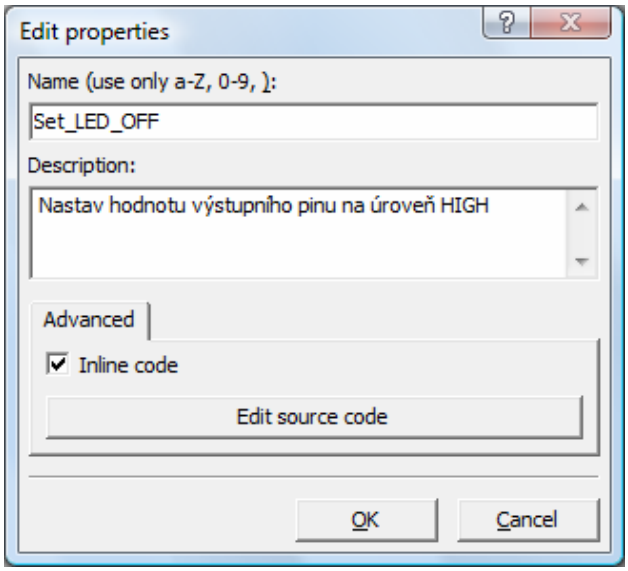


Obr. 72 Vytvoření nové globální proměnné

d. Vytvoříme jednu globální proměnnou nazvanou `m_nErrMsg`, kterou budeme používat jako kontejner pro návratové hodnoty z funkcí PE API. Z panelu nástrojů vybereme funkci „*Add new variable*“ a vlastnosti nové proměnné nastavíme podle obrázku 72.

e. Dále vytvoříme akci přechodu mezi stavy „*Start*“ a „*LED is OFF*“ nazvanou `Set_LED_OFF` tak, že klikneme na žlutý obdélník představující popisek hrany přechodu pravým tlačítkem myši a z kontextové nabídky zvolíme funkci „*Create action*“. Aplikace SB zobrazí dialogové okno akce, které vyplníme podle obrázku 73. Jméno akce v editační políčku „*Name*“ musí odpovídat konvenci ANSI C pro pojmenovávání programových identifikátorů. Přepínač

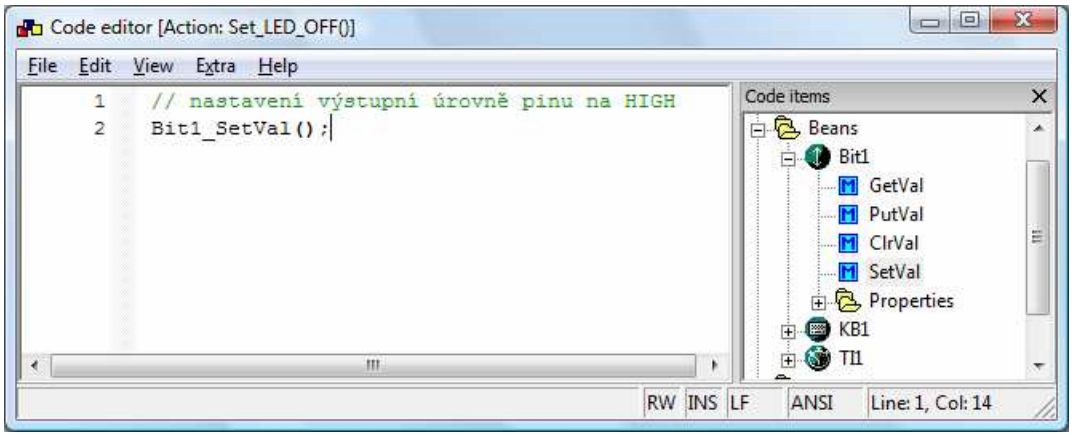
„*Inline code*“ by měl být aktivován; tím zajistíme, že tato akce (respektive její programový kód) bude v průběhu generování vložena přímo do těla funkce automatu a nebude volána jako další, vnořená funkce, čímž ušetříme místo v zásobníku aplikace.



Obr. 73 Vlastnosti akce „Set_LED_OFF“

f. Po nastavení základních vlastností akce můžeme přistoupit k editaci jejího programového kódu. Dialogové okno vlastností akce **necháme otevřené** a klikneme na tlačítko „*Edit source code*“. Tím otevřeme okno integrovaného editoru programových komponent a v něm vytvoříme programový kód akce. Při tvorbě můžeme využívat importovaných PE komponent, které se zobrazují v panelu na pravé straně okna editoru; ze stromu zobrazeného v tomto panelu můžeme jednoduše potřebné metody přetahovat pomocí myši.

Upravme tedy programový kód akce podle následujícího obrázku:

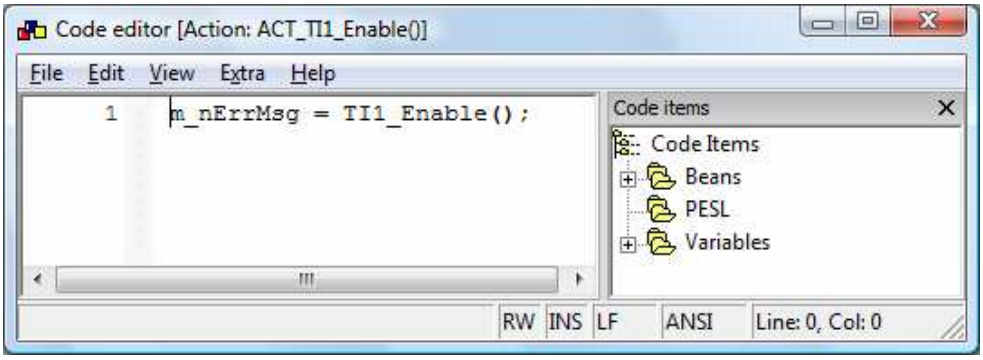


Obr. 74 Programový kód akce „Set_LED_OFF“

g. Změnu programového kódu potvrdíme zavřením okna a proces tvorby akce ukončíme zavřením dialogového okna vlastností akce pomocí tlačítka „*OK*“.

h. Akce přechodů lze také velmi jednoduše tvořit/přiřazovat prostým přetažením požadované funkce z panelu programových komponent. Přetahovat lze jak uživatelsky vytvořené programové komponenty (např. výše uvedenou komponentu „`Set_LED_OFF`“), tak také metody a události importovaných beanů. Tímto způsobem také vytvoříme druhou akci prováděnou při přechodu ze stavu „Start“ do „LED is OFF“. Pomocí levého tlačítka myši uchopíme importovanou metodu beanu timeru TI1 s názvem `Enable` a přetáhneme ji nad popisek vlastností přechodu, kde ji upustíme. Tím se vytvoří nová uživatelská programová komponenta s výchozím názvem `ACT_TI1_Enable` a přiřadí se k danému přechodu. Programový kód této komponenty bude obsahovat volání požadované PE API funkce (v tomto případě funkce `TI1_Enable()`). Vlastnosti nově vytvořené akce i jejího programového kódu lze pak kdykoliv změnit (stejným způsobem, jako u uživatelsky vytvořené programové komponenty). Abychom

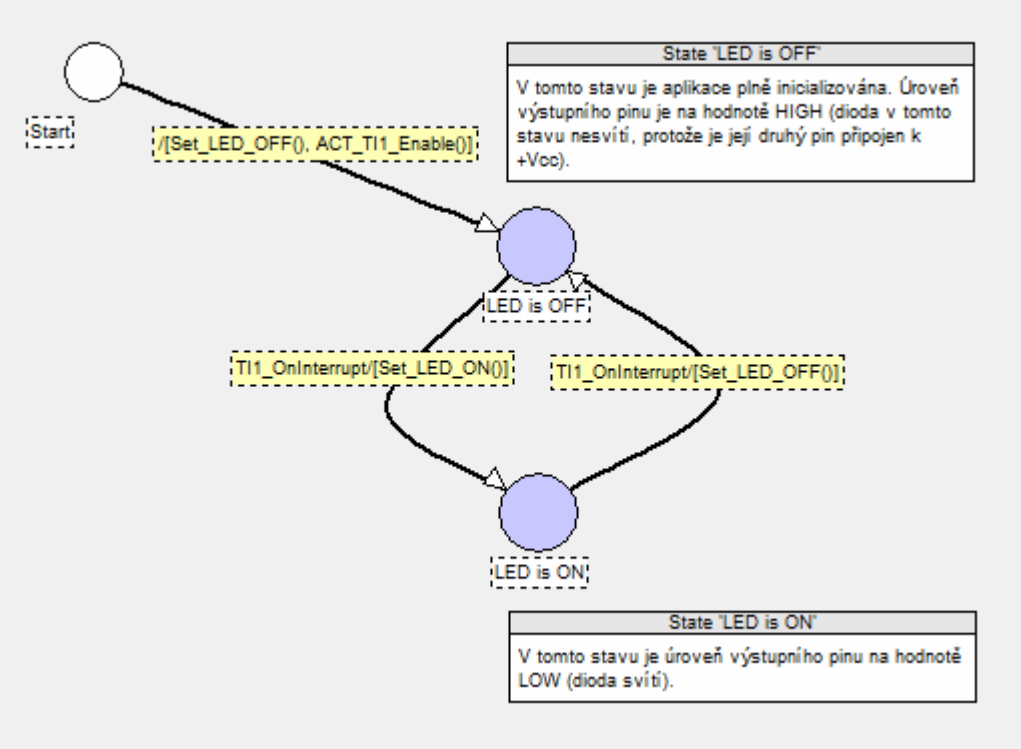
zabránili generování varovných hlášení při překladu aplikace, upravme kód akce podle následujícího obrázku.



Obr. 75 Programový kód akce „ACT_TI1_Enable“

i. Dalším krokem bude vytvoření stavu, do kterého se aplikace dostane po tiku časovače (po příchodu požadavku na obsluhu jeho přerušení). Do návrhové plochy vložíme nový stav, pojmenujeme ho „*LED is ON*“ a provážeme ho přechodem se stavem „*LED is OFF*“. Jako podmínku strážící tento přechod použijeme událost `TI1_OnInterrupt` importovanou z PE a zobrazenou v panelu programových komponent v sekci „*PE Conditions*“. Přiřazení této programové komponenty opět provedeme jednoduše tak, že ji uchopíme a pomocí myši přetáhneme nad plochu popisku dané hrany přechodu. Tím bude vytvořena nová speciální neveřejná programová komponenta podmínky zapouzdřující programový příznak, který bude nastavován v rámci ISR rutiny přerušení časovače a ten bude testován při strážení přechodu mezi stavy „*LED is OFF*“ a „*LED is ON*“. Ke hraně přechodu také přidáme akci (stejným způsobem jako jsme přidávali akci k inicializačnímu přechodu) nazvanou `Set_LED_ON`, v jejímž těle budeme nastavovat hodnotu výstupního pinu na úroveň LOW, čímž dojde k rozsvícení připojené LED diody.

j. Po dosažení stavu „*LED is ON*“ pak na další tik časovače přejdeme zpět do stavu „*LED is OFF*“. Jako akce přechodu bude opět volána funkce `Set_LED_OFF`, čímž zajistíme nastavení příslušné úrovně výstupního pinu a potažmo zhasnutí připojené diody. Výsledný diagram naší aplikace bude v tomto okamžiku vypadat následovně:



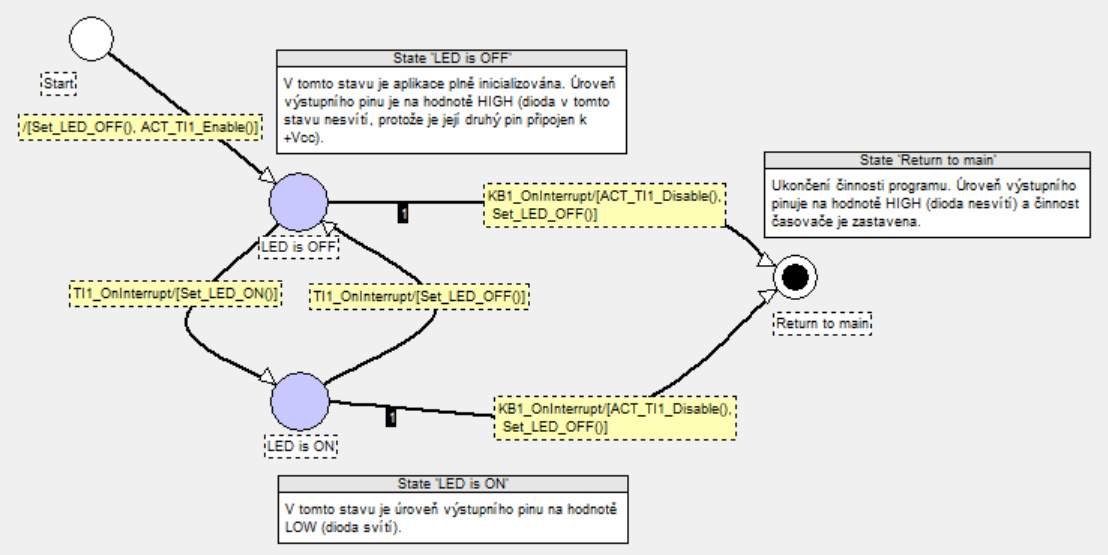
Obr. 76 První verze stavového diagramu

k. Následujícím krokem bude doplnění aplikační logiky programu o možnost přerušit a ukončit svou činnost. Tuto akci provedeme jako odezvu stisku tlačítka, tzn. jako odezvu na přerušení od klávesnice. Do diagramu vložíme koncový stav pojmenovaný „*Return to main*“. Do tohoto stavu přivedeme přechody z obou normálních stavů „*LED is OFF*“ a „*LED is ON*“. Strážící podmínkou těchto přechodů bude importovaná podmínková komponenta události přerušení od

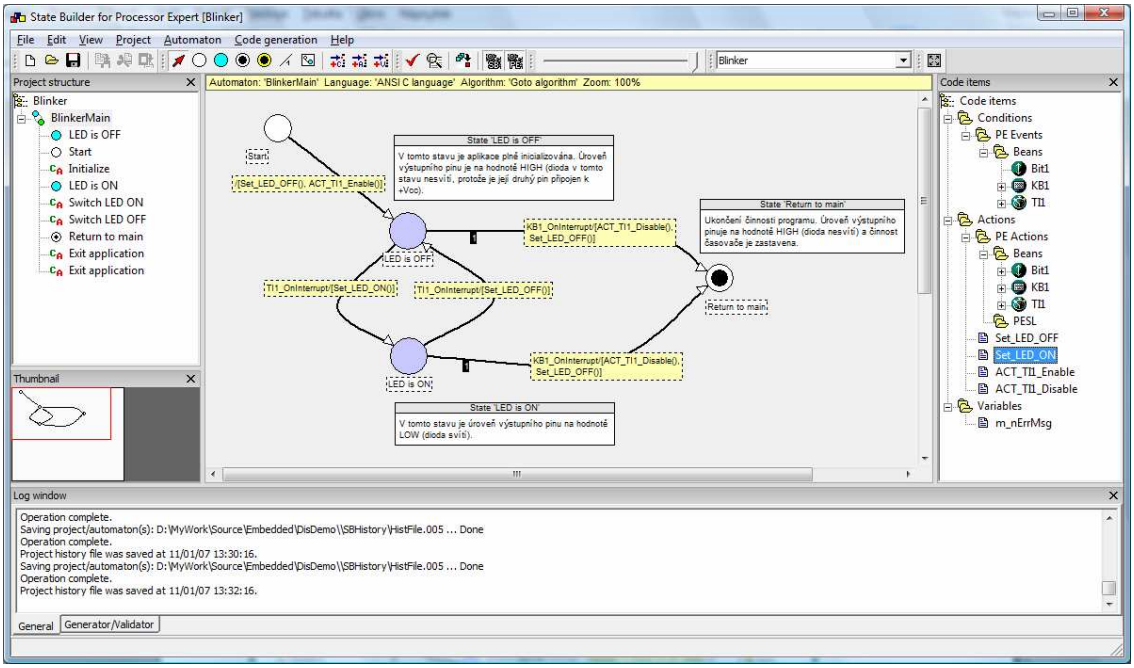
klávesnice (`KB1_OnInterrupt` ve složce *PE Events* panelu programových komponent) a akcí bude již dříve vytvořená uživatelská komponenta `Set_LED_OFF` a nová komponenta vytvořená přetažením metody `TI1_Disable` do popisku vlastností přechodu. Nově vzniklá komponenta `ACT_TI1_Disable` zajistí zastavení činnosti časovače a tím i generování jeho přerušení.

l. Ačkoliv se může zdát, že nastavení hran přechodů začínajících ve stavech „*LED is ON*“ a „*LED is OFF*“ je již úplné, opak je pravdou. Podmínka `KB1_OnInterrupt` by měla být testována vždy přednostně a proto musí mít přechod který stráží nejvyšší **prioritu**. To lze provést jednoduchým nastavením v dialogovém okně vlastností přechodu, které lze vyvolat buď prostřednictvím kontextové nabídky přechodu, nebo po dvojkliku levého tlačítka myši na plochu popisku přechodu. V zobrazeném okně pak na záložce „*Advanced*“ povolíme prioritu přechodu a její hodnotu nastavíme na 1 (čím nižší hodnota, tím vyšší priorita)

A to je z procesu tvorby stavového diagramu popisujícího naši embedded aplikaci vše! Výsledný vzhled diagramu je znázorněn na obrázku 77. Nyní již můžeme přistoupit k vlastnímu procesu generování zdrojového programového kódu vytvářené embedded aplikace.



Obr. 77 Kompletní stavový popis aplikace PulseGenerator

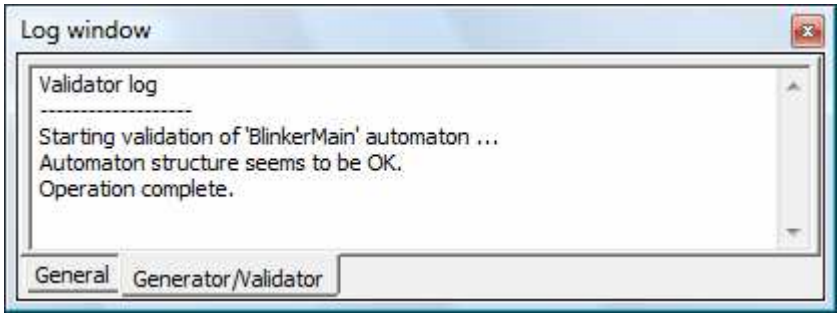


Obr. 78 Prostředí SB s dokončeným návrhem embedded aplikace

3.5.2.4 Generování programového kódu

Máme-li dokončený návrh stavového diagramu, můžeme přistoupit k závěrečnému**, třetímu kroku** vývojového procesu a tím je **generování zdrojového kódu aplikace**.

Před zahájením vlastního procesu generování programového kódu je nutné ověřit formální správnost návrhu. Tato operace je prováděna automaticky při zahájení procesu generování z aplikace SB (nebo při zahájení generování z aplikace PE, je-li v prostředí SB aktivováno vynucené generování SB projektu – viz kapitola 3.5.1.2 a 3.5.1.3), uživatel však může kontrolu provádět kdykoliv v průběhu návrhu stavového diagramu prostřednictvím položky menu *Automaton -> Validate*, nebo příslušného tlačítka na panelu nástrojů generátoru kódu. Je-li vše v pořádku, bude výstup generátoru zobrazovaný v okně zpráv vypadat následovně:



Obr. 79 Výstup validátoru stavového automatu

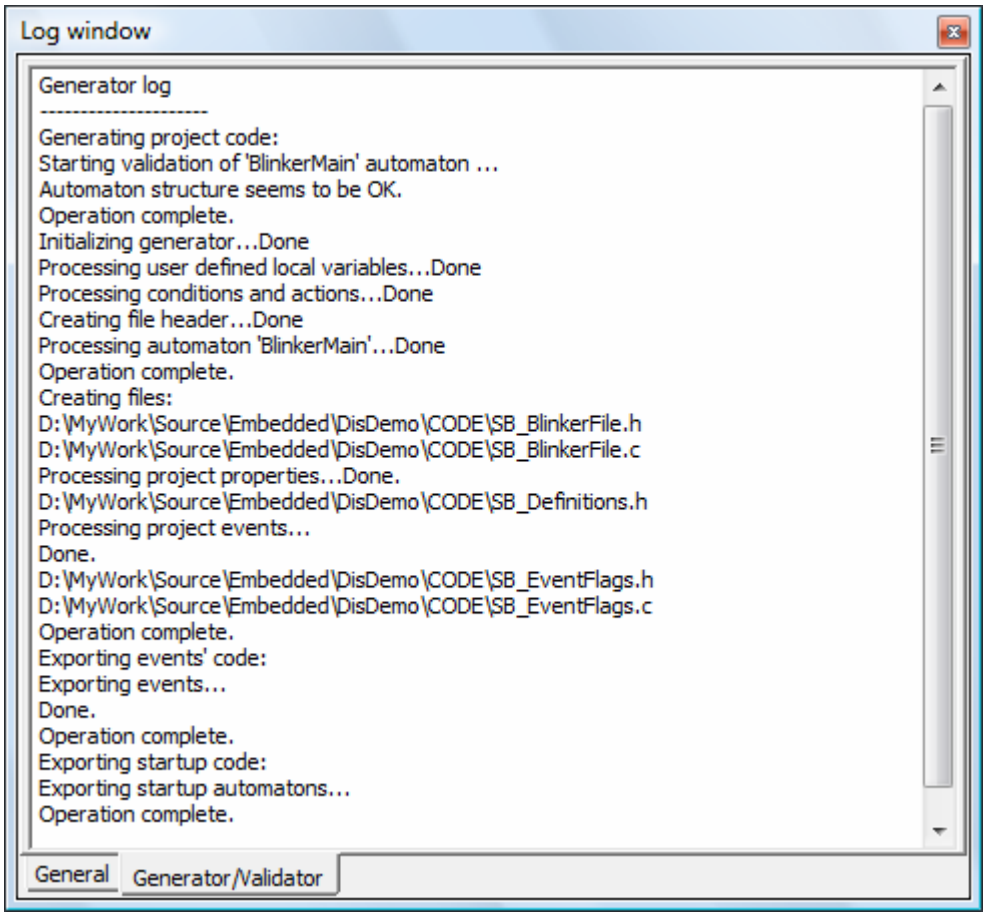
V opačném případě bude okno zpráv obsahovat informace o nalezených problémech.

Po úspěšné validaci, popřípadě opravách problematických stavů, můžeme přistoupit k vlastnímu generování programového kódu. To zahájíme pomocí položky menu *Code generation -> Generate project code*, klávesové zkratky *Ctrl+G*, nebo příslušného tlačítka na panelu nástrojů generátoru programového kódu.

V průběhu procesu generování programového kódu je vytvořeno několik nových programových souborů, které jsou přidány do PE projektu a zároveň jsou modifikovány již existující soubory, které byly vytvořeny prostředím PE. Všechen programový kód generovaný aplikací SB je ve zdrojových souborech „zabalen“ do programových značek (speciálních komentářů), což umožňuje jednak snadnou orientaci v generovaném kódu – je jasně vidět, které programové fragmenty přidal do zdrojového kódu sám uživatel a které vznikly generováním – a zároveň také umožňuje zpětnou synchronizaci programového kódu mezi PE a SB. Synchronizace programového kódu je užitečná tehdy, změní-li uživatel kód programových komponent stavového diagramu prostřednictvím editoru aplikace PE a chce, aby se tyto změny promítly také do vlastního stavového diagramu (synchronizovat lze pouze těla podmínek a akcí, ne strukturu stavového diagramu). V tom případě lze z prostředí aplikace SB spustit synchronizaci kódu prostřednictvím položky menu *Project -> Synchronize code*.

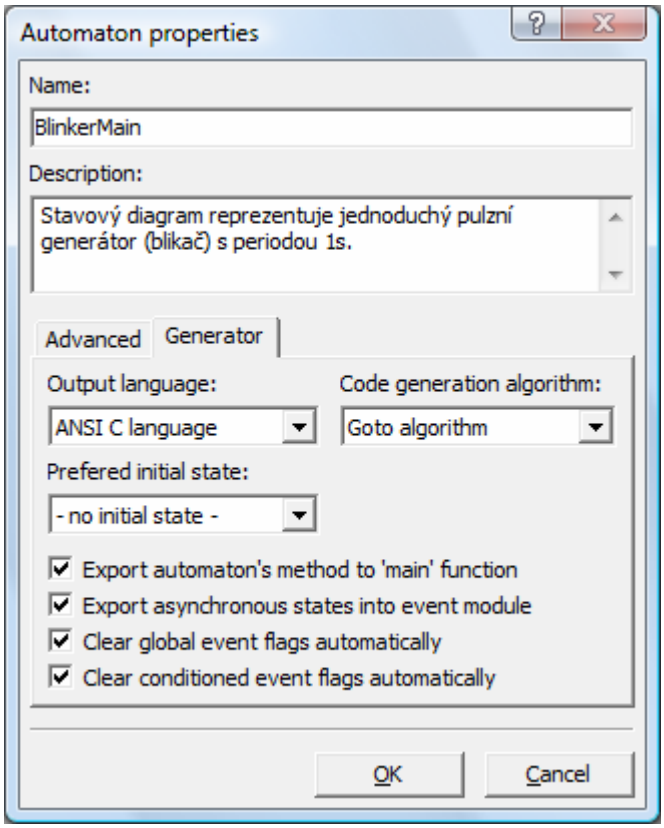
Obrázek 80 ukazuje výstup generátoru programového kódu při zpracování naší embedded aplikace.

Jak je vidět, v průběhu generování bylo vytvořeno 5 nových programových souborů. Soubory **SB_BlinkerFile.h** a **SB_BlinkerFile.c** obsahují programový kód generovaného stavového automatu, soubor **SB_Definitions.h** obsahuje deklarace uživatelských datových typů a soubory **SB_EventFlags.h** a **SB_EventFlags.c** obsahují deklarace a definice příznaků událostí testovaných v kódu stavového automatu (tyto příznaky jsou nastavovány v rámci ISR rutin příslušných přerušení). Vlastní rutiny přerušení jsou generovány aplikací PE (typicky do souborů **Events.h** a **Events.c**) a aplikace SB tyto rutiny pouze modifikuje. Obdobným způsobem je modifikována také hlavní funkce aplikace (`void main(void)`), generovaná aplikací PE. Je-li to požadováno (a ve vlastnostech automatu nastaveno), bude v průběhu generování kódu do funkce `main` vloženo volání funkce automatu, čímž zajistíme, že ihned po startu aplikace bude spuštěn také náš generovaný kód. Toto nastavení, stejně tak jako další nastavení ovlivňující formu generovaného kódu lze modifikovat pomocí dialogových vlastností stavových automatů, nebo celého SB projektu.



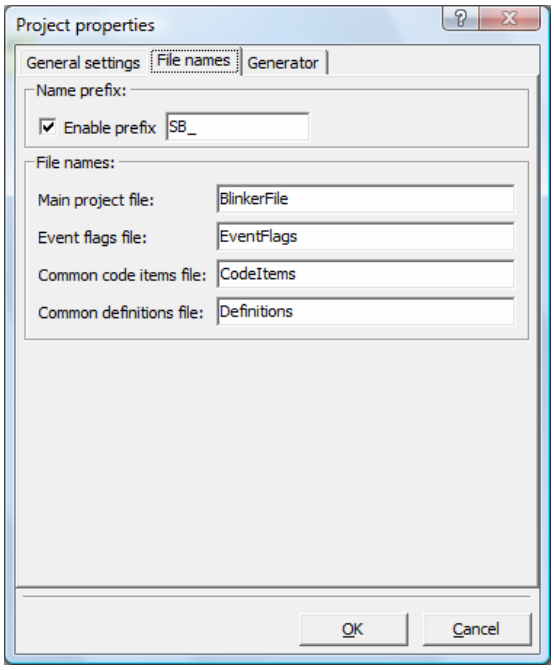
Obr. 80 Výstup generátoru programového kódu

Vlastnosti stavových automatů určují, jaký generující algoritmus bude pro daný automat použit, zda bude tento automat importován do hlavní funkce `main`, či jakým způsobem se bude pracovat s programovým kódem příznaků události a jejich akcí. Dialogové okno vlastností automatu lze vyvolat například prostřednictvím kontextové nabídky pracovní plochy, nebo dvojklikem levého tlačítka myši na komponentu automatu v okně panelu komponent SB projektu.

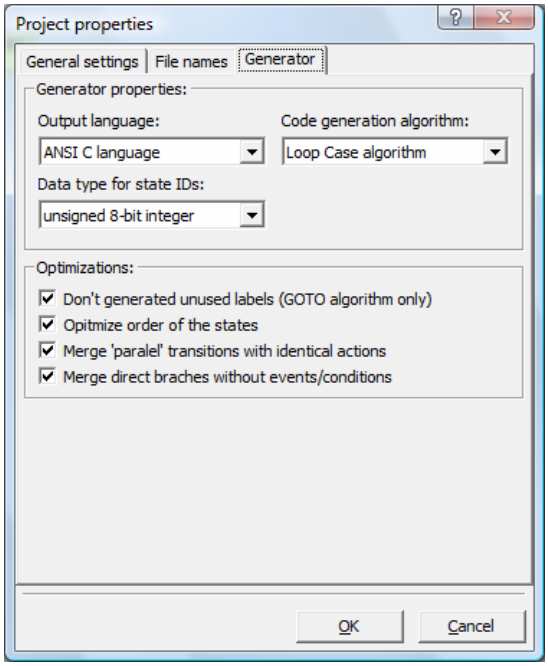


Obr. 81 Dialogové okno vlastností stavového automatu

Vlastnosti projektu ovlivňující proces generování programového kódu umožňují nastavit názvy vytvářených (generovaných) programových souborů a některé další vlastnosti, jakými jsou např. rozšířené možnosti optimalizace struktury zdrojového automatu i programového kódu, datový typ použitý pro označení stavů, upřednostňovaný generující algoritmus, výstupní programovací jazyk a podobně.



Obr. 82 Dialogové okno vlastností projektu (názvy generovaných souborů)



Obr. 83 Dialogové okno vlastností projektu (vlastnosti generátoru programového kódu)

Nyní si již ukažme programový kód, který vznikl generováním ze stavového automatu popsaného výše. Aby bylo naprosto zřejmé, jaký je rozsah generovaného kódu, je zde uveden kompletní výpis všech, pro nás podstatných, programových souborů PE projektu a je-li to potřeba, je programový kód generovaný aplikací SB zvýrazněn.

Hlavní soubor aplikace **Blinker.c** (generováno PE, modifikováno SB)

```
/* MODULE Blinker */

  /*** State Builder headers: begin of included headers. DO NOT
REMOVE THIS LINE. ***/
#include "SB_BlinkerFile.h"
  /*** State Builder headers: end of included headers. DO NOT
REMOVE THIS LINE. ***/

/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
#include "Bit1.h"
#include "KB1.h"
#include "TI1.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

void main(void)
{
  /*** Processor Expert internal initialization. DON'T REMOVE
THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.
***/

 /*** State Builder : begin of generated code. DO NOT REMOVE THIS
LINE. ***/
  BlinkerMain();
  /*** State Builder : end of generated code. DO NOT REMOVE THIS
LINE. ***/
  /* For example: for(;;) { } */
```

```
  /*** Don't write any code pass this line, or it will be deleted
during code generation. ***/
  /*** Processor Expert end of main routine. DON'T MODIFY THIS
CODE!!! ***/
  for(;;){}
  /*** Processor Expert end of main routine. DON'T WRITE CODE
BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/

/* END Blinker */
```

Hlavičkový soubor automatu **SB_BlinkerFile.h** (generováno SB)

```
/*** State Builder : begin of generated code. DO NOT REMOVE THIS
LINE. ***/

#ifndef _SB_BlinkerFile_h
#define _SB_BlinkerFile_h

/*****************************************************************
This part is implementation of BlinkerMain automaton
where ANSI C language and 'Goto algorithm' were used.
The code was generated by State Builder for ProcessorExpert
(c) 2004, 2005 Unis s.r.o., /MB
Automaton's description:
  Stavový diagram reprezentuje jednoduchý pulzní generátor
(blikač) s periodou 1s.
*****************************************************************/

#include "SB_Definitions.h"

/* used state IDs */
#define ID_LED_is_OFF 1
#define ID_Start 2
#define ID_LED_is_ON 3
#define ID_Return_to_main 4

/* conditions */

/* actions */

/* variables */
/* Návratová hodnota z funkcí PE API */
extern byte m_nErrMsg;
```

```
/* automaton */
TYPE_STATE BlinkerMain(void);

#endif //_SB_BlinkerFile_h

/*** State Builder : end of generated code. DO NOT REMOVE THIS
LINE. ***/
```

Implementační soubor automatu **SB_BlinkerFile.c** (generováno SB)

```
/*** State Builder : begin of generated code. DO NOT REMOVE THIS
LINE. ***/

/*****************************************************************
Header files
*****************************************************************/
#include "SB_BlinkerFile.h"
#include "SB_EventFlags.h"
#include "Bit1.h"
#include "KB1.h"
#include "TI1.h"

/*****************************************************************
This part is implementation of BlinkerMain automaton
where ANSI C language and 'Goto algorithm' were used.
The code was generated by State Builder for ProcessorExpert
(c) 2004, 2005 Unis s.r.o., /MB
Automaton's description:
  Stavový diagram reprezentuje jednoduchý pulzní generátor
(blikač) s periodou 1s.
*****************************************************************/

byte m_nErrMsg=0;

/*****************************************************************
Automaton code implementation.
*****************************************************************/
TYPE_STATE BlinkerMain(void)
{
  /* State: Start */
  /*** State Builder code: begin of 'Set_LED_OFF' action code. DO
NOT REMOVE THIS LINE. ***/
  // nastavení výstupní úrovně pinu na HIGH
  Bit1_SetVal();
```

```
  /*** State Builder code: end of 'Set_LED_OFF' action code. DO
NOT REMOVE THIS LINE. ***/
  /*** State Builder code: begin of 'ACT_TI1_Enable' action code.
DO NOT REMOVE THIS LINE. ***/
  m_nErrMsg = TI1_Enable();
  /*** State Builder code: end of 'ACT_TI1_Enable' action code.
DO NOT REMOVE THIS LINE. ***/

  /* State: LED is OFF */
  State_ID_LED_is_OFF:
  if( EVT_KB1_OnInterrupt )
  {
    EVT_KB1_OnInterrupt=0;
    /*** State Builder code: begin of 'ACT_TI1_Disable' action
code. DO NOT REMOVE THIS LINE. ***/
    m_nErrMsg = TI1_Disable();
    /*** State Builder code: end of 'ACT_TI1_Disable' action
code. DO NOT REMOVE THIS LINE. ***/
    /*** State Builder code: begin of 'Set_LED_OFF' action code.
DO NOT REMOVE THIS LINE. ***/

// nastavení výstupní úrovně pinu na HIGH
    Bit1_SetVal();
    /*** State Builder code: end of 'Set_LED_OFF' action code. DO
NOT REMOVE THIS LINE. ***/
    goto State_ID_Return_to_main;
  }
  else if( EVT_TI1_OnInterrupt )
  {
    EVT_TI1_OnInterrupt=0;
    /*** State Builder code: begin of 'Set_LED_ON' action code.
DO NOT REMOVE THIS LINE. ***/
    Bit1_ClrVal();
    /*** State Builder code: end of 'Set_LED_ON' action code. DO
NOT REMOVE THIS LINE. ***/
  }
  else
    goto State_ID_LED_is_OFF;

  /* State: LED is ON */
  State_ID_LED_is_ON:
  if( EVT_KB1_OnInterrupt )
  {
    EVT_KB1_OnInterrupt=0;
    /*** State Builder code: begin of 'ACT_TI1_Disable' action
code. DO NOT REMOVE THIS LINE. ***/
```

```
    m_nErrMsg = TI1_Disable();
    /*** State Builder code: end of 'ACT_TI1_Disable' action
code. DO NOT REMOVE THIS LINE. ***/
    /*** State Builder code: begin of 'Set_LED_OFF' action code.
DO NOT REMOVE THIS LINE. ***/
    // nastavení výstupní úrovně pinu na HIGH
    Bit1_SetVal();
    /*** State Builder code: end of 'Set_LED_OFF' action code. DO
NOT REMOVE THIS LINE. ***/
  }
  else if( EVT_TI1_OnInterrupt )
  {
    EVT_TI1_OnInterrupt=0;
    /*** State Builder code: begin of 'Set_LED_OFF' action code.
DO NOT REMOVE THIS LINE. ***/
    // nastavení výstupní úrovně pinu na HIGH
    Bit1_SetVal();
    /*** State Builder code: end of 'Set_LED_OFF' action code. DO
NOT REMOVE THIS LINE. ***/
    goto State_ID_LED_is_OFF;
  }
  else
    goto State_ID_LED_is_ON;

  /* State: Return to main */
  State_ID_Return_to_main:
  return ID_Return_to_main;
}


/*** State Builder : end of generated code. DO NOT REMOVE THIS
LINE. ***/
```

Hlavičkový soubor uživ. datových typů **SB_Definitions.h** (generováno SB)

```
/*** State Builder : begin of generated code. DO NOT REMOVE THIS
LINE. ***/

#ifndef _SB_Definitions_h
#define _SB_Definitions_h

#include "D:\MyWork\Source\Embedded\DisDemo\CODE\PE_Types.h"
```

```
/*****************************************************************
Common definitions
*****************************************************************/

/* Data type for IDs */
typedef unsigned char TYPE_STATE;

/* User defined types */

/* Definition of a PE properties */


#endif //_SB_Definitions_h
```

Hlavičkový soubor příznaků událostí **SB_EventFlags.h** (generováno SB)

```
/*** State Builder : begin of generated code. DO NOT REMOVE THIS
LINE. ***/

#ifndef _SB_EventFlags_h
#define _SB_EventFlags_h

/*****************************************************************/
/* Event flags declaration                                       */
/* This code was generated by State Builder for ProcessorExpert
*/
/*****************************************************************/

extern volatile unsigned char EVT_TI1_OnInterrupt;
extern volatile unsigned char EVT_KB1_OnInterrupt;

#endif //_SB_EventFlags_h

/*** State Builder : end of generated code. DO NOT REMOVE THIS
LINE. ***/
```

Implementační soubor příznaků událostí **SB_EventFlags.c** (generováno SB)

```
/*** State Builder : begin of generated code. DO NOT REMOVE THIS
LINE. ***/

/*****************************************************************/
/* Event flags definition                                        */
/* This code was generated by State Builder for ProcessorExpert
*/
/*****************************************************************/

unsigned char EVT_TI1_OnInterrupt=0;
unsigned char EVT_KB1_OnInterrupt=0;

/*** State Builder : end of generated code. DO NOT REMOVE THIS
LINE. ***/
```

Implementační soubor ISR rutin událostí **Events.c** (generováno PE, modifikováno SB)

```
/* MODULE Events */

#include "Cpu.h"
#include "Events.h"

  /*** State Builder headers: begin of included headers. DO NOT
REMOVE THIS LINE. ***/
#include "SB_EventFlags.h"
#include "SB_BlinkerFile.h"
  /*** State Builder headers: end of included headers. DO NOT
REMOVE THIS LINE. ***/

/*
**
===================================================================
**     Event       :  TI1_OnInterrupt (module Events)
**
**     From bean   :  TI1 [TimerInt]
**     Description :
**         When a timer interrupt occurs this event is called
(only
```

```
**         when the bean is enabled - <"Enable"> and the events
are
**         enabled - <"EnableEvent">). This event is enabled only
if
**         a interrupt service/event is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/
void TI1_OnInterrupt(void)
{
  /*** State Builder code: begin of 'TI1_OnInterrupt' event code.
DO NOT REMOVE THIS LINE. ***/
  EVT_TI1_OnInterrupt=1;
  /*** State Builder code: end of 'TI1_OnInterrupt' event code.
DO NOT REMOVE THIS LINE. ***/
  /* Write your code here ... */

}

/*
**
============================================================
**     Event       :  KB1_OnInterrupt (module Events)
**
**     From bean   :  KB1 [KBI]
**     Description :
**         This event is called when the active signal edge/level
**         occurs. This event is enabled only if Interrupt
**         service/events are enabled.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/

void KB1_OnInterrupt(void)
{
  /*** State Builder code: begin of 'KB1_OnInterrupt' event code.
DO NOT REMOVE THIS LINE. ***/
  EVT_KB1_OnInterrupt=1;
  /*** State Builder code: end of 'KB1_OnInterrupt' event code.
DO NOT REMOVE THIS LINE. ***/
  /* Write your code here ... */
}

/* END Events */
```

ZÁVĚR

Je nesporné, že oblast využití embedded zařízení a systémů je jednou z nejrychleji se rozvíjejících oblastí současné informatiky a spotřební elektroniky. Jak již bylo řečeno v úvodu této práce, embedded systémy lze nalézt v široké paletě současné spotřební elektroniky, řídících a monitorovacích systémů a lze předpokládat, že tempo růstu objemu nasazených embedded systému se v nejbližší době nezastaví.

Nejen z tohoto důvodu je nutné, aby vývoj softwarového vybavení embedded systémů nezaostával za rozvojem a inovacemi jejich hardwarových částí. Proto je důležité hledat a aplikovat nové a vylepšené metody tvorby programového vybavení určených pro embedded zařízení, které by umožňovaly vyvíjet rychleji, spolehlivěji a tím i levněji. Takový způsob vývoje se ve výsledku promítne nejen v užitné hodnotě daného embedded zařízení, ale také jeho cenové dostupnosti; vždyť je známou pravdou, že cena tvorby softwarové vybavení může být mnohem vyšší, než náklady vynaložené na pořízení, nebo vývoj hardwaru daného zařízení.

Smyslem této disertační práce bylo ukázat, že i v natolik specifické oblasti, jakou vývoj software pro embedded systémy bezesporu je, lze stále nalézat a uplatňovat nové postupy vedoucí ke zkvalitnění celého vývojového procesu.

Práce ukázala, jakým způsobem lze aplikovat vybrané formální metody návrhu a vývoje aplikaci na embedded systémy a jak lze z těchto formálních, a do značné míry platformě nezávislých popisů automatizovaně vytvářet produkční programový kód určený pro specifické hardwarové platformy embedded systémů.

Součástí práce je také stručný popis vytvořeného aplikačního systému a ukázka práce s ním. Tento dokument si v žádném případě nečiní nárok na to být vyčerpávající referenční uživatelskou příručkou nebo ucelenou metodikou, dle mého názoru však umožňuje dostatečně srozumitelně nahlédnout do řešené problematiky a může sloužit jako dobrý výchozí bod pro další výzkum a bádání v oblasti dané problematiky.

POUŽITÁ LITERATURA A DALŠÍ ZDROJE

[1] Qing Li, Real-Time Concepts for Embedded Systems, CMPBooks, 2003, ISBN 1-57820-124-1

[2] Barr, M., Programming Embedded Systems, O'Reilly, 1999, ISBN: 1-56592-354-5

[3] Timothy K, Synthesis of Finite State Machines: Functional Optimization. Kluwer Academic Publishers, Boston 1997, ISBN 0-7923-9842-4

[4] Carroll, J., Long, D., Theory of Finite Automata with an Introduction to Formal Languages. Prentice Hall. Englewood Cliffs, 1989.

[5] Chytil, M., Automaty a gramatiky, SNTL, 1984, 04-012-84

[6] Kocur, P., Úvod do teorie konečných automatů a formálních jazyků, Západočeská univerzita va Plzni, 2001, ISBN 80-7082-813-7

[7] Arlow, J., Neustadt, I., UML a unifikovaný process vývoje aplikací, Computer Press, 2003, ISBN 20-7226-947-X

[8] Bližňák, M., Systémové programování, Univerzita Tomáše Bati ve Zlíně, 2005, ISBN 80-7318-364-1

[9] Harel., D., Statecharts: A visual formalism for complex systems, Science of Computer Programming, 8(3):231--274, June 1987

[10] Douglass, B. P., Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley, 1999, ISBN 0-201-49837-5

[11] UML (http://www.uml.org/)

[12] Wikipedia, Finite State Machine (http://en.wikipedia.org/wiki/Finite_state_machine)

[13] Quantum Leaps (http://www.quantum-leaps.com/products/index.htm#Overview)

[14] SME (http://www.programmersheaven.com/2/Design-State-Machine-Engine)

[15] SMC (http://smc.sourceforge.net/)

[16] StateWizard (http://www.intelliwizard.com/)

[17] ArgoUML (http://argouml.tigris.org/)

[18] Poseidon for UML (http://www.gentleware.com/)

[19] Enterprise Architect (http://www.sparxsystems.com/)

[20] VisualState (http://www.iar.com/index.php?show=1014_ENG&reflogin=1014_ENG)

[21] Processor Expert (http://www.processorexpert.com)

[22] Rational Suite Development Studio Real-Time (Rational Software Corp., www.rational.com)

[23] BetterState (WindRiver Systems, www.wrs.com)

[24] Stateflow (MathWorks, www.mathworks.com)

[25] QNX RTOS (http://www.qnx.com/)

[26] LINUXWORKS RTOS (http://www.lynuxworks.com/)

[27] FUSION RTOS (http://www.unicoi.com/index.html)

[28] KATIX RTOS (http://www.funet.fi/~kate/katix.html)

[29] VxWORKS RTOS (http://www.windriver.com/products/platforms/general_purpose/index.html)

[30] CodeWarrior (http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=012726)

[31] wxWidgets (http://www.wxwidgets.org)

[32] wxDesigner GUI editor (http://www.roebling.de/)

[33] DialogBlocks GUI editor (www.anthemion.co.uk/dialogblocks/)

[34] wxGlade GUI editor (http://wxglade.sourceforge.net/)

[35] VisualWx IDE (http://visualwx.altervista.org/)

[36] wxDev-Cpp IDE (http://wxdsgn.sourceforge.net/)

[37] Knihovna wxAUI (http://www.kirix.com/community/wxaui/screenshots.html)

[38] Programmers heaven (http://www.programmersheaven.com/)

PUBLIKAČNÍ ČINNOST

1) Bližňák, M., Mikropočítačový systém pro evidenci počtu kopií na rozmnožovacím stroji, STOČ ´1998, Ostrava
2) Bližňák, M., Real-time aplikace řídicích a monitorovacích systémů pod OS MS Windows (součást grantového projektu Fondu rozvoje VŠ MŠMT G1/1503/2000), STOČ' 2001, Ostrava
3) Vasek, V., Bliznak, M., Program For Interactive Models Control In The Real Time Conditions, PROCESS CONTROL 2002, Pardubice
4) Bliznak, M., Vasek, V., Janacova, D., WCONTROL – Program System for Control Theory Laboratory Education, ICEE, Valencia, ESP, 2003
5) Bliznak, M., Dulik, T., A Real-Time Advanced Control Application, DAAAM, Vienna, 2003, Austria
6) Bliznak, M., Dulik, T.: The Program Tool for Control Theory Laboratory Education, IADAT (International Association for the Development of Advances in Technology) - e2004, in conference proceedings, page 377-381, ISBN 84-933971-0-5, Bilbao, Spain, July 7-9 2004
7) Bližňák, M.: Systémové programování, Univerzita Tomáše Bati ve Zlíně, ISBN 80-7318-364-1, Zlín, 2005
8) Bliznak, M., Kolar, D.: Formal-method-based Software Development Applied on Embedded Systems: Platform-independent Source Code, MITIP 2006, in conference proceedings, page 487-492, ISBN 963-86586-5-7,Budapest, 2006
9) Bliznak, M., Kolar, D.: Formal-method-based Software Development Applied on Embedded Systems: Basic concepts, 17th International DAAAM Symposium 2006, in conference proceedings, page 45-46, ISBN 3-901509-57-7, Vienna, 2006
10) Bližňák, M., Kolář, D.: Formální metody návrhu software aplikované na embedded systémy: Platformně nezávislý zdrojový kód (první část), AT&P Journal, číslo 12/06, strany 69-72
11) Bližňák, M., Kolář, D.: Formální metody návrhu software aplikované na embedded systémy: Platformně nezávislý zdrojový kód (druhá část), AT&P Journal, číslo 01/07, strany 57-58, ISSN 1335-2237, Bratislava, SR, 2007
12) Bližňák, M., Kolář, D.: Formální metody návrhu software aplikované na embedded systémy: Platformně nezávislý zdrojový kód (třetí část), AT&P Journal, číslo 02/07, strany 57-58, ISSN 1336-5010, Bratislava, SR, 2007

CURRICULUM VITAE

Jméno:	**Ing. Michal Bližňák**
Datum narození:	3.6.1977
Stav:	svobodný
Státní občanství:	Česká Republika
Národnost:	česká
Telefon:	+420604506619
E-mail:	bliznak@fai.utb.cz
Adresa:	Květná 432, Slavičín 76321

Vzdělání:

Od 2001 — Student doktorského studia v oboru Technická kybernetika na Univerzitě Tomáše Bati ve Zlíně.

1999 – 2001 — Vysoké učení technické v Brně / Univerzita Tomáše Bati ve Zlíně

Fakulta Technologická

- Obor: Automatizované systémy řízení technologických procesů
- Diplomová práce: Real-time řídící a monitorovací systém pro Windows NT/2000

1995 – 1998 — Vysoké učení technické v Brně, Fakulta Technologická ve Zlíně

- Obor: Automatizace a informatika
- Bakalářská diplomová práce: Elektronický počítačový systém pro evidenci užívání kopírky

1991 – 1995 — Střední průmyslová škola elektrotechnická v Brně

- Obor: Elektronické počítačové systémy na bázi jednočipových mikropočítačů

Jazykové znalosti:

Angličtina: výborná pasivní znalost (schopnost studia odborných anglických textů), dobré komunikativní schopnosti.

Ruština: základy

Odborná praxe:

Od 2003 UTB Zlín, FT, Institut Informačních Technologií, Zlín

Asistent

- Řešení problematiky řízení technologických procesů v reálném čase, real-timové systémy
- Výzkumná a vývojová činnost v oblasti embedded systémů
- Výzkumná a vývojová činnost v oblasti tvorby software pro desktopové a embedded systémy a paralelní výpočetní systémy
- Aplikace formálních jazyků a metod na vývojový proces SW
- Pedagogická činnost

2001 - 2003 VTÚVM Slavičín, s.p., Slavičín

Programátor, analytik

- Zpracování systému elektronické dokumentace zbraňových systému.
- Tvorba animací zbraňových systémů.
- Překlady obranných standardů NATO

1999 – 2001 Meditronic, s.r.o., Zlín

Programátor, technik

- Vývoj urodynamického diagnostického lékařského systému
- Řešení zpracování a analýzy videosekvencí
- Technická a servisní činnost

Profesní dovednosti:

- Programování v jazycích Assembler, Pascal, ANSI C/C++, Delphi, Visual C++, C++/CLI, Visual C#, SQL, Java, Python …
- Programování a tvorba WWW (PHP, MySQL, ASP.NET, JavaScript, HTML,…)
- Programování 3D grafiky s použitím rozhraní OpenGL API
- Znalost technologií .NET, Mono, Java, wxWidgets, wxPython, atd…
- Tvorba instalačních balíků (InstallShield Express, InstallShield Developer, Inno Setup, …)
- 2D grafika (Adobe Photoshop, Corel Draw, GIMP, …)
- 3D grafika (trueSpace, 3DStudio, Blender …)
- Znalost problematiky jednočipových mikropočítačů a PLC systémů, technických prostředků automatizace a teorie automatizovaného řízení.