ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ „АВТОМАТИКА”

Катедра „Системи и управление”

инж. Иван Евгениев Иванов

ДИСЕРТАЦИЯ

ГЕНЕРИРАНЕ НА УПРАВЛЯВАЩИ ПРОГРАМИ ВЪЗ ОСНОВА НА КОМПОНЕНТНИ СПЕЦИФИКАЦИИ

Шифър 02.21.06

„Автоматизация на инженерния труд и системи за автоматизирано проектиране”

НАУЧЕН РЪКОВОДИТЕЛ

Доц. д-р Христо Кирилов Ангелов

София
2005

Глава 1 – Обзор на компютърни системи за управление в реално време, средства и методи за спецификация и генерация

1.1 Увод

От началото на 80-те години на миналия век започна бурно развитие на компютърната техника, свързано с появата на микропроцесорите и все по-бързото развитие на електронните технологии. В началото на 90-те години компютърните мрежи все още представляваха малка и високоспециализиран част на компютърните системи. Днес това е основата на модерните технологии от всеки вид. "Законът на Мур" (Moore), който гласи, че на всеки 18 месеца плътността на микропроцесорните чипове ще се удвоява е все още валиден. Идеи и математически разработки, които са били невъзможни за реализация само преди няколко години, днес са напълно възможни. Тези постижения обусловиха развитието на компютърните системи за управление в този период. Нарасналите изчислителни възможности на модерните компютри, високата пропускателна способност на компютърните мрежи, повишеното качество и понижени цени на сензори и изпълнителни механизми позволиха повсеместното използуване на компютърно управление на всевъзможни машини, прибори и апарати. Нарастваща популярност добива терминът "pervasive computing"[152], отразяващ създаването на принципно нова основа на ежедневния живот. В областта на компютърните системи за управление (КСУ) освен известните от десетилетия индустриални КСУ, предназначени за управление на промишлени инсталации от различни типове, се оформи и ново направление – "вградени системи за управление (ВСУ)" (embedded systems). Количеството на последните като вид, предназначение и сложност, нарастват все повече и те се оформиха като самостоятелно научно и приложно направление. Разликата между долните нива на промишлените КСУ и ВСУ практически се размива все повече, тъй като големите многонивови (multi-tier) системи за автоматизация са базирани на редица компоненти от тип "вградени", напр. "интелигентни датчици", "интелигентни изпълнителни механизми" и др. подобни. Основната разлика се проявява в това, че ВСУ се намират в устройства, които преди всичко не са компютри. Това са превозни средства, телефони, пейджъри, аудио екипировка, самолети, джобни устройства, играчки, охранителни системи, игри, оръжия, медицински диагностични прибори, телевизори, превключватели за мрежи, принтери, скенери, климатизатори и т.н. ВСУ се свързват с физическият свят, взаимодействайки направо със сензори и изпълнителни механизми . Според [123] днес хората взаимодействат редовно с много повече ВСУ, отколкото с конвенционални компютри.

Друга основна тенденция, характеризираща развитието на съвременните КСУ е все по-широкото разпространение на компютърните мрежи и изградените на тяхна основа разпределени системи за управление (РСУ). В много случаи РСУ може да се разглежда като съвокупност от взаимодействуващи локални системи за управление, функциониращи в реално време. В този смисъл методологиите за изграждане на КСУ могат да бъдат обобщени и за случаите на разпределени системи за управление. Това обаче е свързано с допълнителни проблеми, произтичащи от необходимостта за осъществяване както на локални, така и на "отдалечени" взаимодействия между подсистемите на РСУ и съответните изграждащи ги софтуерни компоненти. При това трябва да се отчитат редица специфични изисквания, свързани със спазването на времеви ограничения, прозрачност и надеждност на взаимодействията и т. н. Характерът на тези изисквания зависи съществено от архитектурите на РСУ, които понастоящем се отличават със значително многообразие и висока степен на сложност.

Особен интерес предизвикват специфичните проблеми, които възникват при проектирането на различните видове РСУ и преди всичко - на хоризонтално-интегрираните системи, характеризиращи се с критични по време

взаимодействия между подсистемите им. При това, по-специално внимание се отделя на въпросите, свързани с изграждането на разпределени програмни системи въз основа на различни видове конфигурационни спецификации.

Някои от най-характерните области на приложение на КСУ и ВСУ включват:

- битова техника;
- автомобилна електроника;
- електромедицинска апаратура;
- автоматизация на лабораторни експерименти;
- автоматизация на технологични обекти и процеси;
- управление на енергийни обекти и системи;
- мониторинг и управление на екосистеми и природни ресурси;
- комуникационни системи;
- транспортни обекти и системи;
- военно-технически обекти и системи;
- авио-космически обекти и системи.

Така направеното (макар и непълно) изброяване илюстрира широкото разпространение на КСУ и следователно - изключително голямото значение, което те имат за съвременното общество. Както се изтъква в [168], системите за реално време (които в по-голямата си част се отъждествяват с КСУ) представляват съществен елемент от промишлената, транспортната и отбранителната инфраструктура на съвременните технологично-развити държави.

Според различни пазарни проучвания до 2002-а година са продадени повече ВСУ от конвенционални компютри. Този нов пазар включва огромно количество стоки и изделия, които заемат над 98% от пазара на компютърна техника. Според същите проучвания тези устройства изискват комплексен дизайн на електрониката и интеграция на система, реализирани в тесните граници, поставяни за разработка на нови пазарни изделия. В анализ на Samsung от 2003 г., около 57% от времето за разработка на ново изделие от този тип се падат на системната спецификация и съвместен дизайн на софтуера и хардуера и 42-43% на прототипиране и тестване на изделието.

1.2 Класификация и характеристики

Цялостният анализ и класификация на видовете КСУ е обемиста задача, стояща извън обхвата на тази работа. На нея са посветени студии [115][116][117][193], както и изследвания на IFAC и ISO – стандарт *ISO/TR 10314-1,* работен документ *ENV 40 003* и др. Тук ще бъде направена една най-обща класификация на базата на някои конструктивни и функционални признаци, ориентирано към задачата на дисертационния труд.

В конструктивно отношение КСУ се подразделят на вградени (embedded) и самостоятелни (stand-alone) системи. В първия случай КСУ се проектира и изгражда като съставен елемент на управлявания обект - специализирана апаратура, машина, транспортна единица и пр. Във втория случай КСУ е самостоятелна подсистема, която най-често има блочно-модулна структура.

Във функционално отношение КСУ се подразделят на специализирани системи и системи със стандартизирани, в рамките на някаква приложна област, функции. Връзката между направените класификации е очевидна. Встроените системи по правило са специализирани, докато самостоятелните системи са ориентирани към клас от приложения, характеризиращи се с типизирани и унифицирани функции на управлението. Типичен пример в това отношение са автоматизираните системи за управление на производството (АСУП), които се характеризират с масово разпространение и най-висока степен на унификация на управляващите функции [32][150][157][190].

АСУП на свой ред могат да бъдат подразделени на :

- системи за управление от горно ниво, в това число:

- управление на стопанската дейност,
- управление на производствената дейност;

- системи за управление от долно ниво, в това число:
  - събиране и предварителна обработка на данни,
  - супервизорно управление на технологични подсистеми,
  - пряко цифрово управление на технологични обекти;
- интегрирани системи за комплексна автоматизация на производството.

Системите за пряко цифрово управление на свой ред се подразделят на няколко вида в зависимост от особеностите на управляваните обекти: системи за управление на непрекъснати обекти, системи за управление на дискретни обекти и системи от дискретно-непрекъснат тип (хибридни системи), характеризиращи се с комбиниран механизъм на функциониране на управляващото устройство и/или обекта за управление.

Друг важен клас системи за управление, получили широко разпространение, са т. нар. системи за диспечерско управление (supervisory control and data acquisition systems - SCADA). Тези системи се използват в транспортните и енергийни инфраструктури, при управлението на големи разсредоточени обекти и т.н.

Многообразието на КСУ затруднява не само класифицирането, но и характеризирането им. Независимо от това, може да се посочат някои общи техни характеристики:

- Компютърните системи за управление са реактивни системи, работещи в реално време. Функционирането на подобни системи се изразява в своевременното възприемане и реагиране на събитията, възникващи в околната среда и по-специално, в управлявания обект. При това, реакциите на компютъра трябва да бъдат формирани с отчитане на определени времеви граници (deadlines), произтичащи от динамиката на процесите, протичащи в околната среда. В някои случаи - напр. при прякото цифрово управление на обекти и процеси, тези времеви граници са задължителни (твърди), като нарушаването им води до тежки, а понякога и катастрофални последици. Системите от този клас се означават като системи с "твърди" времеви ограничения (hard real-time systems). В други случаи, напр. в АСУП от горно ниво, информационни системи и др., времевите граници се дефинират като препоръчителни величини, чието спазване е желателно (но не и задължително), с оглед на осигуряване на необходимото средно време за реагиране на съответните събития (soft real-time systems). В практиката най-често се срещат системи от смесен тип, в които част от реакциите са с твърди времеви граници (напр. реагиране на събития в обекта за управление), а други - с незадължителни времеви граници (напр. при взаимодействие с операторите от различните нива на КСУ).

Реактивните системи като начин на работа се противопоставят на интерактивните системи [87][159], които реагират на околната среда със собствената си скорост и на трансформационните системи, които просто получават входна информация и я преобразуват до изходни данни. За разлика от трансформационните системи, при реактивните системи типично няма състояние “край на работа” (освен при повреда) .

- Друга характерна особеност на системите за пряко цифрово управление, респ. на използваните управляващи компютри е изискването за висока надеждност, което се постига чрез използване на специални видове интегрални схеми и защитени конструкции, а така също - на отказоустойчиви архитектури с резервиране на апаратните и програмните средства, в т.ч.:
  - архитектури, които запазват пълна работоспособност при настъпване на единични или кратни откази в системата (fail-safe архитектури);
  - системи с постепенна деградация, които запазват частична (редуцирана) работоспособност при настъпване на откази (fail-operational архитектури).

Използването на отказоустойчиви архитектури несъмнено усложнява и оскъпява КСУ, поради което намира приложение при реализацията на системи за управление на специални и високоотговорни обекти (напр. в авиацията, космонавтиката, ядрената енергетика и др.).

При тях отделните им функционални подсистеми (напр. автопилота в самолет) имат собствена вградена резервираща част. Отделните подсистеми не споделят ресурси и това повишава допълнително надеждността на цялото устройство. Въпреки това резервирането е скъп подход (защото всяка подсистема има собствена дублираща система) и все повече приложения минават към по-цялостни решения, където някои ресурси са поделени между различни подсистеми. В промишлени условия се разчита на технологични способи за повишаване на надеждността, в съчетание с ръчно управление и някои прости форми за резервиране (напр. аналогови резервни регулатори за по-важните контури). При необходимост, обаче, се използват по-сложни методи за резервиране, осигуряващи нужната степен на надеждност.

Все повече КСУ представляват разпределени системи за управление с интензивен обмен на променливи на състоянието между отделните компютърни елементи на системата. Това налага ограничения от времеви тип върху начина на работа на комуникационната среда. От тази гледна точка разпределените hard real-time системи пораждат обобщено изискване към адекватно време за реакция, което включва както собственото време за реакция на всеки компютърен елемент (възел), така времето за предаване на данните от един към друг възел. Това предполага както ново развитие на теорията на компютърните системи за управление, така и нов подход при конструирането, валидирането, настройката и пускането в експлоатация на системи от този вид.

1.3 Архитектури на КСУ

Развитието на различни архитектури за системи за управление е активна област на проучвания в последните две десетилетия. В тях термините архитектура и среда (framework) често се използват взаимозаменяемо. В тази работа те ще бъдат разграничавани. Терминът “архитектура” ще се използва за описание на дизайна и структурата на система. Терминът среда (framework) ще бъде отнасян към нещо външно за архитектура, което служи за нейното създаване.

Различните архитектури на управляващи системи адресират широк кръг от приложения. ***Това поражда и големи вариации на терминологията***. Има различия в терминологията при описанието на вградени системи и системи за промишлена автоматизация, например. Тук ще се ползува по-обща терминология, характерна повече за областта на теория на управлението (control engineering), отколкото за областта на компютърните науки (computer science). Още тук е мястото да бъде посочено, терминът ***компютърна система за управление*** се използва в смисъла на ***компютърно-реализирано управляващо устройство*** в рамките на една система за управление, когато последната се разглежда като двойката <обект на управлението, управляващо устройство>. В тази работа терминът ***компютърна система за управление*** ще бъде използван в този смисъл, с цел да не бъде нарушаван вече утвърдения му смисъл.

Изборът на ***архитектура***, т.е. на ***колекция от компоненти***, които могат да бъдат или апаратни, или програмни, които са реконфигурируеми или адаптируеми, е един от най-важните етапи на дизайна. [119]

Архитектурите за компютърно-подпомагано-проектиране (Computer Integrated Manufacturing-CIM) и за вградени приложения все още са различни, въпреки вече посоченото приближаване в долните нива на CIM системите и вградените системи.[38][40]

Разглеждането на съвременните тенденции в развитието на архитектурата на КСУ ще бъде направено в контекста на двата основни класа системи: вградени (embedded) системи и самостоятелни (блочно-модулни, stand-alone) системи.

При вградените контролери се забелязват две тенденции:

- изграждане на модулни системи на базата на стандартни системни магистрали от рода на VME, MULTIBUS I, MULTIBUS II, STD и др. п., в това число мултипроцесорни и мултикомпютърни системи. Специално внимание заслужава стандартът FUTUREBUS+ [165], разработен с оглед изискванията на системите за реално време, в който се предвиждат средства за приоритетно диспечиране на магистралата, предсказуема комуникация между отдалечени процеси, поддържане на общо астрономическо време чрез синхронизация на часовниците на отделните процесорни модули и пр.
- изграждане на едноплаткови контролери с възможност за навесно монтиране на допълнителни схемни модули - В/И контролери, устройства за връзка с обектите, RAM/ROM модули и пр. С оглед на унифицирането на тези модули, са въведени стандарти за локални магистрали (mezanine buses) за едноплаткови контролери, напр. PC 104, PCMCIA и др.

Блочно-модулните системи с промишлено предназначение (Computer Integrated Manufacturing - CIM) представляват слабо-свързани мултикомпютърни системи (мрежи) , чрез които се реализират разпределени системи за управление (РСУ), съставени от специализирани функционални подсистеми - непрекъснати и дискретни (програмируеми) контролери, хибридни (многофункционални) контролери, операторски и инженерни станции и пр. [150][157]. В преобладаващата си част това са децентрализирани системи за управление, чиито първообраз е известната система TDC 2000 на Honeywell. В съвременните системи от този клас са налице тенденции за "отваряне" на архитектурата, които са очевидни в образците на РСУ от последното десетилетие, напр. DCI System 6 на Fisher & Porter, в т. ч.:

- използване на стандартна мрежова архитектура, напр. Ethernet, в съчетание с TCP/IP, вместо на затворена фирмена мрежа;
- клиент-сървър взаимодействия между подсистемите на РСУ;
- използване на стандартни операционни среди (Unix, Windows, Linux) в операторските станции, с произтичащите от това възможности за интегриране на широко-разпространени програмни продукти за бази данни, електронни таблици и др.

През последните години идеологията на РСУ получи по-нататъшно развитие, което се изразява в създаването на няколко вида системни архитектури, поддържащи не само вертикални взаимодействия (както при децентрализираните системи), но и хоризонтални взаимодеиствия между подсистемите на РСУ, в т. ч.:

- системи за управление с отдалечен вход/изход [139][175];
- разпределени системи за управление, изградени на базата на мрежи от вида Fieldbus, CAN (Controller Area Network), ModBus и Sensor Bus [72][160].

Архитектурата на съвременните КСУ с промишлено предназначение се характеризира с още една особеност, първоначално породена от чисто конюнктурни съображения, но в последствие станало обичайна практика. Тя се състои в масовото приложение на персонални компютри тип IBM PC (крайно остарял термин), PC-съвместими промишлени контролери и операторски станции. Това се обяснява не толкова с техническите предимства на този тип системи, колкото с масовото им разпространение и наличието на огромен фонд от програмно осигуряване, приложимо в една или друга степен за целите на КСУ - както системно (DOS и основно ROMDOS за бездискови системи, Windows, компилатори за различни езици за програмиране

и т.н.), така и приложно - пакети за изграждане на графични интерфейси, бази данни, електрони таблици и пр. [78][79][94][139][192]

В заключение следва да се отбележи, че разграничението между подходите за изграждане на вградени и самостоятелни системи е донякъде условно. Това е така, тъй като от една страна, функционалните блокове на промишлените (stand-alone) системи се изграждат като едноплаткови и по-често - като многоплаткови конструкции, базирани на стандартна системна магистрала (в това число мултипроцесорни и мултикомпютърни системи). От друга страна, вградените системи за управление на сложни обекти в авиацията, военните приложения и др. се реализират като управляващи мрежи, въз основа на съответния стандарт, напр. MIL-STD 1553, MIL-STD 1753 и др. [157]. В отделни случаи могат да бъдат използувани и по-сложни архитектури, и по-специално - мултипроцесори или мултикомпютри, реализирани на базата на директни или индиректни съединителни мрежи (interconnection networks), напр. транспютърни системи .

По отношение на развитието на разпределените системи за управление съществува голямо многообразие от системни архитектури, които могат да бъдат групирани в два основни класа: вертикално-интегрирани и хоризонтално-интегрирани системи.

Първият тип РСУ представляват децентрализирани информационно-управляващи системи, характеризиращи се с йерархична функционална организация и вертикален информационен обмен между подсистемите от различните йерархични нива - промишлени контролери, подсистеми за събиране на данни, управляващи компютри от горно ниво, операторски станции. Споменатите подсистеми се реализират като автономни (self-contained) компоненти на РСУ, които си взаимодействуват епизодично, обменяйки си информация чрез мрежата. Така описаната системна архитектура е получила масово разпространение и се поддържа от голям брой производители, като Honeywell, Telemechanics, Siemens, ABB и др. [157].

В рамките на тази системна архитектура, но на другия Ł полюс (по отношение на цена и функционални възможности) са системите с висока степен на децентрализация. В този случай РСУ включва определен брой прости регулатори (1-2 контура), изградени с едночипови микроконтролери и операторска станция, реализирана на базата на персонален компютър от определен клас.

През последните десетилетия идеологията на РСУ получи по-нататъшно развитие, което се изразява в изграждане на “същински” РСУ, т.е. на хоризонтално-интегрирани архитектури, поддържащи не само вертикални, но и хоризонтални взаимодействия между подсистемите на РСУ, в т. ч.:

- системи за управление с отдалечен вход / изход;
- клиент-сървър системи, изградени на базата на персонални компютри и техни промишлени аналози;
- системи, изградени на базата на промишлени мрежи тип Fieldbus, CAN и др.

Увеличаването на изчислителната мощност и експлоатационната надеждност на персоналните компютри прави възможна реализацията на системи за управление с отдалечен вход / изход . В тази архитектура микропроцесорните контролери се редуцират до интелигентни В/И устройства (аналогови, дискретни и число-импулсни входове и изходи), а управляващите функции се поемат от персоналния компютър, който изпълнява ролята и на операторска станция [175]. Изграждането на системи с отдалечен вход / изход се облегчава и от наличието на подходящи комуникационни магистрали и стандарти, напр. Genius на General Electric и др. [4][5].

Така-разгледаната системна архитектура е разпределен аналог на класическите централизирани системи за пряко цифрово управление. В този смисъл, възможностите Ł са ограничени от изчислителната мощност на централния компютър. В някои системи за автоматизация на промишлени процеси това ограничение се премахва чрез използуването на мрежови

клиент-сървър архитектури [92][93]. Типичен пример в това отношение е системата Paragon TNT [93], реализирана въз основа на мрежа, съставена от персонални компютри (PC) и PC-съвместими промишлени контролери, в които се формира и поддържа разпределена база данни. Подсистемите обменят информация помежду си посредством клиент-сървър взаимодействия, осъществявани чрез приложното ниво на протокола Common Resource Access (CRA). При това е възможна както хоризонтална, така и вертикална комуникация между подсистеми от рода на операторска станция (клиент); процесен вход/изход (сървър); управляваща подсистема (сървър); сървър за релационна база данни. Протоколът CRA поддържа гъвкави и прозрачни взаимодействия между подсистемите (включително динамична промяна на ролите между клиенти и сървъри) с помощта на специфичен механизъм за асоциативно адресиране на съобщенията - content-oriented message addressing.

Друг клас хоризонтално-интегрирани системи, които имат много силно развитие през последните години са т.нар. полеви мрежи (fieldbuses) [26][154][160][180]. За разлика от системите с отдалечен вход/изход, интелигентните В/И устройства на полевите мрежи изпълняват редица допълнителни функции: предварителна (респ. постобработка) на въвежданите и извежданите сигнали, диагностика, калибриране и пр., като намаляват по този начин изчислителния товар на основния контролер. Това дава възможност за създаване на принципно нов тип системи за управление в реално време, напр. разпределени контури за регулиране. В една полева мрежа могат да бъдат вложени множество разпределени контури, както и дискретни управляващи структури - за логическо управление, за аварийна сигнализация и блокировка и др. В резултат на това се постига значително съкращаване на дължината на кабелните връзки, а така също -оперативност и взаимозаменяемост на функционалните блокове, произвеждани от различни фирми в рамките на един и същи fieldbus стандарт.

Разработени са редица национални и международни стандарти (напр. FIP във Франция и PROFIBUS в Германия [26][180]), ISA SP50 [72][95]. Тези fieldbus стандарти са ориентирани предимно към по-крупни приложения (инсталации) в химическата и нефтената промишленост, енергетиката и пр. В някои области, напр. автомобилната промишленост, при управлението на бързи машини и агрегати се използва друг вид управляващи мрежи - т.нар. Controller Area Networks (CAN), които имат по-висока реакционна способност в сравнение с fieldbus мрежите. Това се обяснява с някои специфични особености на системната архитектура (стандарт ISO 11898): приоритетен достъп до комуникационната среда, разпределено динамично диспечиране на достъпа, асоциативно адресиране на съобщенията и др. [188][199]. Обща особеност на двата класа системи обаче е функционирането в твърдо реално време, респ. поддържането на отдалечени и ограничени във времето взаимодействия между подсистемите на РСУ.

В момента според редица анализатори вече завършва “битката на стандартите”, т.е. разпределението на приложните области за различните видове полеви мрежи.

Освен изброените мрежи, които имат типично „събитийно-активирано” (event-driven) поведение, в последното десетилетие се разви и ново направление – активирано-по-време (time-triggered) мрежи. При тях всички дейности по магистралата са свързани с хода на времето чрез твърд график. И двата подхода имат своите застъпници противници, като “time-triggered” подходът е предпочитан при системите с изискване на гарантирано качество и отказоустойчивост (“safety-critical”).

Технологията на съвременните комуникационни системи е в процес на бурно развитие, характеризиращо се с:

- използване на различни видове комуникационни среди - кабелни връзки, оптични влакна, радиовръзки, инфрачервени лъчи и др;
- значително поевтиняване на комуникационните интерфейси;

- създаването на мрежови архитектури с разнообразни и взаимно-допълващи се функции.

В близко бъдеще това ще доведе до създаването на хетерогенни йерархични мрежи с глобален обхват на управляващите функции. Пример за подобна система е мрежовата архитектура Lonworks на фирмата Echelon Corp. [160]. В момента се работи и върху мрежи, които ползвайки захранващите линии като физическа преносна среда, осигуряват скорости на обмен до 1 Mbit/s [197], което позволява много по-силна връзка между отделните елементи от едно ниво на йерархия (физически отдалечени), а така и между различните нива.

Разгледаните видове разпределени системи за управление могат да бъдат окачествени като слабо-свързани мултикомпютърни системи, реализирани на базата на конвенционални мрежови архитектури със сериен обмен на данни (съобщения). В повечето случаи динамичните характеристики на този вид мрежи (пропускателна способност, латентност) са адекватни на изискванията на управлявания обект и на системата за управление.

В някои случаи, обаче, напр. при управление на сложни и/или бързи обекти, конвенционалните мрежи нямат нужното бързодействие. В такива случаи се използват нови системни и технически решения, в т.ч.:

- бързи комуникационни мрежи, изградени на базата на оптични технологии, напр. FDDI [169];
- специфични решения, характерни за силно-свързаните мултикомпютърни системи, в т.ч. паралелни комуникационни магистрали, директни и индиректни съединителни мрежи.

Известни са стандарти за паралелни системни шини, които поддържат изграждането не само на съсредоточени (едно- и многопроцесни системи с обща памет), но и на разпределени мултикомпютърни системи, напр. MULTIBUS II, FASTBUS и др. Специално внимание заслужава стандартът FUTUREBUS+, разработен с оглед изискванията на системите за реално време, в който се предвиждат средства за разпределено приоритетно управление на достъпа до магистралата, предсказуема комуникация между отдалечени изчислителни процеси, поддържане на общо астрономическо време чрез синхронизиране на часовниците на съставните компютри (възли) – стандарт IEEE 1588 и пр. .
Адекватното функциониране на РСУ с твърди времеви граници е възможно при интегрирано диспечиране на ресурсите на компютърната мрежа, в това число и на комуникационната среда. Това изисква подходяща организация на мрежата (и по-специално на достъпа до комуникационната среда), която е разгледана в заключителния раздел на обзора.

1.4 Модели за представяне на КСУ

В областта на формалните модели на системите една от най-общите концепции е “моделът на изчисления”. Тя се отнася към математическите модели , които определят семантиката на изчисленията и на конкурентността. Конкурентността е най-важният елемент на изчислителните модели. Възможността за съчетаване на различни изчислителни модели в една система поражда понятието за хетерогенни системи. Най-важната особеност и изискване при този тип системи е необходимостта от възможност за комуникация и взаимодействие между различни подсистеми, реализиращи различни изчислителни модели по унифициран, независим от модела на изчисления, начин. [121]

Предвид двете основни положения – един компютър изпълнява квази-паралелно някакво множество от програми или множество от свързани по някакъв начин компютри изпълняват множество от взаимодействащи си по някакъв начин програми, възниква въпроса с поддръжката на конкурентни изчисления. Разработката на свързани в мрежа системи за управление изисква спецификации и техники за моделиране, които поддържат конкурентност на изчисленията. Масово поддържано мнение е, че не

съществува универсален модел за представяне на конкурентни изчисления. Обратно, при последователните изчисления, моделът на фон Нойман е изключително успешна универсална абстракция [123]. Основен елемент на това е фактът, че ходът на времето е заменен от елементарна последователност от събития, при които сериализирането е достатъчно, за да се осигури коректността. В разпределените системи подобна поддръжка е практически нереализуема и може да се разчита най-много на частична подредба на събитията. Това прави ключовото при модела на фон Нойман понятие „състояние на системата" неопределимо в общ смисъл. Друг проблем е, че при различни пропускателни способности на съединителната мрежа са подходящи различни модели на изчисления. Освен това различните апаратни среди предполагат различни модели на абстракция на хардуера (напр. наличие или липса на кеш-памет, тип на комуникационната мрежа и др.). Приложими са различни методи на абстракция [29][60][69][107][104] и др.

Независимо дали този факт се отчита или не, всяка система е построена на базата на някакъв модел на изчисления. Проблемът се състои в това, че така се определят приложимите алгоритми и много често избраните модели водят по необходимост до използуване на неадекватни алгоритми.

Масово решение в момента е поддръжката на различните модели за изчисления да се реализира с нови езици за програмиране (текстови и графични) или разширения към съществуващи езици. Основен проблем за тези езици е, че те по принцип са слабо разпространени, имат сериозни платформени ограничения, поддръжка и слаба приемственост между версиите. Друг подход е използуването на моделите на изчисление за *координация* чрез програмни модули, написани на обикновени (стандартни) езици за програмиране. Това разделяне на езика за програмиране от реализирания програмен модел е едно от най-перспективните решения на изложените по-горе проблеми. Така и смесването на модели на изчисление между отделни модули и подсистеми става много по-просто и естествено. Позволява се използуването на модели, най-подходящи за отделните проблемни области, без да има съществени влияния между локалните решения в отделните подсистеми. [121]

Известни са разнообразни изчислителни модели .

Според [147-126] по-основни от тях са:

- **Диференциални уравнения**

Това е един от най-разпространените математически модели, които се използуват в областта на теорията на управлението. Поради това няма да им се отделя допълнително внимание. Единственото, което трябва да се отбележи е, че този модел е подходящ за моделиране на непрекъснати процеси. Основен проблем е, че при реализация с числов компютър се получава несъответствие между начина на работа на устройството "компютър" (дискретно устройство) и модела, който се представя (непрекъснат).

Този модел се използува например в Simulink, Saber, VHDL-AMS и др.

- **Диференчни уравнения**

Диференчните уравнения представляват дискретизирана форма на диференциалните уравнения. Като такава те нямат проблема за реализация на непрекъснат модел с дискретно устройство. При тях също има ограничения в употребата – не са подходящи за реализация на управляваща логика и реакции на случайни събития.

- **Крайни автомати**

Това е другият най-използван модел. Използува се за представяне на дискретно-събитийни системи. Базовият модел е напълно неконкурентен. Изпълнението е представено от серия от подредени преходи от едно в друго състояние. Една по-обобщена форма на това са системи, при които едно състояние на автомата може да представя на повече от едно състояния на системата, а и броя на състоянията не е ограничен по принцип.

Основните проблеми на този модел са:

- Не е подходящ за описание на непрекъснати процеси.
- При големи системи броят на състоянията нараства много бързо (експлозия на състоянията), което е сериозно затруднение при компютърното представяне.

Решаването на проблема с експлозията на състоянията се решава, като се декомпозират автоматите на субавтомати и прилагане на паралелно (конкурентно) описание и изпълнение на тези субавтомати. Harel въвежда този подход, като предлага формализма на STATECHARTS (диаграми на състоянията) [109]. Това е един от базовите модели, залегнали в спецификационните езици от последно поколение и главно UML.

- **Синхронно-реактивен модел**

При този модел на изчисления се определя връзката между входните и изходните сигнали на всеки тактов сигнал. [29]. Разликата от диференчните уравнения е, че не се изисква да се определят нови стойности на входните сигнали на всеки такт. Примери на езици, които реализират този модел са ESTEREL [33], SIGNAL[31], LUSTRE [61] и ARGOS[143] (виж по-долу в текста). Този модел е много подходящ за конкурентни сложни системи. Поради много силната синхронизация между събитие-реакция много често се получава свръхопределяне на конструираната система, като затруднява конструирането, а често и разделянето на системата на подсистеми и модули.

- **Дискретно-събитийни системи**

При тези модели събитията се представят като стойност и времева марка към него. Този модел е особено разпространен в средите за конструиране на хардуер и комуникационни системи, напр. VHDL, Verilog и SDL. При този модел няма общ системен тактов сигнал, но отчитането на хода на времето се поддържа консистентно за цялата система.

- **Синхронни модели**

При редица системи активността на система е пряко обвързана с появата на периодично събитие – тактов сигнал. Това се отнася еднакво за системи за управление на непрекъснати и дискретни, по своя начин на работа, обекти. Това е голяма разлика от програмируемите автомати, които са по принцип асинхронни (въпреки, че въвеждането на синхронизъм при тях е отдавна известно и често срещано) и напр. диференчните уравнения, които са тясно обвързани с хода на времето, но не включват експлицитно привързване към явен тактов сигнал. Най-известен представител на езиците, реализиращи този тип модели е SCENIC [135]. Той е развит на базата на спецификацията SystemC за базов дизайн на хардуер [178].

- **Синхронна комуникация**

При синхронната комуникация компонентите са реализирани като процеси, които си комуникират чрез атомарно “моментално” действие, наречено “*рандеву*”. Същината на този процес се състои в това, че при необходимост от комуникация, процесът който достига пръв точката на комуникация остава блокиран да изчака втория процес да достигне и той точката на взаимодействие. Обменът се извършва като неделима (атомарна) операция. Най-известни примери за формални модели реализирани на базата на рандеву са “Communicating sequential processes” на C.A.R. Hoare [90] и “Calculus of communicating systems” на Milner. Той е реализиран в редица езици за реално време, включително Occam и Lotos. Този модел е ориентиран главно към системи, при които главен проблем е ползването на общи ресурси, например “клиент/сървър” взаимодействия или съвместно ползване на някакъв хардуер. Главен проблем тук е пълната недетерминираност на реализираните системи.

- **Асинхронна комуникация**

При асинхронната комуникация, процесите си комуникират изпращайки съобщения през канали, които могат да буферират съобщенията. Не се налага подателят на съобщението да изчаква приемникът да бъде готов за

получаването му. Вариантите на тези модели, които гарантират детерминираност на поведението на системите са дефинираните от Kahn процесни мрежи [104] и моделът “поток на данните” (ПД, data-flow).

Процесните мрежи са много подходящи за реализиране на разпределени и паралелни системи. Основен недостатък при тях е практическата невъзможност да се реализира управляваща логика. [1]

“Синхронен поток на данните” (synchronous data flow) е специфичен вариант на процесна мрежа, при който се получава възможността за елиминиране на взаимните блокировки и неограничената работа [67][121]. Има варианти и на Булеви ПД и динамични ПД, които предлагат определени предимства при някои анализи, но и те имат редица ограничения.

- **Производител/потребител**

При този модел същината за представянето на взаимодействията е в това, че има два типа компоненти – потребители, които регистрират “интерес” към определени събития и производители, които пораждат тези събития. Повечето от известните системи, реализиращи този модел, са на базата на Linda [7][60][189] Този модел е подходящ за слабо свързани разпределени системи с асинхронна комуникация между компонентите. Тук производителят не знае и не се интересува кои са потребителите на събитията (данните), които генерира.

Както се вижда, при цитираните анализи [77][82][121-126][165] има голямо смесване на чисто математически модели, описващи начини на функциониране на “управлявани” обекти с модели, описващи функциониране на елементи от компютърната система. Много от моделите са ориентирани към различни комуникационни модели, които определят взаимодействието между отделните компоненти и оттам общото поведение на системата. Тези модели са основа на т.нар. “комуникационно-базиран” подход на дизайн на системи. Тук комуникацията е относително независима от модулите (като формален модел и поведение), които участвуват в нея. Това е и основния начин на смесване на модели – за комуникация и за изграждане на модулите.

Една от основните разлики на тези спецификационни модели е включването на времето като явен елемент на спецификацията. При някои модели представянето е елементарно - те разглеждат времето като число, което се увеличава равномерно и непрекъснато. Събитията се разполагат върху времевата ос или се генерират сигнали, които се развиват по времевата ос. Други модели са по-абстрактни, разглеждайки времето като дискретно или още по-абстрактно - като причинно-следствена връзка. В този случай се разглежда времето като частично подредено. Това позволява формален анализ и сравнение. Процесните мрежи и моделите, базирани на рандеву, са базирани на това, което е обяснение на тяхната изразителност. Мнозина изследователи работят по въпроса за ролята на времето в изчисленията. Albert Benveniste твърди, че в определени класове системи "природата на времето е универсална, но по-скоро местна за всяка подсистема и следователно разнообразна" [29]. Leslie Lamport твърди, че няма начин точно да се поддържа координирана представяне на време в разпределени системи и показва, че частичната подредба е достатъчна [120].

При синхронните системи комуникация може да се реализира само в определени моменти от времето. Това е много важен фактор за намаляване на непредсказуемостта на поведението им. Това обаче, както беше вече отбелязано, води до свръхопределеност много пъти, което затруднява синтеза и води освен това до понижена производителност на получената система. Това поражда необходимостта от асинхронна комуникация. При нея обаче възможността за поява на непредсказуемост в поведението е много по-голяма, което изисква специален анализ. Така се стига много пъти до хибридна схема на комуникация – глобално асинхронна, локално синхронна (GALS - globally asynchronous locally synchronous) [25].

Във връзка с това подходът за решаване на задачите за отразяване на непрекъснати и дискретни събития е разработен в рамките на “синхронните системи и езици” като SIGNAL, LUSTRE, ESTEREL и др. [28][29][31] Този подход се основава на моделиране поведението на КСУ в логическото, а не във физическото време и се основава на няколко основни принципа:

- КСУ се разглежда като реактивна дискретно-събитийна система, която възприема външните събития и реагира мигновено на тях в съответните моменти от времето (**синхронност на реакциите**);
- Реакциите на КСУ са **атомарни**, т.е. неделими. Това означава, че системата не може да реагира едновременно (конкурентно) на повече от едно събитие. Горното на свой ред следва от хипотезита за синхронност, при което реакциите на системата съвпадат върху времевата ос с моментите на възникване на съответните събития;
- **Мигновена комуникация** между подсистемите на управляващата система (instantaneous broadcast). Това изискване следва логически от хипотезата за синхронност, тогава когато системата за управление представлява съвокупност (композиция) от взаимодействуващи подсистеми;
- **Детерминизъм** в поведението, т.е при една и съща последователност от събития системата винаги трябва да генерира една и съща последователност от реакции.

Характерът на тези реакции зависи от типа на реализираната система и заложения в езика операционен модел, напр, автоматен модел (ESTEREL), или по-общ изчислителен модел, описван чрез система от рекурентни уравнения (SIGNAL, LUSTRE), който може да бъде използуван както за описание на управляващи автомати, така и на аналогови устройства - филтри, регулатори и пр.

1.4.1 Системи за управление на непрекъснати обекти

Съвременната теория на управлението се развива изключително интензивно през последните десетилетия, стимулирана, от една страна, от реалните нужди на промишлеността (и преди всичко на военното дело и авио-космическите приложения) и от друга страна - от интензивното развитие на изчислителната техника, използувана като средство за изследване и реализация на сложни системи за управление. На фона на това развитие изглежда парадоксална констатацията за сериозен разрив между теорията и практиката на системите за уравление. Такъв разрив обаче действително съществува, що се отнася до промишлените системи за управление на непрекъснати обекти. По-специално, констатира се, че в 95% от инсталираните промишлени системи се използуват "класически" ПИД-регулатори, като половината от тях се експлоатират в режим на ръчно управление. От друга страна, алгоритмичното осигуряване на редица най-съвременни фирмени системи не отива по-далеч от споменатия ПИД -регулатор и някои негови разновидности [32][150][157][191]. Тази ситуация е била многократно обсъждана в средите на специалистите по автоматика и причините (повечето от които с рационално обяснение) са добре известни, напр. адекватността на PID-регулатора за една голяма част от промишлените приложения.

Едновременно с това са налице признаци за постепенна промяна на така описаната ситуация, предвид появата на достатъчно мощни и същевременно, евтини промишлени контролери. По-конкретно, очертават се четири направления на развитие и приложение на съвременната теория на управлението за промишлени цели [21]:

- Контролери, използуващи параметрични количествени модели на обекта, респ. системата за управление, в това число: регулатори с обратна връзка по изхода / състоянието, някои видове адаптивни и

самонастройващи се регултори, напр. системите IDCOM, Horizon Predictive Control и пр.

- Контролери, използуващи непараметрични количествени модели от рода на невронните мрежи.
- Контролери, използуващи полуколичествени модели; към тази категория се отнасят размитите модели и съответно, регулатори, реализирани като специализирани (вградени) системи, а така също - като програмни пакети за PC-съвместими контролери от рода на Fuzzy-Tech, Fuzzy Control Manager и др.
- Контролери, използуващи качествени модели, т.е. емпирично-натрупано експертно знание, а от там и названието "експертни управляващи системи", респ. "системи, базирани на знания", напр. системите KATE, G2 и др.

До скоро всяко едно от посочените направления се развиваше самостоятелно (в аспект на теория и практическа реализация), като по-широко приложение са получили контролерите, реализирани на базата на съвременната (количествена) теория на управлението, а в последно време и размитите регулатори. Приложението на невронните мрежи и експертните системи за целите на прякото цифрово управление е по-ограничено, като експертните системи се използуват предимно в по-високите нива на КСУ. Следва да се отбележат и някои интеграционни тенденции, напр. комбинираното използуване на невронни мрежи и размити регулатори, а така също - създаването на интегрирана архитектура на системи за управление на непрекъснати обекти на базата на споменатите по-горе подходи. Предполага се, че подобна архитектура ще бъде характерна за следващото поколение разпределени системи за управление.

1.4.2 Системи за управление на дискретни обекти

И при този тип системи за управление се констатира значително разминаване между теорията и практиката. Показателно за това е масовото приложение на една остаряла методология за синтез на дискретни управляващи устройства, респ. управляващи програми - т.нар. релейни диаграми, както и емпирично-интуитивният подход при съставянето на релейни диаграми [156][191]. От друга страна, съществува мощна теория за абстрактен и структурен синтез на дискретни управляващи устройства (ДУУ), зададени с последователностни (автоматни) модели или с мрежови модели, напр. мрежи на Петри със съответната входно-изходна интерпретация. На основата на тези модели са разработени редица символни и графични езици, които обаче не са получили широко разпространение, с изключение на GRAFCET, представляващ модифициран вариант на интерпретирана мрежа на Петри [191] и VHDL, предназначен първоначално за синтез на електронни схеми.[201]

Формалните модели, получени на етапа на абстрактния синтез на ДУУ, могат да бъдат използувани за синтез на съответния управляващ алгоритъм (програма). Това става по различен начин в зависимост от конкретния вид на дискретния контролер. При т. нар. **програмируеми логически контролери** (programmable logic controllers), абстрактният модел на ДУУ се трансформира в структурен модел с помощта на някой от известните методи за структурен синтез. Получената в резултат на синтеза система от булеви функции описва адекватно (макар и индиректно) поведението на управляващото устройство. В този случай изпълнението на управляващия алгоритъм се свежда до периодично преизчисляване на системата от булеви функции, като по този начин се емулира структурата на ДУУ.

При т.нар. **програмируеми последователностни контролери** (state-logic controllers) етапът на структурния синтез се елиминира. В този случай абстрактният модел на ДУУ представлява изпълнима спецификация на

управляващия алгоритъм. В някои контролери тази спецификация се транслира директно до изпълним код, а в други се интерпретира от специализирана служебна програма. В този смисъл може да се счита, че програмируемият последователностен контролер емулира поведението на ДУУ.

Използването на програмируеми логически контролери има съществени недостатъци и по-специално:

- необходимост от структурен синтез на ДУУ, което е нетривиална задача при автомати с голям брой състояния / входове / изходи;
- преразход на машинно време поради необходимостта от изчисляване на пълната система от булеви функции във всеки такт (цикъл) на контролера, като не се отчита ефектът на локалност в поведението на ДУУ.

Посочените недостатъци водят до преминаване към последователностни контролери, които са освободени от тях [12][42][78][156].

Известни са и контролери от смесен тип, напр. SIMATIC S5 и S7 на Siemens, в които графсет-диаграмата (наречена GRAF5-диаграма) се използва като неизпълнима спецификация , чиито елементи (оператори и преходи) се реализират с помощта на изпълним инструментален език - асемблерен език STEP 5, релейни диаграми или логически графични диаграми. Сред проектантите и потребителите на подобни системи съществува мнението, че наличието на няколко взаимно-допълващи се езика облекчава използването на системата и я прави по-гъвкава и, в известен смисъл, по-отворена. Тази идея е заложена в стандарта IEC 61131 за "отворени" програмируеми контролери [187], който включва пет алтернативни езика, в т.ч.:

- Sequential Function Chart - графичен език подобен на Графсет;
- Instruction List - разширен процесорно-независим асемблерен език;
- Structured Text -паскалоподобен език за структурно програмиране;
- Ladder Diagram - релейни диаграми;
- Function Block Diagram - графичен език за описание на структурни схеми.

Споменатият стандарт е използуван при реализацията на редица "отворени" системи, реализирани с PC-съвместими контролери, напр. Digimatic, ISaGRAF WDS и др.[78].

1.4.3 Хибридни системи за управление

Характерното за този тип системи е комбинацията на дискретно и непрекъснато поведение:

- в системата за управление, която в този случай може да се разглежда като комбинация от автомат и регулатор(и), като например в някои видове адаптивни системи;
- в управлявания обект - напр. при т.нар. batch (циклични) технологични процеси.

Комбинацията на двата типа поведение (респ. на двата типа алгоритми) може да се осъществи тривиално чрез съчетаване на два типа контролери (със съответните алгоритми) или на два типа алгоритми в рамките на един многофункционален контролер. Последната тенденция е характерна за редица съвременни системи. Така например, в програмното осигуряване на редица аналогови контролери се включва не само традиционната библиотека от алгоритми за регулиране (аналогови функционални модули), но и средства за реализация на логически и последователностни алгоритми. Обратно, в повечето програмируеми логически контролери, напоследък, се предвижда възможността за изпълнение на управляващи алгоритми от рода на PID-регулатор или негови модификации.

Следва да се отбележи обаче, че формулираният по-горе проблем засега няма напълно задоволително решение. Както се посочва в [30] "... все още

липсва подходяща парадигма за хибридните системи. В резултат на това аналоговият и дискретният свят остават разделени както по отношение на методологията за изследване, така и по отношение на начина, по който се разработва алгоритмичното и програмното им осигуряване (различните теоретични рамки обикновенно водят до различни и несъвместими програмни средства)".

Известни са не много опити за преодоляване на посочената несъвместимост. Така например, в [29] е показано, че както аналоговите, така и последователностните структури могат са бъдат представени чрез два дуални способа за описание:

а) система от рекурентни уравнения;

б) диаграма на състоянията.

Първият способ е много подходящ за аналогови блокове от типа на филтри, регулатори, и др., но може да бъде използуван с успех и при структурното описание на последователностни устройства; вторият способ е приложим както при последователностни устройства, така и (в по-малка степен) при описанието на непрекъснати динамични системи. Посоченият дуализъм има характер на основополагащ принцип, използуван при реализацията на т.нар. синхронни езици за програмиране (напр. SIGNAL, LUSTRE, ESTEREL), една от целите на които е адекватното представяне на хибридни структури и алгоритми [28][29]. Все пак, в [29] се признава, че езиците, базирани на диаграми на състоянията (като ESTEREL) са по-подходящи за описание на последователностни системи, докато езиците, използуващи рекурентни уравнения (SIGNAL, LUSTRE) са по-подходящи за описание на аналогови структури.

Друг подход за унифицирано описание на управляващи алгоритми е използуван в системите PROCOS [11][12][13] и AT600. В тези системи алгоритмите се специфицират чрез изчислителни графи, които се интерпретират по различен начин, в зависимост от вида на управлението:

- При реализацията на непрекъснати управляващи алгоритми спецификацията се интерпретира като поредица от изчисления (път в графа на изчислението), която се формира динамично във всеки такт на управление. По този начин се реализира входно-изходният оператор на системата за управление (регулатор);
- При реализацията на дискретни управляващи алгоритми спецификацията се интерпретира като поредица от изчисления и/или дискретни стъпки, активирани и съответно - изпълнявани в отделни тактове или тактови поредици на автоматната хронограма. По този начин се реализира синхронно-асинхронен управляващ автомат или набор от автомати;
- При хибридни системи се използува смесена интерпретация на изчислителните графи (или части от тях), с което се осигурява необходимата комбинация от дискретно и непрекъснато поведение.

Този подход се отличава със значителна гъвкавост и опростена реализация. Неудобството при него е, че спецификацията на система включва и описание на управляващата логика, и описание на потока на данните, което предизвиква затруднения при специалистите по автоматика.

1.5 Програмно осигуряване на КСУ

Качеството и цената на програмното осигуряване са въпроси от първостепенно значение при изграждането на КСУ. Структурата на програмното осигуряване може да бъде описана по следния начин [30]:

- Програмно осигуряване (ПО) = приложно ПО + базово ПО.
- Приложно ПО = моделиране + алгоритми + евристики.

- Базово ПО = математически и др. библиотеки + операционни системи и компилатори + комуникационен софтуер + софтуер за интерфейс с потребителя.

Горните зависимости показват, че синтезът на алгоритми за управление е само един от аспектите на проектирането на ПО. В съвременните управляващи системи значителна роля играят евристиките (правила, действия, обработки - специфични за всеки конкретен случай), които се реализират като целеви програми. Според доклад от 2003 г. в една система за управление от реализирания софтуер за управление над 75-80 % се пада на различни форми на базирани на знания и други "неформални" методи за управление. Аналогични елементи могат да бъдат заложени и в първата компонента (моделирането) под формата на бази знания. Друга характерна тенденция е нарастващата роля на софтуера за връзка с оператора - във функционално, количествено и стойностно изражение. Що се отнася до базовото ПО, на съвременния етап то се характеризира с висока степен на сложност и значителни разходи за разработка, които обаче се компенсират при многократното му използване. Друга тенденция е използването на стандартизирани операционни среди и системи за развитие.

Пак в [30] се прави характеристика и на ценовите аспекти на ПО:

- Цена на ПО > цена на апаратните средства:
  - 50% - 50% - във военни и телекомуникационни системи,
  - 90% - 10% - в промишлени системи;

  Приложно ПО > Базово ПО:

  Съотношението варира от 60% - 40% до 80% - 20%.
- Евристиката преобладава силно над математиката.

Според други източници днес положението е още по-крайно.

Въпреки, че математическата част може да бъде от първостепенно значение, евристичната част често доминира с един-два порядъка над математическата част, от гледна точка на цена и разход на труд [21][30].

Приведените количествени характеристики се обясняват с някои специфични особености на програмното осигуряване за управляващи системи:

- висока степен на сложност;
- функциониране в (твърдо) реално време;
- високи изисквания по отношение на надеждност и експлоатационна готовност;
- значителен обем.

По силата на посочените фактори разработката на ПО е първостепенен проблем (primary concern) в промишлените отрасли, ангажирани с проектиране и внедряване на КСУ [30]. Оттук става очевидна несъстоятелността на широко-разпространения в практиката целеви (ad-hoc) подход за проектиране на програмни системи и в частност - на системи за управление.

Следователно, необходима е качествено-нова методология за проектиране на програмни системи (особено на сложни системи). Този подход се състои в използването на формални модели - конфигурационни спецификации, описващи адекватно компонентите на системата, взаимодействията помежду им и с околната среда.

Освен разгледаните теоретични модели и приложението им в описанието на различни типове системи, може да се направи разглеждане според изпозвания програмен подход при изграждането им. Известно е, че в момента проектирането е на базата на два основни подхода – процесно-ориентиран и обектно-ориентиран.

Процесно-ориентираният подход се използва интензивно в теорията на конкурентните и реално-времевите системи [49][83][133]. В този случай програмните системи се изграждат като съвокупност от конкурентни изчислителни процеси, които си взаимодействуват чрез (и при обработката

на) потока от данни, който преминава през системата. Тези взаимодействия се свеждат до два базови типа: синхронизация (респ. сигнализация, взаимно изключване) и комуникация - както силно-свързани взаимодействия чрез общи структури с данни, така и слабо-свързани взаимодействия чрез обмен на съобщения, вкл. синхронни и асинхронни варианти на взаимодействия от вида "производител-потребител" и "клиент-сървър".

Процесно-ориентираният подход може да се разглежда, в известен смисъл, като подход, който отразява адекватно функциите на КСУ, но не всякога - нейната действителна структура. Посоченият недостатък може да бъде отстранен при използване на друг подход за структуриране на програмната система, който е характерен за обектно-ориентираните описания. В този случай системата се изгражда като съвокупност от взаимодействуващи софтуерни обекти, съответствуващи на функционални подсистеми на КСУ или на части от тях. При това, всеки обект капсулира данни и функции (методи) за обработката им, а взаимодействието помежду им се осъществява чрез обмен на съобщения [38][64][98][104].

На настоящия етап в системите за управление в реално време по-често (и по традиция) се използват процесно-ориентираните модели. Това се отнася в най-висока степен за системите с твърди времеви граници. Налице е обаче и тенденция за все по-широкото използване на обектно-ориентирания подход, предимно в некритични по време системи за управление на промишлени обекти. Това се обяснява както с предимствата на обектно-ориентираното програмиране, така и с някои негови недостатъци (преразход на време и памет, непредсказуемост във времето и пр.). По-нататъшното развитие на този подход може да доведе в перспектива до създаването на принципно-нова технология за изграждане на програмни системи въз основа на готови програмни компоненти (софтуерни интегрални схеми) и обектно-ориентирани спецификации [64]. Следва да се отбележи, че тази технология се използва широко и сега при синтеза на някои видове приложни програми, напр. управляващи програми, в съчетание с принципа на програмната генерация.

1.5.1 Процесно-ориентиран подход

Началните десетилетия от развитието на това направление са свързани преди всичко със създаването на методология за изграждане на програмни системи, както и на съответната софтуерна технология - езици, операционни системи и програмни среди. Известни са два способа за изграждане на процесно-ориентирани системи:

- на базата на операционна система за реално време ;
- на базата на език за програмиране в реално време (език за системно програмиране) [49].

В първия случай диспечирането на съставните изчислителни процеси и взаимодействието между тях се осъществява чрез операционна система за реално време (ОСРВ) или ядро (kernel) на ОСРВ; във втория случай това става с помощта на системно ядро, което се поражда от компилатора в процеса на компилиране и свързване на съставните процеси. [80]

Основен недостатък на разглежданата методология до преди около 25 години е била липсата на методи и средства за създаване на системи за реално време с гарантирано и предсказуемо поведение, а това е едно фундаментално изискване при системите за управление. Споменатото изискване обикновено се удовлетворява чрез интензивно тестване на отделните компоненти, процеси, подсистеми и на системите като цяло, с използване на модели на обектите (аналогови, компютърни, натурални) и накрая - изпробване на системата за управление в реални условия.

Подобна ad-hoc методология за валидация на системите за управление е свързана със значителни разходи на труд, време и средства и не гарантира безопасността (отказоустойчивостта) на системите. Ето защо, през

последните 15 години се отделя изключително внимание на създаването на нова методология за изграждане на програмни системи за реално време, осигуряваща гарантирано и предсказуемо поведение в условията на твърди времеви граници и повишени изисквания за надеждност и безопасност.

Така формулираните изисквания определят в най-общ вид функционалните характеристики на системата за управление. В конкретизиран вид те се изразяват чрез съвокупност от свойства на реализираната програмна система, които могат да бъдат отнесени към два основни класа:

- свойства на живост (liveness properties)
- свойства на безопасност (safety properties).

Първият вид свойства се формулират като твърдения, свързани с достижимостта на определени ситуации (състояния) в процеса на функциониране на КСУ, т.е., ако системата се намира в някакво текущо състояние, тя евентуално ще премине в друго състояние, регламентирано от съответния експлоатационен режим.

Вторият вид свойства се формулират като твърдения, с които се изключва възможността за преминаване в нерегламентирани (респ. аварийни или предаварийни) състояния и ситуации, застрашаващи безопасността на КСУ. Казано по друг начин, свойствата за безопасност формулират условията, които трябва да са абсолютно валидни, за да се изключи възможността от настъпване на такива ситуации.

Известни са различни способи за формален запис и верификация на подобни свойства, напр. т.нар. темпорална логика, развита за целите на конкурентните (но не реално-времеви) системи [196]. Те обаче не са директно приложими при системите за реално време поради:

- висока степен на сложност на системите и взаимодействията;
- въвеждане на времева метрика при изразяване на свойствата на системата (напр. твърдението "При подаване на авариен сигнал КСУ включва аварийната блокировка не по-късно от 0.1 секунда след възприемането му" вместо по-общото твърдение "...евентуално ще включи блокировката", присъщо на системите, които не работят в "твърдо" реално време).

През последните години интензивно се развиват методи за формална спецификация и верификация на реално-времеви системи, в които горните затруднения се преодоляват чрез използване на комбинирани описания, включващи:

- операционни спецификации, описващи функционирането, т.е. състоянията и преходите на системата за реално време и нейните компоненти, напр. обобщени автомати, взаимодействуващи автомати [170], различни видове автоматни диаграми от вида на STATECHARTS [103], MODECHARTS [175], RSML [105][132], времеви мрежи на Петри [75] и т.н.;
- логически спецификации на изискванията за живост и безопасност (requirements specifications), формулирани по отношение на състоянията и преходите на операционната спецификация, с използване на реално-времева логика, напр. RTTL - темпорална логика с времева метрика.

При такава комбинирана спецификация, верификацията на програмната система се свежда до доказване на твърденията, формулирани по отношение на операционния модел на системата (model checking).

Независимо от интензивните изследвания в областта на формалната спецификация и верификация на реално-времевите системи, това направление все още се развива [49]. В случая критерий за зрелост е възможността за спецификация и верификация на сложни реални системи, а не само на илюстративни примери. В това отношение най-голям успех представляват системите STATECHARTS и RSML, които са използвани успешно за спецификация (предимно операционна) на големи авиационни системи.

Тези предположения важат при следните посочени и по-горе принципи:

- Реакциите на КСУ са атомарни, т.е. неделими;
- Мигновена комуникация между подсистемите на управляващата система (instantaneous broadcast);
- Детерминизъм в поведението.

Формулираните принципи са валидни при условие, че бързодействието на използувания компютър е много (на порядъци) по-високо от това на обекта. Горното предположение е твърде силно, но при съвременното състояние на изчислителната техника то в много случаи е адекватно. Това дава възможност да се игнорират специфичните проблеми, свързани с отчитане на реалното време, и съответно - да се опрости задачата на верификацията, която се формулира в домена на логическото време. Последното се разглежда като поредица от моменти на възприемане/реагиране на събития, породени в околната среда и по-специално - в управлявания обект. При това, разработените синхронни езици имат двойно предназначение - за спецификация и реализация на управляващи програми; програмите представляват по същество изпълними спецификации, чиито свойства се верифицират на етапа на компилацията.

Този подход има няколко проблематични момента:

- Хипотезата за синхронност и атомарност на реакциите не е валидна при управление на бързи и сложни обекти, изискващи едновременното формиране на множество реакции с твърди времеви граници;
- Предположението за мигновена комуникация между подсистемите е очевидно невалидно при разпределени системи за управление.

Друго направление, което се развива особено успешно през последните десет години в САЩ, Англия и др. страни, се базира на съвременната теория на системите с фиксирано приоритетно диспечиране (Fixed-Priority Scheduling Theory). В САЩ се развива вариант на теорията, известен като Rate-Monotonic Scheduling Theory (в най-близък превод: интервално-монотонна теория на диспечирането), който е свързан с основополагащата работа [49]. В Англия се развива вариантът, известен като Deadline-Monotonic Scheduling Theory. В последните години обаче се утвърждава становището за обща теория на системите с фиксирани приоритети, предвид на единната методология, характерна и за двата й варианта [21][47][48][65][111][131][165][166][169].

Основните причини за налагането на това направление са:

- Еволюционно развитие на теорията, което се основава, до голяма степен, на известните (към началото на 80-те години) методи за проектиране на конкурентни и реалновремеви системи;
- Разграничаване на задачите (separation of concerns) по отношение на верификацията на системите за реално време, които се обособяват в две отделни категории:
    - логическа, респ. функционална коректност;
    - времева коректност на проектираната система.

Първият тип задачи се решават със стандартни методи и средства, намерили широко приложение при верификацията на системи, които не функционират в реално време (напр. тестване на програми, методи използуващи традиционна формална логика и пр.). Вторият тип задачи се решават с помощта на аналитични зависимости за оценяване диспечируемостта на процесите (задачите), от които е съставена програмната система, а от там и на системата като цяло.

По дефиниция, системата за реално време е диспечируема, ако съставящите я изчислителни процеси се изпълняват, като се спазват времевите им граници. Спазването на времевите граници е основният критерий, по който се оценява времевата коректност на изчислителните процеси. Следователно, диспечируемостта на системата, като цяло, гарантира нейната работоспособност.

Извеждането на споменатите зависимости, обаче, е възможно при две условия:

- Детерминирана функционална рамка, в която се поставят процесите с твърди времеви граници - както периодични, така и апериодични, като последните се третират като псевдопериодични (спорадични) процеси;
- Детерминирани механизми за функциониране на ОСРВ (респ. на езиковото ядро) и на програмната система като цяло, в т.ч.:
  - диспечиране на периодични процеси с твърди времеви граници - при нормални условия и при претоварване [18][19][101][127][130][136];
  - обслужване на апериодични заявки чрез статично резервиране на изчислителен ресурс в рамките на периодични или псевдопериодични задачи - апериодични сървъри [129];
  - обслужване на некритични по време апериодични заявки чрез динамично резервиране на свободния (неизползувания) изчислителен ресурс [20][68][67][128];
  - предсказуеми механизми за локални междупроцесни взаимодействия - т. нар. синхронизационни протоколи, елиминиращи противопоказни за работата на системата ефекти като взаимно блокиране (deadlock), приоритетна инверсия, верижно и транзитивно блокиране на процеси [23][24][161][162][167][168];
  - предсказуеми механизми за отдалечени междупроцесни взаимодействия в мрежи;
  - предсказуеми механизми за смяна на изчислителния товар при превключване на експлоатационните режими - mode change protocols [180].

Така характеризираната теория се основава на динамично (on-line) диспечиране на реално-времевите процеси, в противовес на статичното циклично диспечиране, което се използва все още при реализацията на някои специални видове системи [138]. В последния случай диспечирането на процесите се осъществява чрез статични (off-line) разписания, които представляват периодично-повтаряни последователности от времеви интервали, статично закрепени за съответните процеси (cyclic scheduling). На практика, това е механизъм за твърдо мултиплексиране на процесите, който често се осъществява с ad-hoc методи. Същият е свързан със значителен разход на време, труд и средства, поради преплитането на времевите и алгоритмичните аспекти на проектираната система. Обратно, при използване на Теорията на системите с фиксирани приоритети, тези два аспекта се третират отделно, което улеснява значително проектирането.

Тази теория поставя научно начало при проектирането на системите за реално време с твърди времеви ограничения , като елиминира използваните до момента ad-hoc методи. За това може да се съди от сравнително бързото Ł налагане в инженерната практика и по-специално:

- в проекти и стандарти за ОСРВ, напр. Real-time Mach [186] и POSIX 4 [111];
- във версии на езици за програмиране, напр ADA 95 [22][50], RT C++ [96] и др.;
- в крупни инженерни проекти в областта на авиацията, космонавтиката и военното дело [111].

Показателен е фактът, че Теорията на системите с фиксирани приоритети е възприета като основна работна теория на NASA и ESA при проектирането на системи за управление на сложни космически обекти, напр. бъдещата международна орбитална станция [111][166].

Процесно-ориентираните спецификации описват програмната система като съвокупност от взаимодействуващи изчислителни процеси, функциониращи в средата на разпределена изчислителна система, разглеждана най-често като виртуален единичен компютър [53][82]. Процесите (и нишките) са най-широко използвания елемент на системите, реализиращи разпределени и конкурентни изчисления. Това е така, защото процесите

могат да бъдат разгледани като технология за изграждане на компоненти, при което многозадачна операционна система осигурява синхронизацията и координацията между компонентите – процеси или нишки. [177]

Процесно-ориентираната спецификация на РСУ може да се дефинира йерархично, при което програмната система се разглежда като съвокупност от статично-разположени подсистеми (виртуални възли), а всеки такъв възел представлява подмножество от силно-свързани (и разположени в същия физически възел) процеси, съответствуващи на обособена функционална подсистема на РСУ [53].

В рамките на така дефинираната спецификация могат да бъдат обособени два вида процесни взаимодействия: локални и междувъзлови. В локалните взаимодействия участвуват силно-свързани процеси, принадлежащи на един и същи виртуален (и физически) възел. Най-често за тази цел се използват механизми за комуникация чрез общи информационни структури, формиращи оперативната база с данни на възела.

Междувъзловите взаимодействия се осъществяват посредством двойки процеси, всеки един от които принадлежи на съответния възел-кореспондент. Те се осъществяват чрез локален или отдалечен обмен на съобщения в зависимост от разположението на взаимодействуващите виртуални възли - в един и същи или в различни физически възли на мрежата.

Така описаният модел е използуван при реализацията на редица разпределени ОСРВ и езици за програмиране на разпределени системи, напр. езика SR [53]. Известни са и модели, които се отличават от разгледания, с използуването на един и същи механизъм - обмен на съобщения - както за локални, така и за междувъзлови взаимодействия, напр. системата CONIC . При това следва да се отбележи, че локалните взаимодействия се тълкуват еднозначно, независимо дали става дума за виртуални или физически възли, докато взаимодействието между виртуалните възли може да бъде както локално, така и отдалечено (във физическия смисъл на понятието). Във връзка с това трябва да се отбележи едно основно изискване към програмната система на РСУ (респ. към разпределената ОСРВ, чрез която тя се реализира), а именно - прозрачност на взаимодействията, които трябва да се осъществяват логически по един и същи начин, респ. чрез едни и същи механизми на ОСРВ, независимо от това дали взаимодействуващите процеси са разположени в един и същи или в различни физически възли. В последния случай механизмите на ОСРВ се опосредствуват от съответния комуникационен протокол, но това обстоятелство остава скрито (прозрачно) за приложния програмист.

Междувъзловите взаимодействия на свой ред могат да бъдат класифицирани в зависимост от използуваните механизми за обмен на съобщения, които на свой ред се определят от семантиката на съобщенията, в т. ч.:

- асинхронен обмен на съобщения, който се използува предимно за обмен на променливи на състоянието, както при взаимодействия “един към един”, така и “един към много” (broadcast);
- синхронизиран обмен на съобщения с потвърждение, който се използува при предаването на команди, аларми, информация за настъпили събития и пр.;
- клиент-сървър взаимодействия за обмен на програми, данни, статусна информация.

При това, асинхронният обмен се използува за реализацията както на вертикални, така и на хоризонтални взаимодействия между динамични подсистеми , докато другите два вида механизми се използуват предимно при вертикални взаимодействия, напр. взаимодействие между операторска станция и регулатор.

Практическата реализация на така описания подход за изграждане на РСУ предполага наличие на разпределена операционна система или run-time система на език за програмиране на РСУ. Известни са редица такива ОСРВ и езици (напр. SR, Cedar, V-Kernel и др.), които дават възможност за изграждане на разпределени системи с “меки” времеви ограничения [53].

По-нататъшното развитие на този подход изисква неговото усъвършенствуване с оглед реализацията на РСУ с твърди времеви ограничения. Това развитие се осъществява в рамките на съвременната теория на реално-времевите системи, като се цели реализацията на разпределени програмни среди, поддържащи интегрирано диспечиране на изчислителни и комуникационни процеси в рамките на критични по време отдалечени взаимодействия [74][164].

Така формулираната задача е обект на интензивни изследвания, като все още липсва общоприет и единен подход за начина, по който трябва да бъде решена. Следва да се отбележи, че решението Ł зависи съществено от дисциплината на диспечиране - с фиксирани или с динамично-изменящи се приоритети. Известни са експериментални системи с интегрирано диспечиране, осъществявано на принципа на най-близките времеви граници (nearest-deadline scheduling), при което се осъществява наследяване на времеви граници между изчислителните и комуникационните процеси в рамките на отдалечени взаимодействия. Горният подход обаче е свързан с редица реализационни проблеми, поради което в практиката се използват предимно системи с фиксирани приоритети. За тях се предлага значително по-прост и практически-приложим метод за интегрирано диспечиране на изчислителни и комуникационни процеси, който се изразява в следното [111][112][113][169]:

- РСУ се разглежда като съвкупност от изчислителни и комуникационни ресурси (мрежови възли, комуникационна среда), които формират своеобразен конвейер, поддържащ реализацията на отдалечени взаимодействия. В общия случай в РСУ се изпълняват множество такива взаимодействия, напр. разпределени контури за управление;
- Всеки ресурс се диспечира автономно в съответствие с приоритетна дисциплина, осигуряваща спазването на времевите граници на съответните изчислителни (комуникационни) процеси - напр. диспечиране според абсолютната стойност на времевите граници, които в случая съвпадат с периодите на процесите;
- Диспечируемостта на процесите може да бъде верифицирана с помощта на съответните аналитични зависимости [111]. При това крайната (end-to-end) времева граница на отдалеченото взаимодействие може да бъде гарантирана, ако бъдат спазени междинните времеви граници на процесите, изпълнявани в съответните ресурси.
- Величината на end-to-end закъснението следва да бъде отчетена по подходящ начин в математическия модел на РСУ (диференчно уравнение, дискретна предавателна функция и пр.).

Реализацията на така описания подход е възможна, ако бъдат елиминирани (ограничени) ефектите на приоритетна инверсия в рамките на съответните ресурси и произтичащата от тях опасност от нарушаване на локалните времеви граници. При диспечирането на изчислителните процеси тези ефекти възникват предимно при използването на общи и неделими ресурси (напр. защитени структури с данни) и се елиминират с помощта на т.нар. синхронизационни протоколи [111][169].

При диспечиране на комуникационни процеси възникват аналогични ефекти, чието решаване обаче се затруднява поради това, че комуникационната среда представлява ресурс с разпределен достъп. Това налага използването на специфични технически решения.

Следва да се отбележи, че е възможно интегрирано диспечиране на изчислителни и комуникационни процеси въз основа на статични, съгласувани

помежду си диспечерски разписания на съответните ресурси (взаимодействуващи възли, мрежа), както например, в разпределената система за управление MARS [114] и в системите, изградени въз основа на протокола TTP [73][153][154]. Този подход обаче е свързан с реализационни затруднения поради необходимостта от прецизно синхронизиране (с точност по-добра от μs) на часовниците на мрежовите възли.

1.5.2 Обектно-ориентиран подход

При тази методология за проектиране програмната система се изгражда като съвкупност от взаимодействуващи обекти; при това обектите капсулират данни и функции (методи) за обработката им, а взаимодействието между тях се осъществява чрез обмен на съобщения в рамките на парадигмата клиент-сървър [38][64][98][125]. Тази, на пръв поглед, абстрактна дефиниция има нагледна интерпретация поради аналогията с редица технически и природни системи. Така например, софтуерните обекти могат да се разглеждат като подсистеми, които си взаимодействуват (асоциират се) с други подсистеми в рамките на едно и също йерархично ниво; на свой ред всеки един от обектите може да се разглежда като композиция (агрегат) съставен от подобекти от по-ниско йерархично ниво и т.н.

Направеното по-горе разглеждане разкрива две основни свойства на обектно-ориентирания подход (ООП):

- абстракция на данните, т.е. капсулиране на данни и методи в рамките на обектите;
- модулна структура на системата, характеризираща се с хоризонтално асоциране на обекти от един и същи ранг и вертикално асоциране (агрегиране) на обекти от по-нисък ранг - в обекти от по-висок ранг.

В някои системи, респ. методи и езици за програмиране, се използват само посочените основни свойства. Това са т. нар. обектно-базирани системи, които намират широко приложение (дори когато не се използва обектна терминология) за ефективно и адекватно структуриране на сложни програмни системи [49][144].

Същинските обектно-ориентирани системи притежават някои допълнителни свойства: дефиниране на класове от обекти, динамично създаване и унищожаване на обекти от даден клас, наследяване на свойства (респ. данни и методи) между базовия клас и класа-наследник, полиморфизъм [98].

В контекста на обектно-базираните и обектно-ориентираните системи за реално време представлява интерес връзката между двете ключови понятия - обект и процес. В някои системи тези понятия са ортогонални: системата се изгражда като съвкупност от изчислителни процеси (задачи, приложения), като отделните процеси се изграждат на базата на пасивни обекти, дефинирани в рамките на някаква йерархия от класове. Подобен изчислителен процес може да бъде разглеждан в динамика като поредица от извиквания на обектни методи, активирани в определена последователност (например според потока на сигналите, обменяни между обектите в хода на изчислителния процес).

Алтернативен подход е използването на т. нар. активни обекти, капсулиращи един или няколко изчислителни процеса, т. е обекти, чиито методи могат да бъдат изпълнявани като самостоятелни процеси (нишки - threads) в контекста на обектно-структурирана система. В тази връзка следва да се подчертае, че обектният подход има значение не толкова и не само като методология за програмиране (в конвенционалния смисъл на това понятие), но преди всичко - като методология за проектиране на софтуерната архитектура на сложни технически системи, в т. ч. компютърни системи за управление [84][144].

Посочените предимства, както и мощната технологична поддръжка, определят широкото приложение на обектно-ориентирания подход при изграждане на информационни и управляващи системи, облегчавайки значително проектирането, реализацията и поддържането на програмната система. В контекста на КСУ едно важно предимство е изоморфизмът между софтуерната и функционалната структура на системата, респ. съответствието между софтуерните обекти и отделните функционални подсистеми и компоненти - датчици, изпълнителни механизми, регулатори, операторски панели и пр.

Трябва да се отбележи, че обектно-ориентираният подход има и отрицателни страни, които са по-съществени при изграждане на системи с твърди времеви граници: програмната система е по-бавна и недетерминирана, респ. непредсказуема - във времеви аспект - поради ефекти дължащи се на:

- динамично създаване / унищожаване на обекти, с произтичащата от това необходимост от динамично разпределение на паметта;
- т.нар. късно свързване (late binding) на методите със съответните програмни кодове, с оглед реализацията на полиморфизма - при работа с динамични обекти.

Посочените недостатъци са присъщи в най-голяма степен на класическите интерпретивни езици за обектно програмиране (напр. Smalltalk) и в значително по-малка степен - на компилираните езици от рода на C++, които позволяват използването както на статични, така и на динамични обекти и съответно ранно (статично) и късно (динамично) свързване [38][98].

По силата на посочените особености няма еднозначно становище за приложимостта на обектно-ориентираното програмиране в системите за реално време [49]; този подход има както привърженици така и опоненти, които са скептично настроени за възможностите му при изграждане на КСУ. Една от най-характерните тенденции, която се забелязва отчетливо през последните десетина години е все по-широкото прилагане на обектно-ориентирания подход при изграждане на КСУ с промишлено предназначение и по-специално: операторски станции, системи за супервизорно управление, както и системи за пряко цифрово управление на бавни процеси. В изброените видове приложения проблемът за работа в реално време не е критичен, при което бързодействието на компютъра успява да подтисне споменатите по-горе нежелателни ефекти, присъщи на ООП. От друга страна, съществените методологични предимства на ООП в съчетание с наличието на мощна технологична поддръжка (езици и среди от рода на RT C++, RT Java и пр.) създават предпоставки за промишлено производство на програмно осигуряване (ПО). Това е особено важно обстоятелство предвид на все по-голямата сложност на ПО и все по-масовото разпространение на КСУ, в т.ч. системи, изградени на базата на индустриални варианти на персонални компютри и 32-битови вградени системи, базирани на процесори от типа x86, XScale, ARM, SHxx, 68HC300 и други. Все по-голям пазарен дял заемат вградените системи, реализирани на базата на Windows CE и Embedded Linux. Посочените обстоятелства се отчитат от редица производители, които предлагат голям брой Windows-базирани програмни системи за управление, например Intouch 4.0, FIX DMACS, Flexis for Windows, Gello, Web, Factory Link IV, Visual Logic Controller, Control Suite, WinWor X, Genesis, Creon, Visual Control и др. [79][94][140][192][193].

В промишлените системи за управление (вкл. и изброените по-горе) широко се използва принципът на агрегиране на обекти в рамките на по-сложни (композиционни) обекти, в съчетание с принципа на програмната генерация. За целта, системата за управление се комплектува с библиотека от обекти - функционални модули. Функционалните модули се използват като градивни елементи при конфигурирането на управляващата програма. При

необходимост, последната може да бъде капсулирана в обект от по-висок ранг (композиционен функционален модул), който се включва в библиотеката като нейно разширение. В този смисъл, в съвременните системи за управление е налице тенденцията за използуване на отворени обектни библиотеки, за разлика от по-старите системи, в които номенклатурата на библиотечните модули не може да бъде променяна от потребителя.

В системите за управление процесът на конфигуриране на управляващи програми (респ. композиционни модули) е автоматизиран, като за целта се използуват специални инструментални средства, т. нар. програмни генератори (ПГ). ПГ възприема обектно-ориентираната спецификация на управляващата програма и я трансформира в изпълнима програма, съставена от библиотечни модули.

Течението, което в момента се отбелязва като основно в областта на компютърната наука – развитие на *обектно-ориентирания* подход до *модулно-ориентиран* (*компонентно-ориентиран*), е отдавна известно в теорията и все по-широко прилагано в практиката на средствата за генерация на КСУ (РСУ, ВСУ).

Съществува голямо многообразие от конфигурационни езици, като в последно време по правило се използуват графични езици със съответните средства за обобщено и детайлизирано представяне на обектите - икони и графични диаграми. Независимо от многообразието на езиците, обектните спецификации могат да бъдат отнесени към два основни класа в зависимост от начина, по който става изграждането на композиционните обекти:

- Спецификации, изградени на принципа на потока на данните (data flow). Във функционално отношение същите могат да се разглеждат като софтуерни аналози на управляващи устройства, описани с аналогови блок-диаграми (при управление на непрекъснати обекти) или логически структурни схеми (при управление на дискретни обекти). Такива спецификации се използуват предимно в системи за управление на непрекъснати обекти, напр. MATRIX/X, LabVIEW и много други [36][99][110][121][122][193];
- Спецификации изградени, на принципа на потока на управлението (control flow), които са особено подходящи при изграждане на системи за управление на дискретни обекти и по-специално, на програмируеми последователностни контролери, както и на хибридни контролери. Този подход се илюстрира с редица най-нови системи като FLO-PRO, GELLO-THREADS, Visual Logic Controller и Control Suite [78][140][172][192]. Тук спадат и езиците от класа на SDL, UML, VHDL и др.

Както беше посочено по-горе, в резултат от работата на програмния генератор се формира изпълнима управляваща програма (или композиционен обект). Последната на свой ред може да бъде компилирана или интерпретирана. Първият вариант на програмната генерация се характеризира с по-голямо бързодействие, поради което се предпочита в системи за лабораторни и експериментални изследвания, при управление на по-сложни и/или бързи обекти и процеси (напр. системата MATRIX/X [146]). Вторият вариант се характеризира с по-голяма оперативност и гъвкавост, за сметка на по-ниско бързодействие, което го прави приложим при изграждане на промишлени системи за управление.

Направените до тук разглеждания се отнасят предимно за промишлени системи и обекти с не-особено високо бързодействие. При критични по време системи, характеризиращи се с твърди и малки по абсолютна стойност времеви граници (hard real-time systems) приложението на обектно-ориентирания подход е дискусионно, предвид на посочените по-горе негови недостатъци. В такива системи се налага модифициране и/или по-нататъшно развитие на ООП, в т.ч.:

- редуциране на метода, напр. изграждане на статични обектно-ориентирани или обектно-базирани системи, в които не се използуват

специални средства като динамични обекти, наследяване и полиморфизъм, водещи до недетерминирано и непредсказуемо поведение на системата [144].

- използване на специфични средства за намаляване влиянието на нежелателните ефекти, като например описаните в [38];
- разширяване на метода чрез използване на активни обекти, т.е. обекти в които са капсулирани един или няколко процеса (нишки), взаимодействуващи си чрез структурите с данни на обекта [84][144];
- свързване на обектния подход с Теорията на системите с фиксирано приоритетно диспечиране, което е предмет на някои най-нови изследвания и разработки като: методологията за проектиране на твърди реално-времеви обектно-ориентирани системи HRT-HOOD (Hard Real-Time Hierarchical Object-Oriented Design) [51][52], езиците за програмиране Real-Time C++ [96], RT Java и Ada 95 [22][50].

Основен недостатък на процесно-ориентираните спецификации е, че при тях се акцентира на функционалната, а не на структурната декомпозиция на системата. Известно е, че структурната декомпозиция е в основата на инженерните методи за проектиране на динамични системи, в т.ч. и системи за управление. Тя е присъща на обектно-ориентирания подход за изграждане на приложни програмни системи.

Обектно-ориентираните спецификации обединяват структурните и функционалните аспекти на РСУ: в този случай системата за управление се разглежда като съвкупност от взаимодействуващи активни обекти (функционални блокове), капсулиращи подсистеми на РСУ от рода на аналогови и дискретни контролери, интелигентни входове и изходи, операторски станции и пр. На свой ред функционалните блокове (ФБ) могат да се разглеждат като композиция от обекти с по-нисък ранг, чиито операции (методи) се активират в рамките на една или повече процесни нишки, капсулирани във ФБ.

Функционалните блокове влизат помежду си в информационни взаимодействия, напр. “производител-потребител” или “клиент-сървър”. Отдалечените взаимодействия между ФБ се поддържат от комуникационен протокол, който също може да бъде обектно-ориентиран.

Така описаните ФБ се характеризират със сравнително сложна вътрешна структура (large-grain objects). Подобни обекти се използват в редица обектно-ориентирани системи, напр. Paragon TNT [93]. Възможен е и алтернативен подход, при който РСУ се изгражда от по-малки обекти (fine-grain objects), съответствуващи на сравнително-прости устройства на промишлената автоматика (т. нар. полеви устройства). Подобен подход е използуван в потребителското ниво на Fieldbus стандарта ISA SP50 [72][95], в което се дефинират елементите на конфигурационната спецификация на РСУ.

Последната се състои от функционални блокове, представляващи fine-grain обекти, всеки един от които отговаря на определено полево устройство или на негова част, напр. аналогов вход/изход, дискретен вход/изход, ПИД-регулатор, двупозиционен регулатор, брояч, регистър, логически блок и т. н. ФБ се свързват помежду си в структурна схема, респ. спецификация тип “поток на сигналите”, при което обменът на сигнали между тях се осъществява с помощта на комуникационни обекти, наречени пойнтери (агенти). Пойнтерите на свой ред се подразделят на пасивни - използувани при локални взаимодействия и активни - използувани при отдалечени взаимодействия между ФБ. В последния случай обменът на информация между отдалечените ФБ се опосредствува от приложното ниво на комуникационния протокол, който също така е обектно-ориентиран.

За целта всяко едно от полевите устройства се разглежда като съвкупност от комуникационни обекти от рода на домени (т. е зони, съдържащи програми или данни), изпълними програми, В/И променливи, синхронизационни променливи (събития, семафори) и т. н., като всеки обект експортира набор от

операции (услуги), които могат да бъдат извиквани от други полеви устройства, например четене / запис на отдалечена променлива, зареждане (download) на програмен компонент, стартиране / спиране на програма и пр.

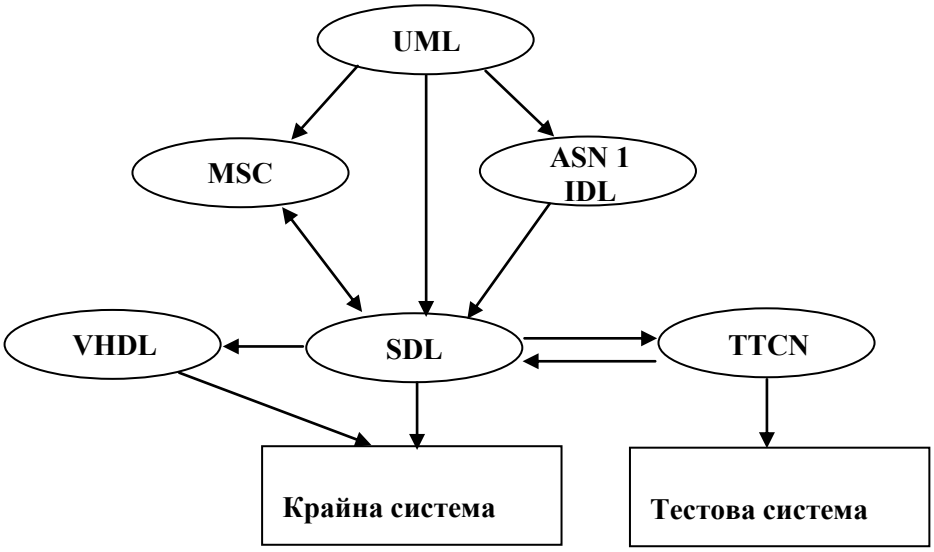
Съвкупността от комуникационни обекти се означава с термина "виртуално полево устройство" (ВПУ). По същество ВПУ представлява абстрактен информационен модел (комуникационен образ) на полево устройство, включено в състава на полевата мрежа. Функционалността на ВПУ се описва в т.нар. речник на комуникационните обекти, който е достъпен за останалите полеви устройства.

Взаимодействията на приложното ниво се поддържат от операциите на каналното ниво, в което са предвидени примитиви от вида : изпращане на съобщения със / без потвърждение, циклично разпитване и др. [154].

Подобна организация на каналното и приложното ниво имат редица други стандарти за промишлени мрежи, като PROFIBUS и FIP [26][180], чийто първообраз е в съпровождащия стандарт MMS (Manufacturing Messaging Service) разработен първоначално за приложното ниво на протокола MAP [154].

1.5.3 Езици за обектно-ориентиран дизайн на компютърни системи за управление

Основните езици, които служат като основа за обектно-ориентиран модулен дизайн на системи и в частност на системи, работещи в реално време са UML, SDL и VHDL. На тяхна база са разработени редица промишлени



Фиг. Обзор - 001

приложения. Също така те са послужили за база за специфични за определени приложни области разширения. Връзката между тези езици е показана на следната схема (фиг. Обзор-001)

Базово значение в сегашно време има UML (Unified Modeling Language) особено развитието му UML за реално време – RT UML. Най-важните характеристики на RT UML са следните:

- UML не зависи от крайния език или система, с които ще се реализира проектираната/моделираната система
- Проектираната система има свойството "четимост" (readability)
- Повторна използваемост на проектираните системи (reusability)
- "Стандартна" форма на описание. Това свойство все повече става наистина реален стандарт.
- Възможност за отчитане на хода времето в създавания модел. [70]

1.6 Методология за проектиране на КСУ, РСУ и ВСУ

От направените до тук анализи се вижда, че обектно-ориентираните спецификации на РСУ в крайна сметка се свеждат до спецификации от процесен тип, тъй като ФБ се реализират като активни обекти, капсулиращи единични или множествени процеси (процесни нишки). От друга страна обаче, в една голяма част от обектно-ориентираните спецификации не се третират детайлно въпросите, свързани с организацията на изчислителните процеси - напр. диспечиране на периодични и апериодични процеси, аналитично оценяване на диспечируемостта, реализацията на синхронизационни протоколи при използуване на общи ресурси, поддържане на времекритични комуникации, интегрирано end-to-end диспечиране на отдалечени процеси и др. въпроси, които са централни за съвременната теория на системите за реално време. Така например, в стандарта ISA SP50 тези въпроси се третират твърде оскъдно (а някои от тях се заобикалят), като за целта се предвижда използуването на статични разписания на ФБ (респ. на капсулираните в тях процеси), изпълнявани в съответните възли на РСУ, а така също - на комуникационната мрежа. Все още липсва обаче задоволително решение на проблема за интегрираното диспечиране на изчислителните и комуникационните процеси [72].

От друга страна, очевидно е, че процесите и взаимодействията между тях могат да бъдат специфицирани адекватно само в контекста на модел от по-горно ниво, в който се описват функционалните подсистеми на РСУ и взаимодействията между тях. Следователно, методологията за проектиране на разпределени програмни системи за управление трябва да отчита в еднаква степен както структурните, така и организационно-изчислителните аспекти на РСУ, което предполага йерархична конфигурационна спецификация на системата.

Например, в системата ARTS [144] подсистемите на РСУ се реализират като активни обекти с една или няколко нишки на активност (multithreaded objects), които си взаимодействуват чрез отдалечено извикване на съответните обектни операции. Посочените взаимодействия се реализират с помощта на разпределена ОСРВ, при реализацията на която са използувани съвременни методи за изграждане на системи за реално време с гарантирано и предсказуемо поведение.

Недостатък на обектно-ориентирания подход, използуван в [144] е статичната организация на системата на ниво обектни интерфейси и междуобектни взаимодействия, които се задават чрез явно (експлицитно) именуване на обектите и операциите, подлежащи на активиране. Този недостатък е избегнат в комбинирания модел на РСУ, използуван в системата SPLICE [40], където се приема, че РСУ е съставена от еднотипни обекти, всеки един от които се състои от приложен изчислителен процес и служебен процес (агент). Агентите поддържат съвместно разпределена база с данни и осигуряват динамични информационни взаимодействия между приложните процеси чрез асинхронно разпространение (broadcast) на съобщения, маркирани с етикетите на съответните комуникационни променливи. Този подход дава възможност за изграждане на големи разпределени системи, поддържащи асинхронни изчисления и комуникации между голям брой възли, вкл. системи с оперативно реконфигуриране. Негов недостатък е сравнително-високата степен на количествена сложност, което го прави приложим при реализацията на някои специални видове системи за управление (напр., големи транспютърно-базирани системи за обработка на радио- и ехолокационна информация за целите на управлението).

Необходимо е разработването на формални методи за анализ и разработка на системи за управление, които адекватно да определят структурата и поведението на системите за широко множество от възможни приложения.[101].

Този преглед е отразен в голям брой разработки, третиращи въпроса за разработка на среди, както следва:

- Реалните системи са съставени от два вида устройства - операционни и управляващи, които си взаимодействат разменяйки си управляващи сигнали и сигнали за обратна връзка . Обединението на операционно и управляващо устройство оформят подсистема и подсистемите образуват система .
- Има три основни вида система за управление: непрекъснати, дискретни (последователностни) и хибридни системи. Вградените системи принадлежат към някой от тези класове. Комплексните системи са композиция на такива системи. Следователно, процесът на разработка на софтуера трябва да започва с функционална декомпозиция на системата на подсистеми, следвана от спецификация на взаимодействията, управляващите сигнали и обратните връзки вътре и между подсистемите . Този процес на разработка може да бъде осъществен на няколко нива в сложни (йерархични) системи за управление.
- Сложното управляващо устройство може също да бъде разложено на няколко подсистеми (компоненти), принадлежащи на едно и също или различни нива на йерархия . В разпределена среда , различни елементи на системата за управление могат да бъдат разположени в отделни физически обекти (възли за мрежата) , т.е. интелигентни сензори и задвижващи механизми , контролери и т.н.
- Поведението на системата трябва да бъде дефинирано в термините на конкурентните процеси , които са капсулирани в различни подсистеми , както и междупроцесни взаимодействия вътре и между подсистеми. Възприетият процесен модел трябва да бъде достатъчно гъвкав , а освен това трябва да предоставя възможност за предсказуемо диспечиране и изпълнение на процесите и междупроцесните взаимодействия.
- Поведението на всеки процес трябва да може да бъде определено използвайки или събитийно управляван автомат, или модел отразяващ синхронизирания с времето поток на данните.В общия случай, двата вида могат да бъдат обединени в хибриден модел, който отчита както реакциите (породени от събития), така и трансформационния (по отношение на данните) аспекти на поведение на системата .
- Елементите на системата трябва да бъдат реализирани съгласно изискванията за създаване на компоненти, т.е. да могат да се използват без преработка за максимално количество случаи от съответния клас и да могат лесно да се включват в процеси, т.е. да реализират елемент на реактивна система.

Такъв модел е достатъчно общ за да включи множество системи и по-специално хибридни системи за управление, които се характеризират с взаимодействието на дискретната и непрекъсната част на системата.

Според [72] в момента не по-малко от 50% от реализираните системи за управление се проектират ad-hoc и се реализират на асемблерен език. Все по-голям брой от реализираните КСУ имат отношение към живота на хората, което поставя много високи изисквания за качеството и безопасността на реализираните системи. Предвид това проектирането на една КСУ (РСУ, ВСУ) се превръща в нетривиална задача. Изискванията към процеса на

проектиране и реализиране на една съвременна система на управление са многостранни, а в редица случаи и противоречиви (по отношение на критерии като “време за разработка/цена”, “време за разработка/качество” и “функционалност/качество”). Анализът на редица източници показва, че главните изисквания, които се поставят при разработката на КСУ са следните (без претенции за абсолютна изчерпателност):

- Времето за разработка и цената са доминиращи в процеса на дизайн на системите. Това определя *повторната използваемост* (*module reuse*) [8] на разработените елементи като главно изискване към системите за разработка. Това поражда като следствие изискването за максимална гъвкавост на разработваните модули. Така се позволява покриване на все по-разширяващите се приложни области без непрекъснато обновяване на вече създадените елементи.
- Началният дизайн на системата трябва да се провежда без разделяне на хардуер и софтуер (т.е. независимост от реализацията).
- Новите платформи трябва да позволяват интегрирана разработка (или поне конфигуриране) на хардуера и софтуера (*software-hardware co-design*).
- Възможността за реализация на конкурентни изчисления и комуникации трябва да бъде заложена в използвания модел.[2] [9]
- Трябва да има заложено разделение на поведението на системата и комуникационните модели. Това позволява повторното използуване на наличните елементи, тъй като методите за комуникация в една система не влияят върху поведението на използваните елементи.
- Трябва да са включени средства за валидиране на спецификациите на проектираната система и крайните реализации, като
  - Автоматично доказателство на теореми и проверка на модели на базата на методи като VDM [10]. Този подход обаче изисква много високо ниво на квалификация. Освен това тук възниква проблема с т.нар. “експлозия на състоянията”.
  - Автоматичното генериране на тестове е друг подход за решаване на проблема за верификацията на системите. [33][46][144]. Принципът на работа е в построяване на тестове, които отговарят на спецификацията на системата и проверка за валидността им върху реалната система. Това обаче поражда изискването да бъде верифицирана спецификацията преди да бъдат генерирани тестовете. Пример за подобна система е новият език A#.
- Моделът и реализацията да позволяват оперативно реконфигуриране на реализираната КСУ.[201]

Постигането на поставените по-горе изисквания не може да бъде направено при наличните предимно ръчни технологии на проектиране, изграждане и тестване на системите от ново поколение. Главна задача на проучвания, започнати в средата на 80-те години на 20-ти век е преминаването към “производство” на софтуер по начин, подобен на производството на електронни компоненти. При този подход се очертават няколко основни посоки на развитие:

- Генерация на КСУ на базата на спецификации на езици от високо ниво [47][105][106][173][198] и др.
- Конфигуриране на КСУ на базата на налични библиотеки от специализирани програми, частично или изцяло готови решения за типови ситуации и др.
- Преобразуване на обектно-ориентирани езици и среди за разработка на системи, работещи в реално време или създаване на езици за реално време като ADA [35][49][50][158].
- Създаване на стандарти за езици и методи за проектиране на системи

за управление в реално време – IEC 1131, IEC 61131, IEC 61499, както и платформено- независими среди като OMG-MetaMatrix – Model Driven Architecture [102][134][146].

Тези решения не са напълно противоречиви, но имат едни или други недостатъци, които правят невъзможно посрещането на всички изисквания на съвременните КСУ. Главен проблем при тези системи се явява невъзможността за оперативна реконфигурация, поради факта че системите се получават след преобразуване на входната спецификация до програмен код и последваща компилация.

Вторият най-важен проблем е, че сегашните методи за формално генериране на КСУ (особено в областта на непрекъснато разширяващата се зона на разпределените системи за управление) изискват преобладаващо познания в областта на computer science, а не в областта на control theory във всичките ѝ разновидности.

Наличните към момента средства за компютърно генериране на КСУ могат да бъдат разгледани в още един аспект. Това е типът обекти, към които са ориентирани съответните средства. Те се разделят на два основни класа – системи, ориентирани към обекти с преобладаващо дискретен начин на функциониране (автоматни системи) и системи, ориентирани към обекти с преобладаващо непрекъснат начин на функциониране (непрекъснати системи) [123]. Както беше споменато многократно, математическите модели, описващи тези два типа системи са съществено различни (автомати и диференциални уравнения) и това определя съществените различия в структурата на КСУ. Това различие обаче е изкуствено, породено поради трудностите, свързани с обединяването на различните математически форми на описание и реализиращите ги системи. Примерите в това отношение са многобройни и засягат както фирмените разработки, така и научните изследвания. Пример за това е стандартът IEC 1131, разширението му IEC 61131 и IEC 61499. Това е опит да се обединят на едно място средства за конфигуриране (или по-уместно "програмиране") на базата на противоречиви средства като "релейни диаграми", "Grafset" (т.е. модифицирани мрежи на Петри), "structured text" (паскалоподобен структурен език за програмиране) и пр. Сравнението и анализът им показват, че това са по принцип средства за създаване на дискретни системи, като само ST дава възможност за реализиране на системи за управление на непрекъснати обекти. В [66] е направен анализ как подобен опит за унификация води на практика до обратни на желаните резултати и че подобно еклектично смесване не постига нормално поставените цели. Личният опит на автора на този материал при работа със системи за генерация на КСУ, базирани на този стандарт също показва, че привидната простота в началния етап на реализацията и възможността за асоциация на реализирани модули (компоненти) за активизиране по време или по събитие води в крайна сметка до тромава система с много трудно за доказване поведение и качество. Освен това липсва каквато и да е синхронизация на достъпа до системните данни (основно глобално разположени), което рязко влошава качеството и надеждността на функциониране на породените системи.

Следва да бъдат подчертани следните основни тенденции при проектиране на КСУ (РСУ, ВСУ) на съвременния етап:

1. Стремеж към създаване на отворени системи посредством:
   - отворени модулни структури, реализирани чрез стандартни системни шини и взаимозаменяеми схемни модули;
   - отворени мрежови архитектури, реализирани на базата на общоприети системни стандарти;

- отворени системи за управление, което се постига чрез комбиниране на операционни модели, алгоритми, езикови и програмни средства;
- отворени за разширение програмни пакети и системи, функциониращи в стандартни операционни среди.

2. Създаване на адекватна методология за изграждане на "твърди" реално-времеви системи с гарантирано и предсказуемо поведение. Изследванията в тази област са съсредоточени в две основни и в известен смисъл, взаимно-допълващи се направления:
   - методи за формална спецификация и верификация на реално-времеви системи;
   - теория на системите с фиксирано приоритетно диспечиране,

   като засега само второто направление е развито до степен на практическа приложимост.
3. Все по-широко разпространение на обектно-базираните и обектно-ориентираните системи, което е предпоставка за преодоляване на т. нар. софтуерна криза, съответно - за създаване на промишлени методи за разработване на програмно осигуряване на базата на ключови концепции като софтуерни интегрални схеми, съхраняеми обекти (persistent objects), повторна използуваемост, програмна генерация и др. п.
4. Свързване на обектния подход с теорията на системите с фиксирани приоритети, в резултат от стремежа за съчетаване на предимствата на двата подхода, респ. създаване на методология за изграждане на обектно-ориентирани системи за реално време с детермнинирано и предсказуемо поведение.

Проектирането на КСУ и особено на РСУ и ВСУ се състои, в основни линии, в избор на подходяща системна архитектура и синтез (конфигуриране) на разпределена програмна система, реализираща функциите на системата за управление.

Голямото многообразие от системни архитектури, което съществува, налага специфични изисквания към организацията на програмната система. В това отношение най-добре изучени са децентрализираните системи за управление, съставени от вертикално-интегрирани автономни подсистеми - операторски и процесни станции (контролери) и др., чието програмно осигуряване се изгражда с помощта на методология, аналогична на методологията, използувана в съсредоточените системи за управление.

Предвид на вертикалния и епизодичен характер на отдалечените взаимодействия в децентрализираните системи, върху тях обикновено не се налагат твърди времеви ограничения. Такава система може да бъде реализирана чрез конвенционална компютърна мрежа, чието функциониране се описва с модела на отворените системи на ISO [157]. Подобни модели дефинират взаимодействието между отдалечените подсистеми в рамките на йерархия от комуникационни нива, формиращи т.нар. комуникационен стек. В модела на ISO стекът включва седем нива: физическо, канално, мрежово, транспортно, сесийно, представително и приложно. В някои видове мрежи се използуват по-къси комуникационни стекове: така например при локалните мрежи стекът се редуцира до три нива: физическо, канално и приложно. Отдалечените взаимодействия се осъществяват чрез примитивите на приложното ниво, които обикновенно се капсулират в специални процеси - комуникационни сървъри, функциониращи под управлението на локалните операционни системи на мрежовите възли. На свой ред тези примитиви се реализират чрез механизмите и взаимодействията, заложени в по-ниско стоящите нива на протокола.

Друг начин за изграждане на децентрализирани системи се състои в използуването на широко-разпространени мрежови операционни системи и протоколи, напр. Unix+TCP/IP, QNX, Novel, Windows CE и др.

Значително по-сложни и по-малко разработени са въпросите, свързани с изграждането на “същински” РСУ, т.е. на системи, които са не само физически, но и логически разпределени. В общия случай в подобни системи се поддържат както вертикални, така и хоризонтални взаимодействия. При това, последните се характеризират с твърди времеви ограничения, произтичащи от динамиката на РСУ, която на свой ред се определя от динамиката на процесите, протичащи в околната среда и по-специално - в обектите за управление.

При изграждането на такива системи трябва да бъде решен сложен комплекс от взаимно-свързани задачи, което предполага подходящ избор на системни решения, в т. ч.:

- Обща организация на КСУ (РСУ, ВСУ)
  - виртуален единичен компютър, функциониращ под упавлението на разпределена операционна система за реално време (ОСРВ) или
  - отворена система, съставена от взаимодействуващи автономни обекти (компютри, възли)
- Приципи на функциониране
  - взаимодействие между подсистемите на КСУ: системи "клиент-сървър", респ. "производител-потребител"
  - механизми за активиране на взаимодействията: синхронни периодично-активирани системи (time-driven systems), респ. асинхронни системи, активирани при настъпване на събития (event-driven systems)
- Диспечиране на ресурси
  - статично диспечиране с помощта на предварително изготвени диспечерски планове (off-line schedules)
  - динамично диспечиране (on-line scheduling) - с фиксирани или с динамично-изменящи се приоритети
- Конфигурационна спецификация на програмната система
  - процесно-ориентирана или обектно-ориентирана спецификация
  - комбинирана процесно/обектно-ориентирана спецификация
- Разполагане на софтуерните компоненти (обекти, процеси) върху възлите на мрежата
  - статично разполагане на компонентите
  - динамично разполагане
- Адресиране на съобщенията
  - явно именуване на комуникиращите обекти / процеси
  - явно именуване на комуникационни обекти (портове, канали)
  - асоциативно адресиране, т.е. адресиране по съдържанието на съобщението (content-oriented message addressing)
- Организация на компютърната мрежа (ако има)
  - мрежи с централизирано управление на достъпа (master-slave)
  - мрежи с децентрализирано управление на достъпа - token ring, token bus, TDMA, CSMA/CR (CAN) и др.

Както се вижда от направените разглеждания, съществува голямо многообразие от алтернативни решения (и комбинации от решения) на така формулираните задачи. Ето защо, липсва общоприет подход по отношение на софтуерната архитектура и методологията за проектиране на разпределени системи за управление в реално време, независимо от опитите за стандартизирането им. Все пак, могат да бъдат проследени някои характерни тенденции.

Методологията за изграждане на разпределени програмни системи за управление в реално време зависи съществено от начина, по който се третира компютърната мрежа на РСУ, при което са възможни два подхода:

- компютърната мрежа се разглежда като набор от автономни компютри, съдържащи отделни подсистеми, които си взаимодействуват чрез мрежовата среда в процеса на функциониране на РСУ;
- компютърната мрежа се разглежда като виртуален единичен компютър с еднотипна (и прозрачна за приложния програмист) организация на междупроцесните взаимодействия, независимо от разположението на изчислителните процеси в мрежата.

В първия случай възлите на мрежата функционират под управлението на локални ОСРВ, а отдалечените взаимодействия се поддържат от специализиран протокол, реализиран въз основа на редуциран вариант на модела на ISO. Така например, протоколите на полевите мрежи използват комуникационен стек с три нива: физическо, канално и приложно [154]. В някои случаи (напр. в стандарта ISA SP50) над приложното ниво се въвежда т. нар. потребителско ниво, в което се дефинират функционалните подсистеми на РСУ, респ. връзките и взаимодействията между тях.

При вторият подход за изграждане на РСУ програмната система се реализира въз основа на разпределена операционна система за реално време (ОСРВ), която осигурява виртуална компютърна среда, поддържаща изпълнението на процеси и междупроцесни взаимодействия с твърди времеви граници. При това, отдалечените взаимодействия се опосредствуват от опростен протокол, дефиниращ мрежовите взаимодействия на физическо и канално ниво [53]. Някои автори и научни групи считат, че този подход е по-подходящ при реализация на системи за управление на сложни динамични обекти, напр. монтажни, крачещи и автономни роботи. Въпреки това, по-голямо разпространение е добил първият подход, тъй като дава възможност за изграждане на “отворени” системи, включително и системи с взаимозаменяеми блокове, произвеждани от различни потребители.

Друг кръг от задачи при проектирането на КСУ/РСУ е свързан с основополагащите принципи, които определят начина на работа на системата. Един такъв принципа е взаимодействието “клиент-сървър”, която в последно време намира широко приложение при изграждането както на вертикално-интегрирани, така и на някои видове хоризонтално-интегрирани системи (предимно системи с по-ниска сложност и/или бързодействие). Тук проблем е невъзможността за разпространяване (broadcast) на променливи на състоянието между динамичните подсистеми и от друга страна - не използва присъщите на локалните мрежи възможности за осъществяване на такава комуникация. Принципът “производител-потребител” е освободен от този недостатък. Поддържат се както двуточкови взаимодействия (един към един), така и многоточкови взаимодействия, в т. ч. “един към много” (broadcast), което е особено важно при управлението на по-бързи обекти. Поради това той намира приложение в редица видове промишлени мрежи и стандарти, напр. FIP, CAN, CAnOpen, DeviceNet, WestNet и пр. [36][58][59][60][85][150][180][188][199]. Този механизъм на функциониране, в съчетание с асоциативното адресиране на съобщенията , дава възможност за поддържане на разпределена база данни с копиране на данните по възлите на мрежата (както например в системата WestNet на Westinghouse), с което се изключва необходимостта от клиент-сървър взаимодействия. Това обаче предполага мрежа със сравнително висока пропускателна способност, както и увеличен обем на паметта, предназначена за съхраняване на данните, получавани от други възли. Ето защо, в повечето системи се използва комбиниран механизъм: “производител-потребител” - при критични по време хоризонтални (и вертикални) взаимодействия, свързани с обмен на променливи на състоянието и “клиент-сървър” - при некритични вертикални взаимодействия, напр. при обмен на команди, програми, статусни данни и пр.

Системите от асинхронен тип се характеризират с много по-малък комуникационен товар в установен режим на работа, в сравнение със синхронните мрежи. Така например, при реализацията на разпределен контур за регулиране чрез асинхронна система, съобщения се пораждат и предават в мрежата само при съществено отклонение на регулираната променлива от предварително зададената стойност. За сметка на това, поведението на подобна система става непредсказуемо при настъпване на сериозни нарушения на технологичните режими (напр. аварийни ситуации), което може да доведе до претоварване на мрежата и неадекватно поведение на системата за управление. В това отношение синхронните системи имат съществено предимство, а именно - детерминирано и предсказуемо поведение, което е инвариантно от режима на работа и не се влияе от случайни фактори и събития в околната среда. При това, поведението на системата (в частност - диспечируемостта на мрежата) може да бъде верифицирано a-priori с помощта на аналитични зависимости, изведени в Теорията на системите с фиксирано приоритетно диспечиране [6][54][111][168][169][181][183][184]. По тази причина повечето специалисти отдават предпочитание на синхронните мрежи при изграждането на системи за управление с гарантирано и предсказуемо поведение.

В контекста на направените разглеждания представлява интерес един частен случай, отнасящ се до системите с отдалечен вход / изход. Подобни системи имат двойнствена природа - макар и да са физически разпределени те са логически централизирани, доколкото обработката на информацията е съсредоточена в централния компютър на системата.Това облекчава значително реализацията на отдалечените взаимодействия, които се редуцират до локални взаимодействия между приложни процеси (клиенти) и комуникационен сървър, в който са капсулирани процедурите на отдалечения достъп до информацията, съхранявана във входните и изходните възли на РСУ [148].

Възможни са различни варианти за реализация на сървъра, който може да бъде локален процес на управляващия компютър или самостоятелен възел (network manager). При това, сървърът може да бъде активиран по различен начин:

- периодично - при изтичане на интервал от време;
- при настъпване на събитие в обекта; в) по заявка на клиента.

Възможни са и различни варианти на взаимодействието между сървъра и приложните процеси - например чрез обща база данни или чрез обмен на съобщения в рамките на съответния протокол за клиент-сървър взаимодействие. Така описаният подход е характерен за системите с отдалечен вход / изход, в т.ч. и някои най-нови образци като системата FLEXNet [175].

1.7 Разполагане на софтуерните компоненти и диспечиране на ресурсите в КСУ/РСУ

Централен кръг от въпроси при проектирането на РСУ е свързан със структурирането на РСУ и подсистемите Ł (респ. с избора на подходяща конфигурационна спецификация), както и с механизмите за обмен на съобщения между подсистемите на РСУ при реализиране на отдалечени взаимодействия. Във връзка с това възникват редица специфични проблеми, свързани с:

- разполагане (allocation) на софтуерните компоненти на РСУ;
- диспечиране на ресурсите, респ. на софтуерните компоненти, разположени върху тях.

Подсистемите на КУС/РСУ могат да се разглеждат като софтуерни обекти, които на свой ред са съставени от софтуерни компоненти - обекти и/или процеси, в съответствие с вида на използуваната конфигурационна спецификация. При конфигурирането на системата за управление

софтуерните компоненти се разполагат върху възлите на мрежата. При това, разположението може да бъде статично (т.е. неизменно) или динамично, което предполага възможност за миграция на компонентите в мрежата.

Така формулираният проблем е в пряка връзка с диспечирането на ресурсите и съответно - на софтуерните компоненти, разположени върху тях. В случая са възможни два подхода:

- статично диспечиране на ресурси;
- динамично диспечиране на ресурси.

При първия подход системните ресурси (процесори, мрежа) се разпределят априорно между изчислителните, респ. комуникационните процеси, като за целта се изготвят off-line разписания (диспечерски планове) за всеки един от ресурсите. Тези планове представляват циклично-повтаряни последователности от времеви интервали, закрепени статично за съответните процеси (cyclic scheduling). При това, диспечерските планове на отделните ресурси (напр. възел-източник, мрежа, възел-приемник) трябва да бъдат съгласувани така, че да се гарантират времевите граници както на локалните, така и на отдалечените взаимодействия [73][114].

Привържениците на статичното диспечиране твърдят, че това е единственият възможен начин за изграждане на "твърди" СУ с гарантирано поведение. Същевременно, този подход има някои очевидни недостатъци и преди всичко - липса на гъвкавост и значителни разходи на труд и време при конфигурирането и реконфигурирането на мрежата и програмното осигуряване на СУ [176]. Ето защо, напоследък се утвърждава тенденцията за изграждане на системи с динамично диспечиране на ресурсите, в контекста на съвременната Теория на системите с фиксирани приоритети [92][111][112], което се характеризира със следните по-важни особености:

- статично разполагане на процесите (обектите) по възлите на мрежата;
- динамично (on-line) диспечиране на изчислителните и комуникационните процеси с използуване на фиксирани приоритети (rate-monotonic / deadline-monotonic scheduling);
- гарантирано изпълнение на процесите в рамките на времевите им граници, като за целта се оценява предварително тяхната диспечируемост (schedulability);
- интегрирано диспечиране на изчислителни и комуникационни процеси.

В заключение следва да се отбележи, че съществува и друго, по-широко, понятие за динамично диспечиране на изчислителните процеси в мрежи, което се свързва с възможност за миграция на процесите по време на работата на системата [53]. В някои системи за реално време това се прави, за да се гарантира (по възможност) спазване на времевите граници на процеси, текущо разположени във възел, който е в състояние на изчислително претоварване. Така описаният подход е приложим при системи за управление от по-високо ниво, характеризиращи се с динамичен изчислителен товар, напр. командно-контролни системи, системи за оперативно-диспечерско управление и др.п., при които инцидентното нарушаване на времева граница не е свързано с фатални последици.

В съществуващите СУ се използува предимно статично разполагане на софтуерните компоненти в комбинация със статично или по-често динамично (on-line) диспечиране на изчислителните и комуникационните процеси. Това се обяснява както с особеностите на РСУ, които по правило са съставени от функционално-обособени възли (подсистеми), така и от изискването за работоспособност и предсказуемост на системата. Критерий за работоспособността на СУ е диспечируемостта на системата, респ. на съставните изчислителни и комуникационни процеси, която може да бъде оценена с помощта на аналитични зависимости, изведени при направените по-

горе предположения за статичен изчислителен товар и динамично диспечиране на изчислителните и комуникационните процеси [112][169].

Ясно е, че методите за дизайн на софтуер трябва да включват и двата подхода – обектни и процесно-ориентирани спецификации в йерархичен модел. Като резултат от съобразяването с тези изисквания е пораждането на вече многократно споменатите хибридни архитектури и среди за разработка като ARTS, SDL'92 и др., които са едновременно обектно и процесно базирани [84][144]. Така системата е породена като композиция за активни обекти, всеки от тях включващ една или повече нишки (control threads). Нишките извикват операции на пасивните обекти, които могат да извикат операции на други обекти и т.н. Този подход води до създаване на добре структурирани системи, характеризиращи се с ясна йерархия на активни и пасивни обекти. Слабост на подхода е, че всички обекти – активни и пасивни, структурата, връзките между тях и всички други елементи се дефинират от системния дизайнер.

Ad-hoc дизайнът и спецификации ограничават силно повторната употреба на вече разработени компоненти. Това налага необходимостта от разработване на среда, която ще позволи систематична спецификация на реконфигурируеми компоненти, които ще бъдат повторно използваеми по дефиниция [44][44][171]. Това се постига като разширение в стандартите за промишлен софтуер, но те са от относително ниско ниво, пасивни обекти, функционални блокове и подобни [102][134]. Няма обаче никаква поддръжка на реализация на повторно използваеми автомати или хибридни модели, включващи повторно използваеми автомати или функционални блокове. В системи, като напр. AIRES [172][189][190] и STATEWORKS [97], има такава поддръжка, но пък те не поддържат функционални блокове и блок-диаграми в смисъла на IEC 61131-3 и други подобни стандарти.

Взаимодействието на компонентите е друг главен момент, което трябва да бъде решен при специфициране на системната конфигурация. Известни са много видове взаимодействия, които широко са използувани в контекста и на процесно-ориентираните, и на обектно-ориентираните системи: клиент-сървър, производител-потребител, издател-получател. Клиент-сървър е много популярен в IT. Той е използван и в промишлените системи и протоколи. Той обаче има ограничения, произтичащи от блокиращия характер на комуникацията и двуточковата връзка. Поради това комуникацията тип "производител-потребител" е по-подходящ за системи за реално време - комуникацията е неблокираща и поддържа връзки тип "един-към-много" [55]. Комуникацията тип "издател-получател" комбинира полезните отличителни черти на двата други метода и също се използува широко.

Горните образци могат да бъдат реализирани чрез интерфейси на компонентите, които са дефинирани в компонентния интерфейсен модел. Вграден ите системи използват обикновено два вида такива модели: А) моделът на IEC 61131 определя интерфейсите на компонентите в термините на сигналните входове и изходи, където изходът на компонента е директно свързан към един или повече входове на други компоненти (функционални блокове); В) взаимодействия, базирани на портове. Обектите си взаимодействат чрез подходящо свързани входни и изходни портове, които са реализирани като общи области в паметта. Този модел е бил развит за приложения в роботиката [20], но се използува и в други области, като SDL, ROOM, PECOS, SOFIA и др. [84][149][199] Това предлага по-висока гъвкавост, но внася елемент на усложняване на модела.

Този проблем е решен в моделите, подобни на IEC 61331 чрез безусловно дефиниране на буфери за вход/изход и програмно свързване на входове и изходи, което води до прости и лесни модели - функционални блок-диаграми [102]. В този случай, входно/изходните буфери са дефинирани в съответните изпълними записи на свързаните инстанции на функционалните блокове, където стойността на входната променлива може да бъде получена от съответният изход използвайки или присвояване, или дори по-добре - указател посочващ

източника на входно-изходната връзка. Тази техника е директно приложима към компоненти на ниско ниво, каквито са функционалните блокове, но то не е удачно при разпределени приложения. При тях е необходимо да се използват специална компоненти, т. нар. интерфейсни функционални блокове [134], което води до загуба на прозрачност. Поради това е необходимо развитие на този подход, така че подсистемите да бъдат програмно свързани (softwired) с една с друга по унифициран начин и сигналите (данните) са разменят прозрачно между компонентите, независимо от физическото им местоположение. Това изисква разработването на специален протокол, поддържащ прозрачна сигнално-базирана комуникация между взаимодействащите си компоненти. [15][16][17][71].

Два подхода в компонентно-базирания дизайн се открояват в последно време. Първият следва по-конвенционална линия за развитие, която започва с изпълними модели и системи за бързо създаване на прототипи. Той може да бъде характеризиран като компютърно-подпомагано генериране (computer-aided generation) на системи за управление на базата на среди и библиотеки от компоненти (най-често в текстов вид). Този подход е застъпен в много проучвания и комерсиални системи като ROOM, Object Time Developer, Rational Rose, Rhapsody и др. Програмната генерация обаче има един сериозен проблем: не предоставя адекватна поддръжка за динамични реконфигурация, тъй като изисква генериране и компилация на новия код, който трябва да бъде зареден впоследствие в целевата системата. Освен това, програмната генерация е силно повлияна от съществуващите методи и езици за системен дизайн (напр. UML / C+ +), което води до създаване на приложения, които са ориентирани към системи от висок клас, имащи достатъчно изчислителна мощност и памет. Налични са някои среди, които са специално ориентирани към малки вградени системи [92]. Тези системи са базирани на изпълнение на конкурентни автомати (FSM), които са адекватен модел за системи, работещи в твърдо реално време и поради това времевият анализ е затруднен.

Вторият подход следва линия за развитие , която започна с някои системи за промишлена автоматизация преди няколко десетилетия. То може да бъде характеризирано както конфигуриране на системите за управление чрез използване на формални среди и предварително създадени изпълними компоненти. Последното може да бъде реализирано като премествaеми “силиконови” библиотеки, записани на енерго-независима памет, където за всеки компонент са определени функционалните му характеристики и ресурсите, които използва (памет, време за изпълнение на определен процесор и др.) Конфигурацията се записва в RAM-таблици, които съдържат реално-времевата база данни на приложението. При този вид системи програмната генерация се използва в ограничен обем само за да генерира код за компонентите от ниско ниво, като базовите функционални блокове, които ще бъдат включвани в компонентната библиотеката.

Софтуерната конфигурация очевидно е по-добрият избор в термините на качество на услугите (QoS) и предпоставките , които са нужни да се направи преход към промишлен етап на производство на вграден софтуер. Неговото главно преимущество е вградената поддръжка за възможност за реконфигурация на система, която се постига чрез обновяване на структури от данни, докато изпълнимите кодове остават непроменени. Системна реконфигурация може да бъде приложена off-line , т.е. преди системата да е рестартирана или евентуално, on-line по време на работа на система. Въпреки това, конфигурация on-line е извънредно сложен проблем, изискващ задълбочено проучване в различни области:

- Специфични проблеми в контекстът на динамично управлявани (диспечирани) системи, като протоколи със сменяем режим, детерминизъм на отговора и т.н. [137][155] .
- Фундаментални проблеми на динамиката на системата по време на реконфигурацията, които са извън основните въпроси от софтуерната

реконфигурация (устойчивост на системата, безударни превключвания и др.).

Очевидно, тези са проблеми, които трябва да бъдат разглеждани за всеки специфичен клас системи в дадена област на приложение , т.е. модални управляващи системи, устойчиви на грешки системи и т.н., използвайки методологии, развита в съвременната теория на хибридните системи .

Анализът и изброяването на известните подходи за създаване на системи за управление ясно показва, че въпреки голямото многообразие от стандарти, среди и подходи за разработка на системи за управление е необходима още работа в тази посока и разработка на нови среди за генериране на КСУ. Защо трябва да се създават тези нови среди?

- Ходът на времето като събитие систематично се премахва в теорията за изчисления. Фактът, че изчисленията продължават ненулево време най-често просто се изпуска. Рядко се прилага характеристика на управляемите програмни единици (процеси или нишки), която да включва съществени времеви характеристики. Често даже в системи, поддържащи темпорална логика, времето по-често е качествена, а не количествена характеристика. Реализациите на смесен ОО дизайн с включване на реално време са сравнително неуспешни [70]. По-удачни са директните системи за представяне на времето като системен елемент, напр. хронирани В/И автомати (timed I/O automata). Използването на термина “синхронен” най-често също няма отношение към хода на времето, а само до координираната или подредена поява на някакви събития/действия. Това смесване на значенията може добре да се забележи и в този обзор. При системите за работа в реално време все пак този подход е неприемлив. Проблемът се разраства особено, когато се стигне до разпределени системи, работещи в реално време.
- Управлението на конкурентността (паралелността) на работа все още е сравнително “примитивно”. Главен синхронизационен механизъм остават семафорите и мониторите. Те са трудни за използване и освен това са потенциален източник на множество грешки, включително такива изключително тежки ситуации като взаимна и транзитивна блокировка и приоритетна (вкл. неограничена) инверсия. Теорията на паралелните изчисления не е директно приложима, особено при разпределените системи за управление, но нейната адаптация може да даде нови възможности за създаване и управление на паралелни (и конкурентни) изчисления.
- При системите за управление и особено при вградените системи отказоустойчивостта на системата (тук е по-правилно да се каже живучестта “liveliness”) е от решаващо значение. За разлика от обикновените програми, които не завършват сами при програмна грешка, при управляващите системи програмите са с безкраен цикъл на работа.
- Абстрактните типове данни са формалния метод, който осигурява в най-висока степен коректността на софтуера. Обектно-ориентираните езици оказват голямо влияние върху повторната използваемост на софтуера и неговото качество. Съвместно с шаблони за проектиране [89] и моделирането на обектите , типово-базираните системи дават нов начин за осмисляне на структурата на програмите. Това обаче има недостатъка, че става дума само за статична структура. Не се задават нито динамични, нито времеви характеристики на обекта като част от типовото му описание. Други обектни форми, като активните обекти и участници (“actors”) дават по-точно описание на динамичните характеристики, но не описват интерфейсите и реално-времевото

поведение. [3][5]

- Разширените типово-базирани системи трябва да обхващат следните аспекти на системите:
  - протоколи за комуникация между процеси (т.е. рандеву , асинхронно предаване на съобщения, потоци, събития) ;
  - модели на времето (т.е. континуум /непрекъснато/, дискретно, тактувано, частично подредено);
  - управлението на изпълнението(т.е. синхронно , активиране от събития, диспечиране на процеси, реално време).
- Архитектурата на съвременните компютри включва редица елементи за ускоряване на работата. Използува се кеширане на инструкции и данни, паралелно изпълнение на инструкции, предсказване на преходите и др. Това компрометира надеждността на системата за управление. Поради това тези техники не се използуват в системите за управление – напр. DSP или микро-контролери. За да може тези значителни по своето влияние ускорителни техники да могат успешно да се прилагат, е необходимо да се създадат абстракции, които правилно ги описват. Така ще може да се отчита влиянието им върху скоростта на изпълнение на програмите (оттам и по-прецизно отчитане на хода на времето и времевите характеристики на програмите).
- Разполагането на компонентите на системата по апаратните възли трябва да е независим етап при проектирането и да не влияе на общия системен дизайн. Формирането на локалния и мрежов обмен на данни трябва да се поражда автоматизирано като резултат от това разполагане и да бъде прозрачно за системния конструктор. Трябва да се включват и методи за измерване на времената за реакция и времената за транспорт на данни на отделните възли и компоненти с цел извършване на off-line и on-line времеви анализи за диспечируемост.
- При разпределените системи за управление се забелязва, че ядрата на възлите все повече се ориентират от time-triggered към event-driven, докато мрежата за реално време прави точно обратното – има ясен стремеж да се елиминира състезателния елемент върху мрежата и event-driven протоколите да се използуват само за некритични приложения. Това предполага включване на съответните механизми в средите за проектиране.[163]

Цел на дисертационната работа

Основна цел на дисертационната работа е да бъде разработен формален модел, който да позволява:

- описание на хетерогенна разпределена компютърна система за управление (РКСУ) по естествен за специалистите по автоматично управление начин
- създаване на програмен генератор за възприемане на спецификация на РКСУ и преобразуването ѝ с цел реализиране на РКСУ.

За постигането на поставените цели трябва да бъдат решени следните задачи:

- Изследване на модели за формално описание и реализация на компютърни системи.
- Изследване на модели за описание на системи за управление във връзка с моделите за описание на компютърни системи.
- Изследване на начините за формално описание на разпределени компютърни системи и на разпределени компютърни системи за управление.
- Изследване на програмни езици, програмни генератори и среди за разработка на РКСУ.
- Синтез на модел за описание на компютърни системи за управление, който:
  - е независим от физическата структура на компютърната система
  - позволява адекватно описание на хибридна КСУ.
- Синтез на системни компоненти, чрез които да може адекватно да бъде описана и реализираната КСУ.
- Разработване на метод за автоматизирано пренасяне на синтезираната система за управление върху физическата структура на компютърна система (разпределена/съсредоточена).
- Разработване на методи и средства за верификация на времевата коректност на РКСУ, отговаряща на предложения модел.
- Реализация на програмен генератор, реализиращ специфицирането, разполагането и верификацията на РКСУ, според предложения модел.

Бъдещо развитие

Цялостна реализация на среда за проектиране на разпределени компютърни системи за управление със следните възможности:

- Възможност за on-line анализ на времевите характеристики на реалната физическа система,
- Възможност за формална верификация на автоматната структура на генерираната система за управление,
- Валидация на трансформационните части на генерираната система.

Глава 2 – Формален модел на разпределена система за управление

2.1 Постановка на задачата

Основен проблем при създаването на компютърни системи за управление се явяват времевите ограничения, които налагат върху реализацията характеристиките на реалната компютърна система – нейната операционна система, използваната комуникационна мрежа за реално време, характерът на обекта за управление (преобладаващо непрекъснат или преобладаващо дискретен). Всичко това създава редица проблеми, които са особено ясно видими при създаване на разпределени системи за управление. При тези системи обикновеният подход е частична декомпозиция на системата за управление според предвижданите или налични управляващи компютри, конфигуриране (или програмиране) на всеки един от тях като централизирана система за управление и осигуряване на обмена на данни (променливи на състоянието, команди и отговори) между отделните управляващи компютри. Тази практика води до факта, че решения, свързани с архитектурата на управляващата система – брой възли, мрежа, операционна система и др. такива, стават преобладаващи спрямо решенията, отнасящи се до структурата и функциите на управляващата система от гл. т. на управление на обекта на управлението.

Цел на тази работа е предложение на модел, който да съчетава следните страни:

- да описва достатъчно общо разпределени и съсредоточени системи за управление;
- да описва адекватно и без изключения хибридни (непрекъснати, дискретни и дискретно/непрекъснати) системи за управление;
- да позволява изграждане на системата за управление на базата на модерни средства за автоматизирано проектиране/генериране;
- да бъде отворен тип;
- структурата и функциите на средствата за автоматизирана генерация на системи за управление да позволяват проектирането и реализацията да започва от естествената посока – формулиране на функциите и структурата на системата за управление и едва след това отразяването ѝ върху реалната физическа структура;
- диспечиране на процесите на базата теорията за диспечиране с фиксирани приоритети и изследване на тази база на реалната функционалност на системата – диспечируемост, функционална валидност и др.;
- прозрачна, контекстна-ориентирана комуникация между процеси-те/обектите, базирана на приоритетно-ориентиран достъп до комуникационната среда.

2.2 Формален модел на системата

Известен факт е, че системите за управление са разделени на два основни класа – системи за управление на обекти с преобладаващо непрекъснато поведение (описвани основно с диференциални уравнения) и системи за управление на обекти в преобладаващ дискретно поведение (описвани основно с различни автоматни модели). В реалния свят най-често срещаната ситуация са обектите от смесен тип – дискретно-непрекъснатите обекти.

При работата на компютърната система за управление на дискретен обект преобладават решенията, основаващи се на факта, че обектът, а оттам и

управляващото устройство работят като събитийно-активирана система. Това е естествено, тъй като обектите от този тип най-често са различни видове автомати.

Обратно – при системите за управление на обекти от непрекъснат тип преобладаващата тежест в управляващата програма се пада на различни трансформации на данни – от входни сигнали с различен характер, на базата на числени преобразувания се получават стойностите на изходните (управляващи) сигнали.

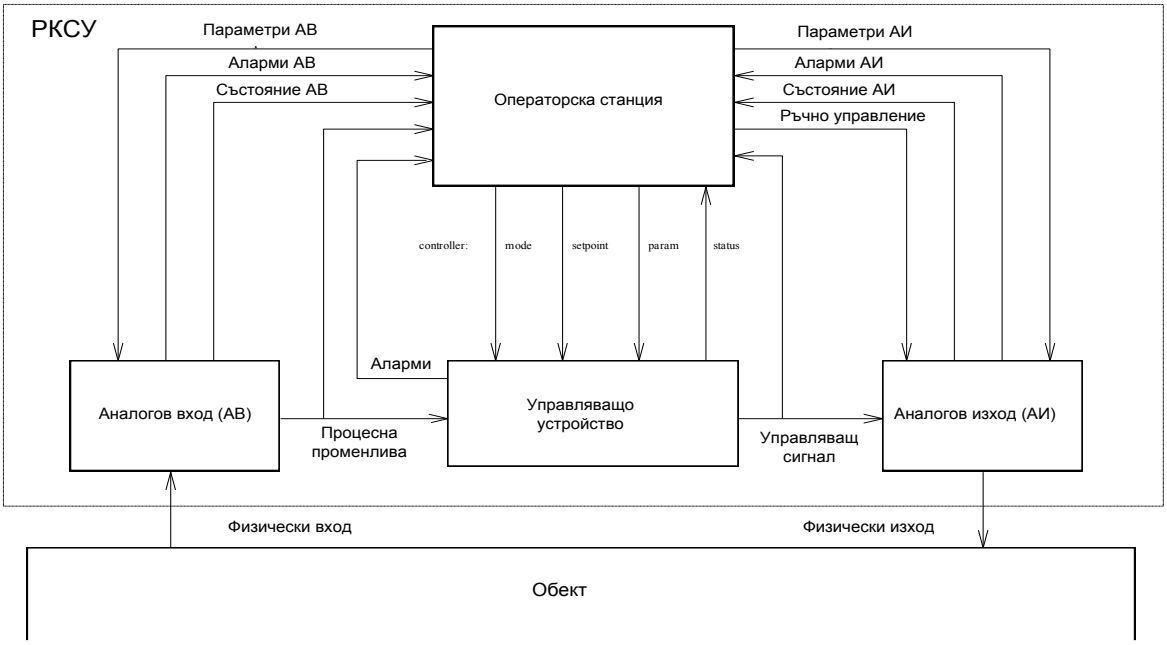
При обект от хетерогенен тип, който има както автоматно поведение, така и непрекъснат характер, се вижда нуждата от управляваща система, която да може да взаимодейства с обекта както събитийно (автомат), така и на базата на обмяна на аналогови сигнали „обикновен" регулатор или др. подобно).

Системата за управление трябва да е изградена на такава база и от такива компоненти, че да може да позволява аналитично изчисление на диспечируемостта на всички процеси и на комуникацията. Оттук нататък под „диспечируемост" ще разбираме възможността на всяко отделно взето и логически разграничимо действие да завърши своята работа в определения за това времеви интервал (deadline).

Така посочените изисквания са изходна точка за разработване на софтуерна архитектура за разпределени компютърни системи за управление. Тук и нататък ще бъде използвано името „Системи за управление с предсказуемо поведение"- СУПП (*Automation Systems with Predictable Scheduling (ASPS)* [13]. Примерна структура на РКСУ, обект на разглежданията е показана на фиг. 2.1.

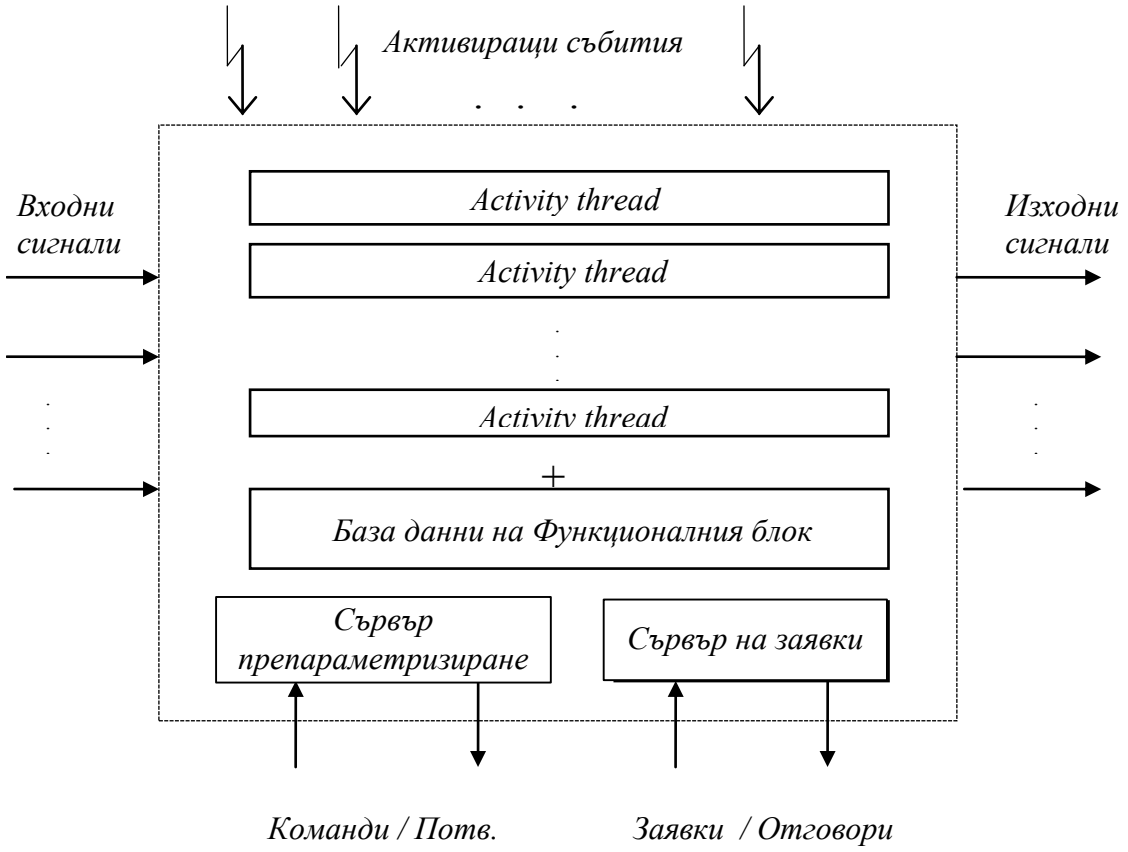
Фиг. 2.1 Обобщена структура на РКСУ – поток на данните

СУПП (*ASPS)* е софтуерна архитектура за проектиране и реализация на hard real-time РКСУ. Едно неформално описание на предлаганата архитектура в термините на РКСУ е следното:



Конфигурацията на РКСУ се определя като компонентен модел, базиран на автономни обекти наречени “Функционални блокове” (ФБ). Тези ФБ си комуникират на базата на обмен на сигнали (наименовани променливи). Комуникацията не предвижда ограничения според типа на информацията (команди, променливи на състоянието и др.), периодичността (периодични, спорадични), а така също и интегритета (прости или композитни данни/команди/запитвания). Пример за обмен на съобщения между ФБ е показан на фиг. 2.1.

Фиг. 2.2 Обобщена структура на Функционален блок

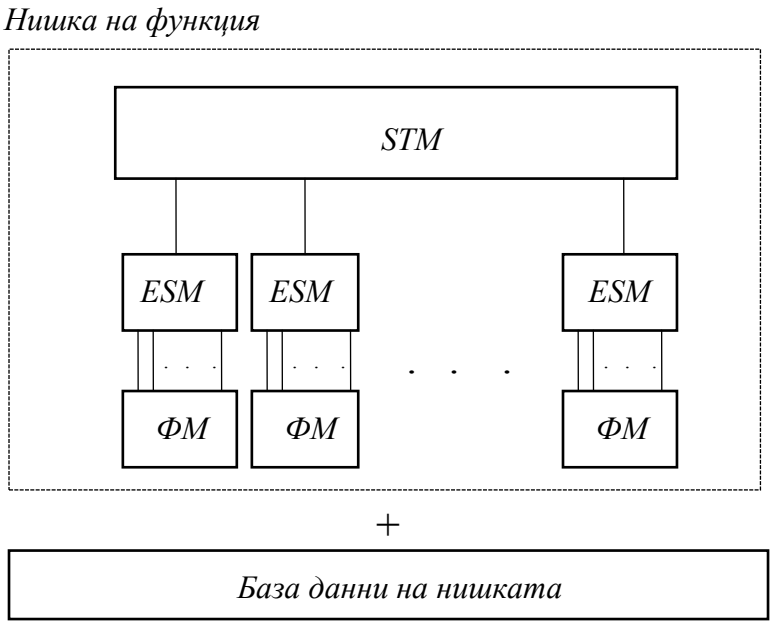


ФБ е активен обект, съдържащ в себе си някакъв набор от събитийно-активирани подобекти (функции). Тези подобекти отговарят за генериране специфични според приложението отговори (реакции) на съответно настъпилите вътрешни или външни събития. Изтичането на определени интервали от време (синхронизационни събития) се третира като обикновено събитие.

„Функциите” са съставни обекти. Те са конструирани от обекти (елементи) от по-ниско ниво (функционални модули). Функционалните модули на този етап могат да бъдат разглеждани като стандартни библиотечни програми, реализиращи различни изчислителни и логически процедури (напр. PID-регулатор, филтър и пр.).

„Функциите”, реализиращи реакциите на ФБ, са реализирани като нишки (activity threads). Тук нишка се използува в смисъл на самостоятелно диспечируема (и активизируема) програмна единица (но не подпрограма). Съвместно с базата от данни за реално време те представляват основните елементи на ФБ. Логически нишките на ФБ могат да са ориентирани към реализиране на реакции, отговарящи на съответни събития, породени от обекта за управление, синхронизационни събития и др. подобни, а така също и на събития, породени от постъпване на команди/запитвания от други ФБ и подобни, но това не води до структурна разлика между тях, а само до разлика в програмните структури според очакваните реакции. В този смисъл например „регулатор” и „команден интерпретатор” са идентични. Обща структура на нишка е показана на фиг. 2.3.

Фиг. 2.3 Обобщена структура на нишка

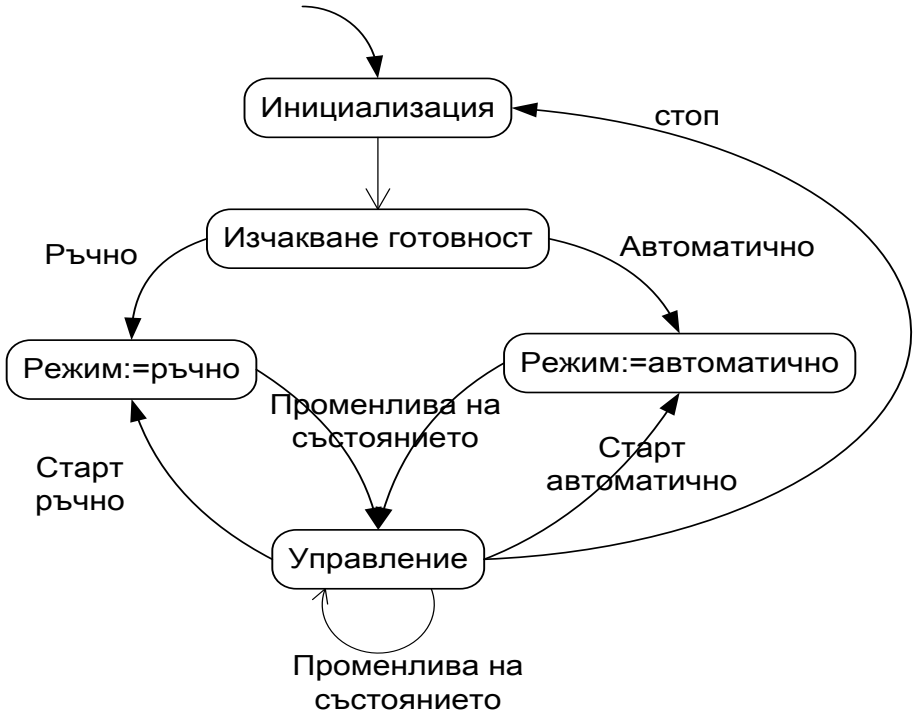


Нишките реализират заложената в тях функционалност на базата на интерпретиране на *графа на състоянията (state transition graph)* – фиг. 2.4. и асоциираните с различни негови състояния *графове на потоците на данни (signal flow graph)*. – фиг. 2.5.

(Заб.: С оглед на елиминиране на опити за превод на термини, по-надолу в тази работа ще бъдат използувани общоприетите английски термини *state transition graph (STG)* и *signal flow graph (SFG)*, както и техните съкращения).

Тази организация позволява идентично подържане на различни типове поведение:

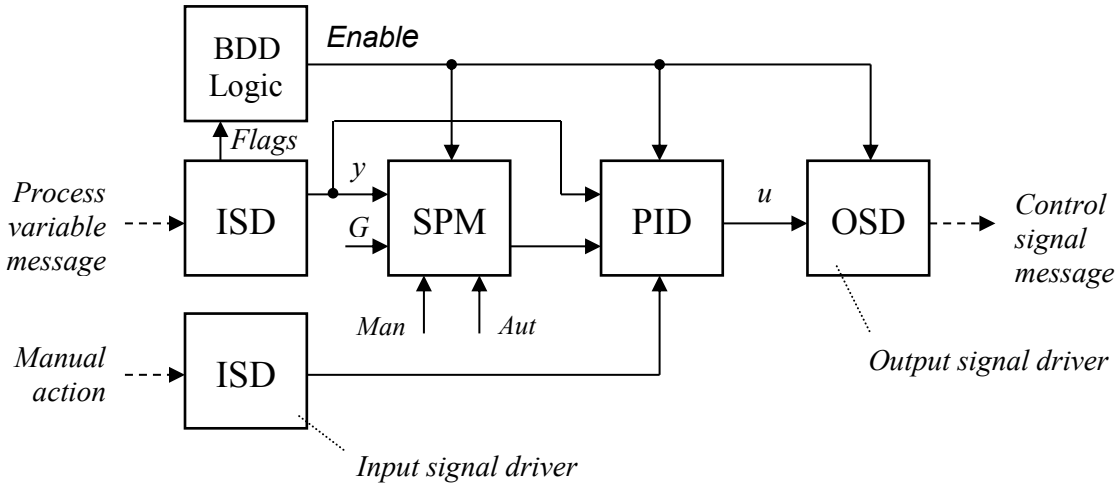
- изчисления (входно→изходни преобразувания), подходящи за реализация на непрекъснати системи за управление;



Фиг. 2.4. Граф на поведението на нишка („функция”)

- автоматно поведение - преход от състояние в състояние и реализация на управляващи функции в отделните състояния, при поява на съответни входни събития;
- комбинация от предишните два типа.

Нишките си обменят информация (комуникират) през защитени записи на базата на протокол за достъп до общи ресурси, осигуряващ защитеност срещу



Фиг. 2.5. Сигнален граф на нишка („функция”)

възникване на събития като взаимна блокировка (deadlock) и неограничена приоритетна инверсия.

Нишките от различни функционални блокове си обменят информация чрез унифицирани комуникационни агенти (*сигнални драйвери)*. Чрез тях се поддържат основните комуникационни схеми на една РКСУ:

- Асинхронен обмен тип „производител-потребител";
- Синхронен обмен, тип „клиент-сървър" или „команда с потвърждение".

Сигналните драйвери комуникират на базата на "контекстно-ориентирано адресиране", т.е. съобщенията съдържат идентификатори на комуникационните променливи. Това е особено важно в смисъла на формулираните изисквания за прозрачност на комуникацията, възможност за реконфигурация и отвореност на системата. В общия случай това води до независимост на логическата от физическата структура на системата.

Предлаганата архитектура се базира на идеята за глобално синхронен – локално асинхронен (GALS - globally asynchronous locally synchronous) [22] подход към разпределените системи. Това на практика предполага използването на схеми за динамично диспечиране в рамките на възлите на мрежата, като транзакциите между ФБ са от тип „само при необходимост" (by need) и достъпът до комуникационната среда е конкурентен динамичен. Това представлява в основата си събитийно-активирана система, но при вече въведената възможност ходът на времето да бъде следен на базата на събития, генерирани от синхроизточник, реализирането на синхронни (time-driven) системи не представлява проблем. Това, от своя страна, улеснява съществено анализа на диспечируемостта на системата за управление – едновременно на процесите и на комуникацията, а от друга страна прави системата адекватна на изискванията за реализация на системи за управление на непрекъснати обекти (отделно от това и за реализация на синхронни управляващи автомати).

Интегрираното диспечиране на процесите и комуникацията се предполага да се базира на следните подходи:

- Разпределени транзакции между събитийно-активирани процеси, при които първият в редицата се активира от периодичен синхросигнал, а следващите от пристигането на комуникационните променливи; [152] [153]
- Разпределени транзакции между периодични процеси, активизирани с фиксирано закъснение (отместване) от началото на комуникационния период.
- Конвейерно изпълнение на транзакциите, известно като „разделение на ресурсите" (*resource decoupling*), комуникиращите си процеси се активират периодично и работят с отделни данни (и ресурси) с период, равен на периода на конвейера. [142].

Най-важната особеност на този модел е компонентното дефиниране на обектите и взаимодействията между тях. ФБ може да се разглеждат като софтуерни аналози на схеми със свръхвисока степен на интеграция. Това позволява, аналогично на конструирането на хардуер, създаването на РКСУ на базата на включването на ФБ с необходимата функционалност и обмяна на сигнали между тях (на база на контекстно-ориентирано адресиране) вместо класическото извикване на методи на обекти, познато от ООП. Тази аналогия отговаря и на споменатото в увода свойство на системите за управление да се базират на процеси, които реализират потокова (неограничена във времето) обработка на неограничен поток от входни данни, а не на старт-стопни действия от тип "извикване на подпрограма". Този общ подход не налага никакви ограничения на структурата на ФБ. ФБ е класически *компонент* в този смисъл, а не обект според ООП терминология. Това позволява и композирането на системата на логически, а не на физически принцип на етапа на нейното

проектиране. Допълнително предимство, произтичащо от компонентното структуриране на РКСУ на базата на ФБ е това, че веднага се постига свойството отвореност (един ФБ се заменя с друг без реструктуриране на системата при изискване само за запазване на сигналния граф). Така моделът ASPS емулира максимално точно структурата и работата на РКСУ по начин, който е близък и разбираем за специалистите по автоматика.

По-надолу ще бъде разгледан формалното описание на модела ASPS, като ще се започне с основния градивен елемент – функционалния модул.

2.3 Модел на представяне на функционален модул

Функционалните модули (ФМ) са основните елементи, изграждащи функционалните блокове. Функционалните модули се разглеждат като стандартни (библиотечни) блокове, които се използват за конфигуриране на нишките на ФБ. Системното действие обикновено се подържа голяма библиотека с функционални модули, включително модули взети от предишни версии на системата *PROCOS* [11]. В подробности наборите модули, техните характеристики и други са разгледани в глава 4.

Функционалните модули са обекти от ниско ниво със стандартна вътрешна структура. ФМ се състои от три компонента: реентрантен код (във форма на подпрограма), известен брой входове и изходи, и информация състояща се от аргументи (параметри). Входно/Изходните връзки и актуалните стойности на параметрите на ФМ са указват по време на конфигурирането на системата (вж. глава 5 “Програмен генератор”).

Типично функционалните модули могат да бъдат използвани за изграждане на *съставни функционални модули* (СФМ), изпълняващи сложни сигнални трансформации, например управление по затворен контур, различни функции обработващи сигнали и други. Съставните функционални модули обикновено са указани чрез *сигнален граф*, който определя безусловно последователността на изпълнение на модулите в СФМ. Функционалните блокове се извикват чрез стандартна процедура – *Execution Sequence Manager (ESM)*, който по същество е интерпретатор на сигналния граф (фиг. 2.4) (виж глава 4).

ФМ представлява атомарен обект по отношение на своето изпълнение. Поради това интерес представлява само моделът му по отношение на данните, върху които оперира. Предложение за реализационна структура на ФМ ще бъде разгледано в гл. 4.

В обобщен вид ФМ има следните типове входни и изходни данни:

$D_{FM} = \{d_b , d_i , d_{li} , d_r , d_{lr}\}$, (2.1)

където

- d_b – логически данни. d_b = {0,1}, т.е. {true, false};
- d_i – 16-битови целочислени данни;
- d_{li} – 32-битови целочислени данни;
- d_r – 32-битови реални данни;
- d_{lr} – 64-битови реални данни.

Освен това ФМ може да бъде източник на събития, вътрешни за ФБ, които са извън потока на данните.

Тъй като източниците на информация на един ФМ могат да бъдат различни – входове на ФБ и данни, породени от други ФМ на ФБ, се налага да могат да бъдат описани формално всички тези източници. Втора особеност на тези източници е фактът, че при някои от тях съответната данна е векторна. Размерът на тези вектори може да бъде функция на величина, задавана при параметризиране на ФМ или фиксиран.

При отчитане на изискването системата за управление да може да бъде препараметризирана (в по-общ случай и преконфигурирана), разглеждането на собствените параметри на модула като входни данни от общ вид елиминира разликите в интерпретацията, свързването с различни източници и в общ смисъл е основа за формалното им описание.

Получава се възможността параметрите на ФМ да постъпват от външен за него източник или да имат константни стойности, записани в някакъв служебен буфер за константи.

Източниците за информация в един ФМ са следните:

- входове на ФБ;
- изходи на функционални модули от функционалния блок;
- от служебна област за съхранение на константи.

Всички типове данни, освен логическите, се организират по естествен и непротиворечив начин във вектори и матрици. Това позволява формалното описание на всяка отделна абстрактна данна D^A да има следния вид:

$D^A = \{ T, S_z \}$, (2.2)

където

T е тип на данната,

S_z е размер на данната.

S_z се представя чрез двойката

$S_z = \{m,n\}$, (2.3)

където

m – брой редове

n – брой колони

в матрицата, представяща данната.

В частен случай $m=n=1$ представя скаларна данна.

Ограничаването на размерността на матриците само до двумерни отговаря на много голяма група алгоритми, прилагани в системите за управление.

На базата на формалното описание на данните може да се изведе формално описание на множеството от входни и изходни сигнали на един ФМ:

$X^A = \{ N, D_{X1}{}^A, \ldots, D_{XN}{}^A\}$, абстрактни входни сигнали;

$Y^A = \{ M, D_{Y1}{}^A, \ldots, D_{YM}{}^A\}$, абстрактни изходни сигнали, (2.4)

където

N е брой на входните сигнали на ФМ,

M е брой на изходните сигнали на ФМ,

$D_{Xi}{}^A$ е описание на характеристиките на i-тия входен сигнал,

$D_{Yj}{}^A$ е описание на характеристиките на j-тия изходен сигнал.

Отчитането на възможните източници на информация на един ФМ и описанието на набора му от входни сигнали позволяват специфициране на реалните (конкретни) източници на информация на един ФМ в конкретна конфигурация по следния начин:

$D^C = \{ S_r, S_z \}$, (2.5)

където

S_r - източник на входен сигнал,

$S_z = \{m,n\}$ - размер на матрицата.

Тук S_r посочва конкретен вход на ФБ, изход на ФМ или поле от зоната за съхранение на константи. Тогава описанието на множеството от входните данни на конкретен ФМ е

$$X^C = \{ D_{X1}^C, \ldots, D_{XN}^C \}, \qquad (2.6)$$

където D_{Xi}^C е конкретното описание на *i*-тата входна данна на ФМ.

Тук не се налага явно посочване на N в представянето на X^C, защото този брой се съдържа в кода на програмата на съответния ФМ.

При описанието на конкретните изходи на един ФМ трябва да се отчита следния допълнителен факт: всеки изходен сигнал, породен от един ФМ, трябва да е достъпен за всички ФМ от ФБ (global data). Въпреки, че при обсъждане на формалния модел на функциониране е неудачно отчитането на реализационни детайли, изискването за глобалност на породените данни (оттам – за множествен достъп до тях) и принципът на работа на фон Нойманови компютър, предполагат като единствена възможна форма на реализиране записването в служебен буфер на всяка породен данна. Така описанието на конкретните изходни сигнали може да се сведе до същия вид, както и конкретните входове:

$$Y^C = \{ D_{Y1}^C, \ldots, D_{YM}^C \}, \qquad (2.7)$$

където D_{Yj}^C е конкретното описание на *j*-тата изходна данна на ФМ. Физически мястото на данната може да бъде единствено в служебен буфер.

Така разгледания модел на входните и изходните данни предлага механизъм на описанието им като абстрактни и конкретни данни. Единственият необхванат тип данни (сигнали) са логическите сигнали. Организацията на паметта на фон Нойманови компютър, организацията на входно/изходната процесна периферия от дискретен тип и множеството от стойностите на логическите сигнали влизат в явно противоречие помежду си.

Противоречията произлизат от следното:

- за числовото представяне на един логически сигнал е достатъчен един бит;
- обичайната форма на организацията на паметта е байтова с 8 бита в един байт;
- най-често срещаната форма на организация на дискретната входно/изходна периферия е байтова или двубайтова, като един байт обхваща 8 входни или изходни канала (напълно независими или организирани в групи от 2 или повече бита).

Това води до проблема, че или за представянето на един логически сигнал трябва физически да се отдели поне един байт, или при използуване на пакетирано представяне (битов вектор), се налага да се използува допълнителна информация кой логически сигнал с кой бит се представя. Тези два начина имат своите предимства и недостатъци, но освен това те пораждат едно допълнително ограничение, а именно – няма естествен начин на групирането им във форма, която да се опише като вектор или матрица, така че абстрактното представяне на логически сигнал да е идентично на това на целочислените или реални данни.

Най-близкото възможно организиране на данните, представящи сигнали от логически тип е пакетирането им байтови (или двубайтови) записи – 8- или 16-битови булеви вектори. Това позволява група от байтове да се интерпретират като едномерен масив от битове, т.е. множествен сигнал.

По този начин логическият сигнал може да се представи чрез същия абстрактен тип данни, както и другите типове:

$$D^A = \{ T, S_z \}$$

$$S_z = \{m,n\},$$

при което

m - броят на битовете, заети от сигнала

n = 1 винаги.

Тук все още няма да бъде разглеждано представянето на конкретен тип D^C за логически сигнали, поради определени машинни зависимости. Варианти на D^C ще бъдат обсъдени по-нататък (виж гл. 4 и 5).

Използваните тук термини “служебен буфер” и “служебна област за константи” ще бъдат разгледани по-нататък, където след обобщение на модела на данните на ФБ ще бъдат заменени с по-подходящи термини.

Към разглеждането на типовете сигнали на ФМ следва да се отбележи следния допълнителен елемент:

При много видове преобразувания на входни сигнали се налага да се пази информация от предишни моменти на работа. Това изисква съхраняване на тази информация между поредните активизации на ФМ. С цел използването на униформно формално описание информацията от предишни извиквания може да се разглежда едновременно като входна и изходна. Това позволява описанието на този род информация по същия начин, както и другите видове входна информация.

Така като обобщение на описанието входните и изходните сигнали на един ФМ включват следното:

- въвеждат се два типа описание на сигналите: абстрактно и конкретно. *Абстрактното* служи за описание на формалния модел на ФМ. *Конкретното* служи за описание на реална инстанция на ФМ и оттам – за връзка между формалния модел на ФБ и формалния модел на ФМ.
  Абстрактното описание на сигнал е
  $D^A = \{ T, S_z \}$
  $S_z = \{m,n\}$.
  Конкретното описание на сигнал е
  $D^C = \{ S_r, S_z \}$,
  $S_z = \{m,n\}$
  S_r = {вход на ФБ, изход на ФМ, служебна зона за параметри, служебна зона за междинни резултати}.
- Параметрите на ФМ се интерпретират като входни сигнали
- Информацията от предишни активизации се разглежда като входни сигнали
- Изходните сигнали на ФМ са глобални на ФБ
- Логическите сигнали се разглеждат като пакетирани (булев вектор). Това позволява абстрактното им описание да е от същия вид, както на останалите типове данни.

2.4 Спецификация на “функционален блок”

Функционалният блок (ФБ) е базовият организационен елемент на системата на предлагания модел. Той е активен програмен обект, който е разположен в някой от възлите на мрежата (в случай на съсредоточена система, всички ФБ са разположени в един възел).

ФБ включва една или повече функции (елементи за самостоятелна активност) на работа. Функциите се реализират чрез нишки на изпълнение. Тук терминът нишка не трябва да се смесва с термина “нишка” (thread) от терминологията на операционните системи. Нишките представляват действията, с които ФБ отговаря на настъпили вътрешни или външни за ФБ събития. Това всъщност е *събитийната* същност на ФБ. В рамките на работата на нишките се изработват изходни сигнали, съответстващи на активизиралото ги събитие. Това представлява *трансформационния* аспект на работата на ФБ. Начините за поддържане на работата на нишките и примерна структура на реално-времевата им база от данни ще бъде разгледана в глави 4 и 5.

Така направеното неформално описание показва отново, че ФБ при такава структура е в състояние да отговори на едно от основните изисквания – да може да служи за основа на изграждане на хетерогенни системи за управление.

Спецификацията на взаимодействията между функционалните блокове и съответните комуникационни протоколи са разгледани в глава 3 .

Формално описание на един функционален блок може да се представи както следва:

$$FB = \{\underline{X}, \underline{Y}, \underline{E}, \underline{P}\}, \quad (2.8)$$

където

$\underline{X}$ - множество на входните сигнали, получавани от ФБ,
$\underline{Y}$ - множество на изходните сигнали, генерирани от ФБ,
$\underline{E}$ - множество на събития, които активират функциите (нишките) на ФБ,
$\underline{P}$ - множество на функциите (нишките), които се активират от $\underline{E}$.

Множеството на входните сигнали в общия случай се състои от множеството на физическите входни сигнали и множеството на входни сигнали, получени от комуникация с други ФБ, т.е.:

$$\underline{X} = X_L \cup X_R, \quad (2.9)$$

където

X_L - (local inputs) физическите входни сигнали на ФБ,
X_R - (remote inputs) входни сигнали, получени от комуникация (от други ФБ).

Множеството на изходните сигнали в общия случай се състои от множеството на физическите изходни сигнали и множеството на изходни сигнали, предадени към други ФБ чрез комуникационната среда, т.е.:

$$\underline{Y} = Y_L \cup Y_R, \quad (2.10)$$

където

Y_L - (local outputs) физическите изходни сигнали на ФБ,
Y_R - (remote outputs) изходни сигнали, предадени към други ФБ през комуникационната среда.

Множеството на активиращите събития $\underline{E}$ може да се опише като:

$$\underline{E} = E_t \cup E_e \cup E_c \cup E_i, \quad (2.11)$$

където

E_t – множество от таймерни събития (събития от синхро-източници)
E_e – множество от външни събития
E_c – множество от комуникационни събития (получаване на данни от комуникация)
E_i – множество от вътрешни за ФБ събития, т.е. събития, генерирани от процеси, включени във ФБ.

Детайлно разглеждане на типовете събития, а така също асоциирането на събития, входни и изходни данни със съответните процеси, включени в един ФБ ще бъде направено в глава 4 и 5 при разглеждането на структурата и реализацията на интерпретиращите среди на ФБ.

Множеството от функциите (нишките, процесите), които са включени в един ФБ и се активират от съответни събития $\underline{E}$, може да се представи по следния начин:

$$P \subseteq \underline{P},$$

при което P се специфицира като разширен автомат на Мур, а именно

$$P = \{ A, X, Y, C, E, R, S \}. \quad (2.12)$$

Тук

A – множество от състоянията на P. Състоянието $a_0 \in A$ се нарича "начално състояние";

X – множество на входните сигнали, обработвани от процеса P. $X \subseteq \underline{X}$;

Y – множество на изходните сигнали, генерирани от процеса P. $Y \subseteq \underline{Y}$;

C – множество от предикати (условия на преходите на автомата).
$C = \{c_k\}, \forall c_k \in C$: $c_k = f_k(x'(t)), x' \subseteq X$ и f_k: $x'(t) \to V, V = \{0,1\}$;

E – множество от събитията, активиращи P, $E \subseteq \underline{E}$;

R – множество от реакциите, асоциирани с процеса и пораждани в отговор на съответните събития E;

S – множество на сигналните трансформации, специфициращи изчислителния изход(и) на процеса.

2.5 Спецификация на сигналните трансформации

Въвеждането на понятие като "сигнална трансформация" в един по същество графов модел на ФБ служи, за да изрази тази страна на ФБ, която е свързана с получаването на изходните сигнали *Y* на блока като резултат от преобразуването на входните сигнали *X* при преход от едно в друго състояние *A* на процесите *P* в резултат на съответните реакции *R*, породени от събитията *E*.

Така за всяка сигнална трансформация S може да се каже:

$$\forall\, y_k \subset Y \leftrightarrow s_k \subset S \text{ и}$$
$$s_k\colon x_k(t) \to y_k(t),\ x_k \subseteq X \qquad (2.12),$$

т.е. всяка сигнална трансформация върху конкретен набор от входни сигнали поражда съответния изходен сигнал.

В частен случай $y_k = const$, когато изходните сигнали на ФБ са фиксирани като резултат от преминаване в определено негово състояние.

От практиката е известно, че s_k рядко е проста едностъпкова процедура, водеща чрез "елементарно" преобразуване от $x_k(t)$ до $y_k(t)$. В общия случай трансформацията е многоетапно преобразуване на входните сигнали до получаването на съответния изходен сигнал, напр.: линеаризация на показанията на датчиците, филтрация, преобразуване до технологични величини, изчисляване на управляващото въздействие, ограничаване на управляващото въздействие до допустими технологични стойности. Това всъщност представя s_k като комбинация от по-"прости" преобразувания. Тук и навсякъде другаде под "прост" ще се разбира "неделим", а няма да се има предвид реалната изчислителна или алгоритмична сложност.

Така

$$s_k = s^k_1 \circ s^k_2 \circ \ldots s^k_m. \quad (2.13)$$

При вече въведения композиционен елемент "функционален модул" (ФМ), s_k може да бъде обяснена като група от ФМ, активизирани в определен ред, които извършват последователни "елементарни" преобразувания върху набора от входни сигнали $x_k(t)$ до получаване на $y_k(t)$ (т.е. съставен функционален модул). Това представяне на сигналните трансформации позволява описание с два типа модели:

- стандартен "control flow" модел – граф, блок-схема и други подобни.
- "data-flow" диаграма.

"Data-flow" моделът е по-разпространен при представянето на сигналните трансформации, тъй като отговаря точно на класическия "аналогов контур на управление". Лесно се вижда, че всяка композиция от вида (2.13), в която няма

цикли (итерации) може да се представи машинно в смисъла на фон Нойманов компютър, дори когато има повече от един независими потока от данни до определена точка в трансформацията, т.е. вариант от типа:

$$\begin{array}{l} x_m \xrightarrow{s_k^m \circ s_k^{m+1} \circ \ldots} \\ x_r \xrightarrow{s_k^r \circ s_k^{r+1} \circ \ldots\ldots} \\ \qquad\qquad s_k^t \end{array} \xrightarrow{s_k^{t+1} \circ \ldots} y_k$$

Тази линейна организация на сигналното преобразуване s_k позволява разглеждането му като обобщено едностъпково преобразуване и оттам – елиминиране на различията при интерпретирането на начина на формиране на булеви сигнали при преминаване на автомата от едно състояние в друго и формирането на не-булеви сигнали при преобразуването на разнородни по характер входни сигнали. Тези две формирания на изходен сигнал вече могат да бъдат разглеждани като изходи на обобщен автомат на Мур, допълнителна обосновка за към което е дадена по-долу.

2.6 Спецификация на системните реакции

Системните реакции отразяват връзката между активиращите събития и породените в отговор на това изходни сигнали. Реакциите $R=\{r_i\}$ могат да бъдат определени като

$$r_i : A \times B \times C \rightarrow Y.$$

Освен това лесно се вижда, че реакцията r_i е съставна функция, т.е. $r_i = o_i \circ p_i$, където p_i е функция на прехода, а o_i е функция на формиране на изходен сигнал.

Функцията на прехода p_i се дефинира като

$$p_i: A \times E \times C \rightarrow A, \qquad (2.14)$$

при което за всяко състояние на процеса *P* е известно множеството от наследниците му, т.е. $\forall a_m \in A$,

$$F_{a_m} = \{a_{n_1}, \ldots, a_{n_r}\}, \qquad (2.15)$$

и съответно, множеството от алтернативни преходи

$$a_m(t_) \,\&\, e_{n_1}^m(t) \,\&\, c_{n_1}^m(x(t)) \rightarrow a_{n_1}(t) \qquad (2.16)$$

.

.

$$a_m(t_) \,\&\, e_{n_r}^m(t) \,\&\, c_{n_r}^m(x(t)) \rightarrow a_{n_r}(t) \cdot$$

Това всъщност представлява графът на състоянията на *P*.

Функцията на изхода o_i задава връзката между генерирания изходен сигнал $y_i \in Y$ в зависимост от състоянието $a_i \in A$, т.е.:

$$o_i: A \rightarrow Y.$$

Това определение на реакцията позволява да я разглеждаме като развита форма на композицията $r_i = o_i \circ p_i$,

$$r_i : A \times E \times C \rightarrow \mathrm{Y} \Leftrightarrow A \times E \times C \xrightarrow{p_i} A \xrightarrow{o_i} Y, \quad (2.17)$$

т.е.

$$\forall r_i \in R: \mathrm{r_i} \leftrightarrow y_i,\ y_i = f(a(\mathrm{t})),$$

т.е. това е форма на автомат на Мур. Тази форма обаче дава възможност за дефиниране на различни типове управляващи машини: дискретни, непрекъснати

и хибридни. Изходните функции се определят като сигнални трансформации, които бяха разгледани по-горе.

Предикатите на преходите в графа на състоянията на процеса се определят като:

$$C = \{c_k\},\ \forall c_k \in C:\ c_k = f_k(x'(\mathrm{t})),\ x' \subseteq X \text{ и } f_k: x'(\mathrm{t}) \to V,\ V = \{0,1\} \qquad (2.18)$$

При тази дефиниция възниква въпроса “Защо се отразяват само външни сигнали като операнди на предикатните функции?” Развитието на този въпрос всъщност би било: “Не може ли величина, представляваща междинен резултат при сигнална трансформация или направо изходен сигнал y_k да бъдат операнди на предикатни изрази?”.

Разглеждането на този проблем е изключително важно, тъй като текущата практика разглежда подобна възможност като естествена за управляващите машини. Ограничението, при което ще бъде направено разглеждането е само едно: Трябва да се запази характерът на управляващата машина като разширен автомат на Мур.

Разглеждането на израза (2.18) показва, че стойността на предиката c_k може да е резултат от трансформация върху сигнала x_r, което означава че е възможно формирането на предикати върху резултати от сигнални трансформации на входни величини, напр.: линеаризация и проверка за валидност на технологични и аварийни граници и градиентен контрол върху входния сигнал x_r.

При възникване на необходимост от формиране на предикати с операнди, които са изходни резултати от сигнални трансформации е достатъчно множеството на входните сигнали да бъде разширено с множеството на необходимите изходни сигнали. Това формира сигнална връзка тип “обратна връзка”, а от друга страна не променя характера на управляващата машина като автомат на Мур. Трябва да се отбележи, че възниква ограничението, че стойността на изходния сигнал, който формира предиката се разглежда при следващото активизиране на процеса на p_j. Това обаче е естествено, тъй като обратното означава, че може сигнал y_k, породен в състоянието a_n да предполага установяването на процеса при текущото му активизиране в състояние $a_l \neq a_n$, което е явно противоречие.

В разглежданията до тук бяха направени формални описания и спецификации на елементите на един ФБ.

ФБ се разглежда двустранно – като поведение (реактивен аспект, респ. формално описание на това поведение) и като система за преобразуване на данни (трансформационен аспект, респ. формално описание на тези данни като поток и параметри).

Входните сигнали на един ФБ се представят с абстрактен и конкретен модел, както следва:

$$X^A = \{ N, D_{X1}{}^A, \ldots, D_{XN}{}^A\}, \text{ абстрактни входни сигнали;} \qquad (2.19)$$

където

N - брой на входните сигнали на ФБ,

$D_{Xi}{}^A$ - описание на характеристиките на *i*-тия вход (абстрактен тип на входна данна).

$$X^C = \{ D_{X1}{}^C, \ldots, D_{XN}{}^C\}, \quad \text{конкретни входни сигнали; (2.20)}$$

където

$D_{Xi}{}^C$ е конкретното описание на *i*-тия вход на ФБ.

Изходните сигнали на един ФБ се представят по идентичен начин с абстрактен и конкретен модел, а именно:

$Y^A = \{ M, D_{Y1}{}^A, ..., D_{YM}{}^A \}$, абстрактни изходни сигнали,

където

M е брой на изходните сигнали на ФБ,

$D_{Yj}{}^A$ е описание на характеристиките на *j*-тия изход (абстрактен тип на изходна данна).

Описанието на конкретните изходни сигнали се свежда до същия вид, както и конкретните входове:

$Y^C = \{ D_{Y1}{}^C, \ldots, D_{YM}{}^C \}$, (2.21)

където $D_{Yj}{}^C$ е конкретното описание на *j*-тия изход на ФБ.

Конкретните типове данни (входове и изходи) се представят по следния начин:

$D^C = \{ S_r, S_z \}$, (2.22)

където

S_r – източник/приемник на сигнал,

$S_z = \{m,n\}$ - размер на данните, представляващи сигнала (брой редове и колони на матрица).

2.7 Изисквания към средата за проектиране на РКСУ и към Програмния генератор като част от нея

- Възприема спецификацията на системата за управление.
- Възприема физическата структура на системата и трансформира спецификацията до физическата конструкция.
- Поражда таблици с параметри за различните елементи на средата (framework).
- Прави тестове на конкретни физически елементи от системата, с цел установяване на времената на изпълнение на програми от възлите и комуникационните подсистеми.
- На базата на (2,3,4) прави анализ на диспечируемостта на системата.
- Определя структурите с данни, до които има конкурентен достъп. Определя тези данни като ресурси и включва информация за това към таблиците на средата с цел автоматична защита на общите данни в “lock-unlock” от елементите на средата.
- Определя данните с конкурентен достъп между елементите на средата с поведение на задача (task) на ОС и тези с поведение на ISR (interrupt service routines), формира информация за това за средата с цел защита на конкурентните данни в участъци със забранени прекъсвания.
- Спецификация на системата за управление представлява набор от функционални блокове (ФБ). Всеки ФБ се представя с разширен автомат на Мур. Всяко състояние може да включва трансформация на сигнали, която представлява композиция от по-прости трансформации.
- ПГ възприема изискванията за тактове на дискретизация, максимално време за изпълнение на ФБ, на отделните сигнални трансформации
- ПГ поражда физическа структура на системата със следните форми на активизиране на процесите
  - периодично
  - периодично с отложено активизиране
  - периодично за първия и по събитие за следващите от трансформационната верига
  - периодично в последователни тактове за задачите от една трансформационна верига (конвейерно активиране).
- ПГ определя за всеки възел от физическата мрежа колко пъти ще се заяви като отделна задача интерпретатора на графа. Определя приоритетите на тези заявки, ΔT и др. свързани с това елементи на конфигурацията на ОСРВ.

Резултати:

- Предложен е модел за формално описание на РСУ, който позволява описанието на хибридна разпределена (и съсредоточена) КСУ.
- Предложената форма на разширен автомат на Мур представлява разширение на сега съществуващите описания, които са по принцип само еднотипни.
- Елиминиран е проблемът с наличието на доминиращ математически апарат – еднакво се представят както непрекъснати, така и дискретни системи.
- Формулирани са основните изисквания към елементите на системата за управление и програмния генератор, необходими за реализирането на предложения формален модел.

Гл.3. Структура на комуникационната подсистема и общи изисквания към ОСРВ, отговарящи на формалния модел на РКСУ

В глава 3 се предлага формален модел за описание на комуникационните транзакции, подходящи за реализиране в РКСУ. Формулират се изискванията и конструкцията на надеждна и предсказуема ОСРВ (safe and predictable). На базата на предложения формален модел и формулираните изисквания към ОСРВ се предлага комуникационен протокол, който е адекватен на предложения в гл. 2 формален модел на РКСУ.

3.1 Анализ на структурата и изискванията към ОСРВ за РКСУ

Функционалните нишки (activity threads) на функционалните блокове трябва да бъдат реализирани програмно с използуване на адекватна програмна технология. Необходимо е тази реализация да отговаря на изискванията за безопасно и предсказуемо поведение (safe and predictable behaviour). От гледна точка на сегашното поколение операционни системи (ОС) това са процеси (задачи) или нишки на операционната система. Различията се базират на това, дали ОС поддържа многопроцесна еднонишкова среда или многопроцесна многонишкова среда (multitask single-threaded / multitask multithreaded). Повечето ОС за управляващи системи поддържат главно първия вариант, защото той има по-малки изисквания към сложността на ядрото на системата.

Използуването на гъвкава и едновременно с това предсказуема операционна среда за РКСУ е един от най-големите проблеми, които трябва да бъдат разрешени. Необходимо е освен това да се поддържат необходимите системни средства, за да бъде реализиран изчислителния модел на СУПП (ASPS). Специално е необходимо да може да се поддържат взаимодействията между компонентите. Тази работа няма за цел предложение на нова архитектура на ОСРВ, отговаряща на съответните изисквания. Тук ще бъде направен анализ на главните функционални изисквания и ще бъде разгледана възможността известно ядро на ОРСВ след съответни модификации да отговори на тях.

Основните изисквания към ядро на ОСРВ за нуждите на РКСУ са обект на множество изследвания [91][150]. Начални изисквания:

- Малко собствено време на работа
- Малък обем памет, заемана от ОСРВ.

Към това се добавят следните главни изисквания:

- Интегрирано диспечиране на задачи (процеси) и ресурси при използуване на подходящи синхронизационни протоколи.
- Константно време за работа на системните примитиви.

Тези проблеми са нерешими при използуване на стандартните системни структури с данни, като напр. свързани списъци, използувани за реализиране на опашките на готовите и изчакващи задачи и пр. Използуването на тези структури води до поява на недетерминизъм (от гледна точка на необходимо време за работа) в работата на ядрото на ОСРВ, т.нар. „*kernel jitter*" . Анализът на проблема [48][90] показва, че операционната сложност е $O(n^2)$, което е недопустимо при управление на бързи процеси.

Този проблем е елиминиран в разработката на ядро на ОСРВ, предназначено специално за системи за управление с предсказуемо поведение. Това е ОСРВ HARTEX [71]. На фиг. 3-1 е показана общата организация на ОСРВ HARTEX.Тази разработка не е обект на настоящата работа. Тук ще бъдат

обсъдени само въпросите, свързани с конструирането и реализацията на комуникационната подсистема.

Фиг. 3-1. Организация на ОСРВ HARTEX

Проблемът с константното време за работа на системните примитиви е елиминиран чрез използване на Булеви вектори за реализирането на системните опашки. Всъщност, както се вижда, системните опашки не са елиминирани като структура с данни, но са представени с друг вид информационен носител (битови стрингове), който при определени ограничения в дължината позволява множествена обработка на всички негови елементи. Това води до елиминиране на флуктуациите във времето за работа на ядрото на ОСРВ.

Основните характеристики на ОСРВ HARTEX са следните:

- Модулна архитектура, позволяваща реконфигурация и мащабиране на ядрото;
- Интегрирано диспечиране на ресурси и процеси с използване на усъвършенстван протокол за наследяване на приоритетните тавани, който осигурява ограничено време за транзитивно блокиране и контекстно-независимо определяне на приоритетните тавани;.
- Високопроизводителна подсистема за отчитане на времето с подходящ набор от примитиви, осигуряващи системните услуги;
- Базирани на булеви вектори системни структури с данни, осигуряващи много малко и константно време за изпълнение на системните примитиви, независещо от броя на обслужваните процеси или ресурси;
- Събитийна сигнализация чрез семафори, базирани на Булеви вектори, осигуряващи едновременно оповестяване на настъпило събитие, независимо от броя на очакващите го процеси;

• Интегриран комуникационен протокол, поддържащ контекстно-ориентирано адресиране на съобщенията.

Анализът на възможностите и услугите, предлагани от ОСРВ HARTEX показва, че за нуждите на РКСУ, състоящи се до значителна степен от малки и средни по големина управляващи компютри, могат да бъдат предложени допълнения, известни от предишни по-малки ОСРВ, а именно:

• Дефиниране на свързани групи от задачи (последователности от задачи), диспечирани по идентичен начин.

• Диспечиране на принципа на фиксираните приоритети (fixed-priority scheduling). При наличие на повече от една готови задачи с еднакъв приоритет, циклично диспечиране с еднакъв период на обслужване, т.е. преминаване от *многозадачно еднонишково* диспечиране към *многозадачно многонишково* диспечиране.

• Интегрирано и идентично управление на времевите и асинхронни събития (integrated timing and event management).

Външните събития се генерират като резултат от апаратни прекъсвания, които се обработват от Диспечера на събитията (ДС) (*Event Manager - EM)*. Последният се състои от определен брой обработчици на събития (event handlers) – по един за всяко събитие. Те, от своя страна, се активират от програми за обработки на прекъсвания (ПОП), които са част от т.нар. “изолационен слой - слой за връзка с апаратурата” (*Hardware Adaptation Layer - HAL*). Събитийните броячи са вградени в основните механизми за обработка на събития, т.е. заявка за обработка на събитие се генерира тогава, когато се получи определен брой апаратни заявки. При този случай една или повече задачи се сигнализират или заявяват за изпълнение. Друга възможност е разпращане на съобщения до една или повече задачи (broadcast messaging).

По същия начин диспечерът на реалното време (ДРВ) (*Time Manager - TM*), генерира събития, отразяващи хода на времето и сигнализира или заявява за изпълнение съответните задачи, когато свързаните с тях времеви интервали изтекат. ДРВ поддържа набор от услуги, които осигуряват безопасно и надеждно изпълнение и взаимодействие между приложните задачи (процеси) в среда за твърдо реално време, базирани за абсолютен таймер с висока разрешаваща способност.

И двата диспечера, ДС и ДРВ, използват механизма на събитийно потвърждение и глобално оповестяване (event notification, message broadcast), осигурени от *HARTEX*.

Изпълнението на задачите се контролира от основната подсистема на ядрото – Диспечерът на задачите (ДЗ) (*Task Manager*). Той поддържа следните основни системни функции:

• Динамично диспечиране на задачи и последователности от задачи;

• Интегрирано диспечиране на задачи и споделени ресурси на базата на „Протокол с условно наследяване на приоритетните тавани”;

• Управление на състоянието на задачите чрез различни системни примитиви.

Взаимодействието между задачите е два типа: *синхронно и асинхронно*.

• Асинхронната комуникация е предназначена за време-критична комуникация, т.е. размяна на променливи на състоянието между динамични подсистеми. Основната схема на взаимодействие е „един към много”.

- Синхронната комуникация поддържа семантично събитийни съобщения. Предназначена е главно за т.нар. „разширени (съставни)” задачи, които могат да се блокират, докато очакват събитие, или в този случай, пристигане на събитийно съобщение. Основната схема на взаимодействие е „един към много”.По същия начин се използват и за предаване на команди при вертикална комуникация между подсистемите на РКСУ. Основната схема на взаимодействие в този случай е „един към един”.

Съобразно с това се дефинират и три типа комуникационни променливи – разпространявани (broadcast), изисквани (requested) и потвърждавани (confirmed).

3.2 Формална спецификация на взаимодействията между подсистемите на РКСУ

Съвременните методи за проектиране на РКСУ се основават на обектния подход, при който РКСУ се изгражда като композиция на взаимодействащи си активни обекти - функционални блокове (ФБ).

В гл. 2 и 4 е разгледан формален модел на ФБ, при което ФБ се формира като композиция от хибридни автомати, а всеки автомат описва поведението на отделна функция (activity), капсулирана във ФБ.

Функциите формират реакциите на определени събития в контекста на съответния функционален блок. В разпределените системи за управление подобни реакции формират изчислителните фази на сложни реакции, реализирани като разпределени транзакции.

В този смисъл взаимодействията между ФБ се свеждат до взаимодействия между съответните функции, които на свой ред се осъществяват чрез обмен на съобщения, адресирани по съдържание.

При тази постановка на задачата спецификацията на взаимодействията между ФБ съдържа два аспекта:

- статична дефиниция на комуникационните връзки между ФБ, респективно на съобщенията (сигналите) обменяни между тях, което е равносилно на изграждане на графа на комуникациите на РКСУ;
- спецификация на взаимодействията в динамика, т.е. спецификация на разпределените транзакции и техните атрибути: активиращи събития, съставни изчислителни и комуникационни процеси, параметри на процесите (напр. период, отместване, времеви граници и т.н.);

Така формулираните подзадачи на спецификацията са разгледани в по-надолу.

3.2.1 Спецификация на връзките между ФБ

Структурата на РКСУ се задава с графа на комуникациите:

$$G = \{B, Y, R\}, \qquad (3.1)$$

където

B е множеството на ФБ; $B = \{b_i\}$

$$B = \bigcup_{i=1}^{k} B_i \ , \qquad (3.2)$$

където B_i са подмножества от ФБ, асоциирани с отделни възли на системата; $B_i \leftrightarrow Node_i$;

Y – множество на комуникационните променливи (сигнали), генерирани от ФБ.

$$Y = \bigcup_{(B)} Y_i \ , \qquad (3.3)$$

където Y_i е подмножество от комуникационни променливи, генерирани от функционален блок b_i,

R – множество на връзките (обменяните комуникационни променливи) между отделните ФБ.

$$R = \{ Rb_1, Rb_1, \ldots\ldots\ldots, Rb_n \} \ , \qquad (3.4)$$

където $Rb_i : \ b_i \rightarrow B$ – изображение, дефиниращо подмножеството от ФБ, които получават съобщения (сигнали) от b_i,

$$Rb_i = \bigcup_{(Y_i)} Rb_j (y_j / y_j \in Y_i) \, , \qquad (3.5)$$

където $Rb_j(y_j)$ дефинира подмножеството от приемници, получаващи сигнала y_j, $y_j \in Y_i$, генериран от ФБ b_i.

Алтернативна дефиниция

$$\boldsymbol{G = \{ B, X, S \}}, \qquad (3.6)$$

където

B е множеството на ФБ; $B = \{ b_i \}$;

X е множеството на комуникационните променливи, получавани от ФБ;

$$X = \bigcup_{(B)} X_i \ , \qquad (3.7)$$

където X_i е подмножество от комуникационни променливи, получавани от функционален блок b_i,

S – множество на връзките (обменяните комуникационни променливи) между отделните ФБ,

$$S = \{ Sb_1, Sb_2, \ldots\ldots\ldots, Sb_n \}, \qquad (3.8)$$

където $Sb_i : b_i \rightarrow B$ изображение, дефиниращо подмножество от ФБ, от които b_i получава сигнали. Очевидно $Sb_i = F^{-1} b_i$.

$$Sb_i = \bigcup_{(X_i)} Sb_j (x_j / x_j \in X_i) \, , \qquad (3.9)$$

където $Sb_i(x_j)$ дефинира в общия случай подмножество от функционални блокове източници, генериращи сигнала y_i. На практика $|Sb_i(x_j)| = 1$, т.е. въпросните подмножества се редуцират до единични ФБ, вследствие изискването за еднозначност на ФБ – източник , респ. присвояването на уникални атрибути (адреси, идентификатори и пр.) на съответните съобщения. (Това води до възможност за по-компактен аналитичен запис.)

Двете дефиниции са по същество еквивалентни, тъй като те задават еднозначно графа на комуникациите под формата на ориентиран сигнален граф, чиито дъги са маркирани с идентификаторите на обменяните сигнали (съобщения). От този модел следва естествено комуникационния протокол да поддържа неявно адресиране според съдържанието (идентификаторите) на обменяните съобщения.

Горният граф може да бъде дефиниран явно – под формата на графична конструкция или на еквивалентен аналитичен запис. Същият може да бъде дефиниран и неявно – чрез задаване на подмножества на генерираните и получавани сигнали – Y_i и X_i , за всеки един от ФБ $b_i \in B$.

3.2.2 Спецификация на взаимодействията между ФБ

Взаимодействията между ФБ се разглеждат в контекста на сложни системни реакции, реализирани като разпределени транзакции. Този вид реакции могат да се разгледат като обобщение на атомарните (прости) реакции.

3.2.2.1 Генериране на сложни реакции чрез управляващи последователности (транзакции)

Разглежданият модел третира реакциите от вида:

$$e \rightarrow r_i \,,\ r_i \in R(\tau_m), \qquad (3.10)$$

където $R(\tau_m)$ е множеството от реакции, пораждани от задачата τ_m. При това реакцията r_i се представя чрез управляващ сигнал, примерно y_i, $y_i = s_i(x_i)$, генериран в съответното състояние (режим) на многорежимната задача τ_m.

Горният модел може да бъде обобщен с оглед реализацията на йерархия от хибридни автомати, при което в отделните състояния на базовия автомат могат да бъдат пораждани дъщерни задачи, които на свой ред се описват с хибридни автомати от второ ниво и т.н. Това дава възможност за дефиниране на реакции, пораждани от дъщерни задачи, заявявани в текущото състояние на базовата задача:

$$e \rightarrow \tau_{n/\mathrm{n}\neq\mathrm{m}} \Rightarrow e \rightarrow r_j \,,\ r_j \in R(\tau_n).$$

В общия случай в дадено състояние могат да бъдат заявени подмножество от задачи, например $\tau_1, \tau_2 \ldots\ldots \tau_k$, чрез които се реализират сложни реакции от вида:

$$e \rightarrow (\ \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \cdots \rightarrow \tau_k\ ) \qquad (3.11)$$

$$e \rightarrow (\ \tau_1 \big|_{o_2} \tau_2 \big|_{o_3} \tau_3 \rightarrow \cdots \rightarrow \big|_{o_k} \tau_k\ ) \qquad (3.12)$$

$$e \rightarrow (\ \tau_1 \parallel \tau_2 \parallel \tau_3 \cdots \parallel \tau_k\ ) \qquad (3.13)$$

$$e \rightarrow ((\tau_1^1 \rightarrow \tau_2^1 \rightarrow \cdots) \| (\tau_1^2 \rightarrow \tau_2^2 \rightarrow \cdots) \| \cdots). \qquad (3.14)$$

Първият случай (3.11) съответства на последователна композиция на задачи, т.е. на подмножество от задачи, намиращи се в отношение "предшестване-следване", например:

$$\tau_1(X) \rightarrow \tau_2(\tilde{y}) \rightarrow \tau_3(y),$$

където

$\tau_1(X)$ – входна задача, която снема входните данни *X,* необходими за обработка на събитието, като го подлага на първична обработка.

$\tau_2(\tilde{y})$ – управляваща задача, реализираща същинската обработка на събитието, респективно генериране на реакцията $\tilde{y}$.

$\tau_3(y)$ – изходна задача, извършваща пост-обработка и извеждане на управляващото въздействие *y*.

Така изброените задачи формират последователност от изпълнителни етапи, означавани с термина "транзакция". Транзакциите могат да бъдат периодични или апериодични (спорадични), в зависимост от вида на активиращите ги събития.

Вторият запис (3.12) съответства на последователности на задачи с отместване (*offsets*), при което времената на заявяване на задачите се отнасят към времето на постъпване на събитието $@e$ както следва:

$$@\,\tau_1 = @e + o_1 = @e \qquad (\text{т.е. } o_1 = 0)$$

$$@\,\tau_2 = @e + o_2$$

$$\vdots$$

$$@\,\tau_k = @e + o_k$$

Така описания подход се използва за елиминиране на т.нар. *jitter* ефект (*release jitter, output jitter*), при реализиране на сложни периодични транзакции и пр.

Записът (3.13) описва друга конструкция, която може да се характеризира като паралелна композиция от задачи, т.е. задачи, чието изпълнение не може да бъде сериализирано. Подобна ситуация възниква при реализация на сложни системи, напр. *multi-rate* системи за управление, характеризиращи се със сложни периодични транзакции, в които съставните задачи се изпълняват с различни честоти на активиране.

Накрая записът (3.14) илюстрира още една възможна ситуация, при която реакциите са реализира чрез последователно-паралелна композиция от задачи, изпълнявани в дадено състояние (режим) на базовия хибриден автомат.

3.2.2.2 Организация на разпределени транзакции в РКСУ.

В РКСУ транзакцията се реализира като последователност от изчислителни и комуникационни фази (задачи), изпълнявани върху физически ресурси на компютърната мрежа, от вида :

$$e(k) \rightarrow (\tau_1 \rightarrow c_{12} \rightarrow \tau_2 \rightarrow c_{23} \rightarrow \ldots \rightarrow c_{k-1,k} \rightarrow \tau_m)^{(k)},$$

където $e(k)$ е k-тата поява на активиращото събитие. Реакцията се формира от изчислителните задачи $\tau_1, \tau_2 \ldots \tau_m$, изпълнявани в отделните възли на мрежата, а взаимодействията между тях се осъществяват чрез предаване на събития, съдържащи информация, необходима за изпълнението на поредния изчислителен етап на транзакцията. Процесите на комуникация могат да се разглеждат като своеобразни задачи - c_{12}, c_{23} и т.н., изпълнявани в комуникационната среда на мрежата.

Специален интерес представляват разпределените транзакции, активирани от периодични постъпващи таймерни събития от вида $e = \uparrow(kT)$. При този вид взаимодействия се създават предпоставки за реализиране на специфични режими на функциониране, в т.ч.:

- Разпределени транзакции с фиксирани отмествания на фазите, с оглед елиминиране на ефектите *release jitter* и *output jitter*, а така също и създаване на предпоставки за прецизно оценяване на диспечируемостта на задачите и транзакциите.
- Паралелни и последователно паралелни транзакции.

Последните дават възможност за използване на пространствен и/или времеви паралелизъм с оглед на реализиране на по-сложни (конвейерни) режими, характеризиращи се със съвместяване на фазите на някаква поредица от периодично активирани транзакции.

Горните възможности могат да бъдат илюстрирани за случая на разпределен контур за управление, съставен от интелигентен датчик (задача τ_1), регулатор (задача τ_2) и интелигентен изпълнителен механизъм (задача τ_1). На фиг. 3.2 са показани някои от възможните режими на функциониране на РКСУ, както следва:

- Последователна транзакция

  $\uparrow(kT) \rightarrow (\tau_1 \rightarrow c_{12} \rightarrow \tau_2 \rightarrow c_{23} \rightarrow \tau_3)^{(k)}$ - фиг.3.2.а

- Последователна транзакция с отмествания

  $\uparrow(kT) \rightarrow (\tau_1 \rightarrow c_{12})^{(k)}\Big|_{o_2} (\tau_2 \rightarrow c_{23})^{(k)}\Big|_{o_3} \tau_3^{(k)}$ - фиг.3.2.б

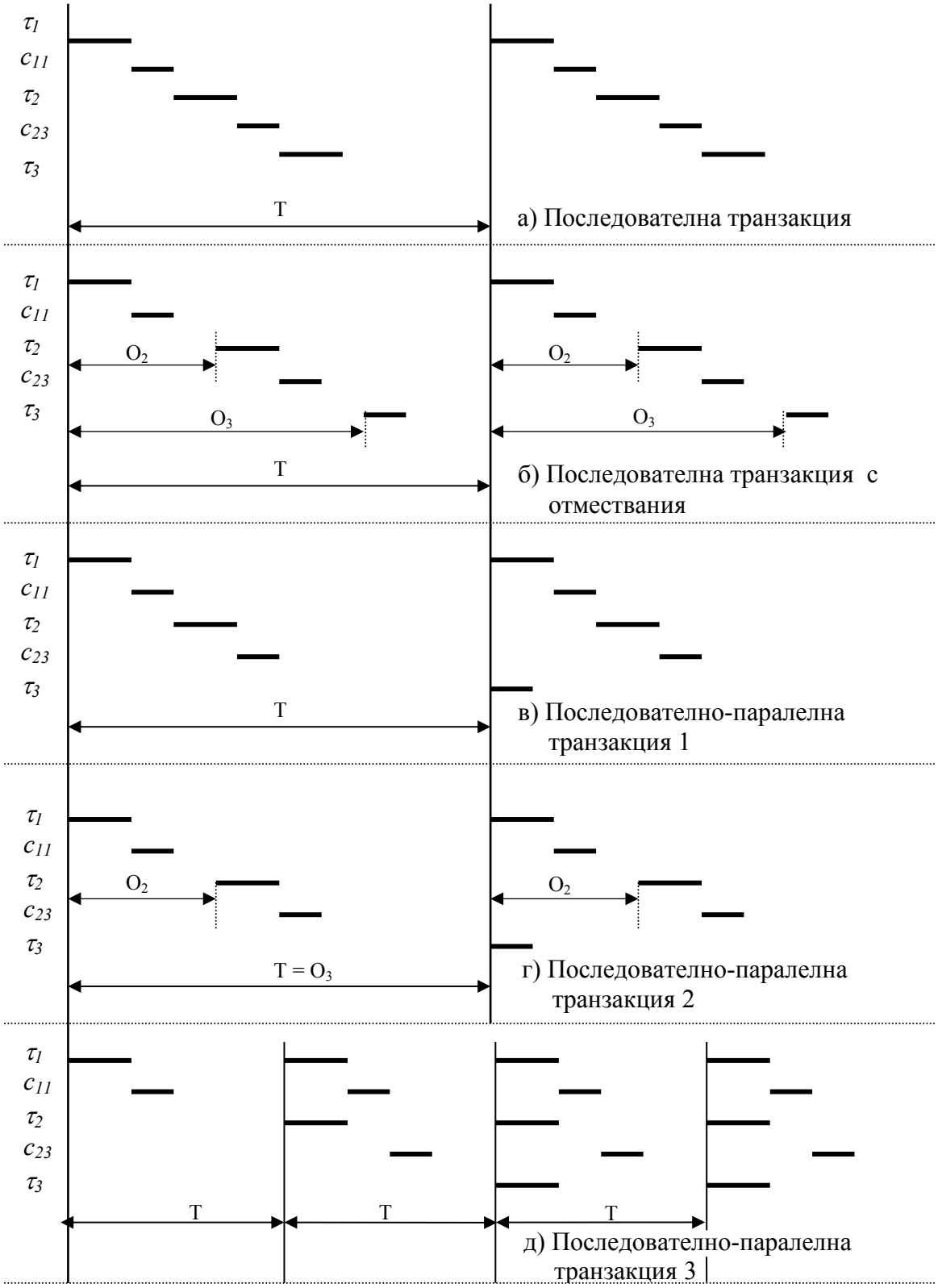
- Последователно-паралелна транзакция

  $\uparrow(kT) \rightarrow (\tau_1 \rightarrow c_{12} \rightarrow \tau_2 \rightarrow c_{23})^{(k)} \| \tau_3^{(k-1)}$ - фиг.3.2.в

  $\uparrow(kT) \rightarrow (\tau_1 \rightarrow c_{12})^{(k)}\Big|_{o_2} (\tau_2 \rightarrow c_{23})^{(k)} \| \tau_3^{(k-1)}$ - фиг.3.2.г

  $\uparrow(kT) \rightarrow (\tau_1 \rightarrow c_{12})^{(k)} \| (\tau_2 \rightarrow c_{23})^{(k)} \| \tau_3^{(k-2)}$ - фиг.3.2.д.

Приведените по-горе примери илюстрират хоризонтални взаимодействия тип “производител-потребител” които обикновено са критични по време, при което времевите граници на транзакциите, респективно на отделните фази могат да бъдат изведени директно от съответните им времедиаграми (фиг. 3.2).



Фиг. 3.2. Варианти на реализация на транзакции

В разглеждания модел на РСУ се предвижда възможност за спомагателни взаимодействия от вида “клиент-сървър” (заявка-отговор, респ. “съобщение-потвърждение”). Подобно взаимодействие може да бъде описано формално с транзакция от вида:

$$e \rightarrow (\tau_1' \rightarrow c_{12} \rightarrow \tau_2 \rightarrow c_{21} \rightarrow \tau_1''),$$

където τ_1' е означението на задачата-клиент респ. на началната и част, в която се формира заявката към сървъра τ_2, а τ_1'' - втората и част, която се изпълнява след връщане на отговора. Подобна транзакция по правило се активира от апериодични външно събитие в рамките на вертикални взаимодействия, които обикновено не са критични по време (например взаимодействия между операторска станция и контролер).

В настоящия подход е направено формално описание на взаимодействията между подсистемите на РСУ в два взаимно свързани аспекта:

- Спецификация на комуникационните вериги между подсистемите на РСУ – т.нар. функционални блокове в рамките на модел, който предполага обмен на съобщения адресирани по съдържание
- Спецификация на взаимодействията в динамика, в контекста на различни видове разпределени транзакции, реализирани като последователни, паралелни или последователно-паралелни композиции от изчислителни и комуникационни фази (задачи).

Въз основа на този формален модел на взаимодействията е разработен комуникационния протокол на системата. В следващата точка се разглежда общата организация на протокола, като се разграничават отделните съставящи нива и поднива.

3.3 Комуникационен протокол за СУПП (ASPS)

Функционирането на РКСУ обикновено се базира на някой от индустриалните мрежови протоколи. Най-често те са структурирани в четири слоя (нива): потребителски, приложен, свързващ и физически (*user, application, data-link* и *physical)*. Предложената архитектура СУПП дефинира основно най-горните два слоя на протокола: потребителският, който специфицира обектния модел на СУПП и приложният, дефиниращ механизмите на взаимодействие между функционалните блокове и съставящите ги нишки. Според това той се декомпозира на два подслоя: слой за обмяна на сигнали и слой за взаимодействие между нишките (фиг.3.3).

Основен комуникационен протокол

(*Basic Communication Protocol* ***)***

- Потребителско ниво: ASPS Обектен модел
  (*User Layer: ASPS Object Model*)
- Приложно ново: Взаимодействие между функционалните блокове
  (*Application Layer: Function Block Interaction*)
  - Обмен на сигнали: Дефиниция на сигнални драйвери
    (*Signal Exchange: Definition of Signal Drivers*)
  - Взаимодействие между процеси(нишки): ***Комуникационен протокол на HARTEX***
    (*Thread Interaction:* ***HARTEX communication protocol)***
    - Обмен на съобщения адресирани по съдържание
      (*Message Exchange via Content-Oriented Message Addressing*)
      - Асинхронна комуникация–обмен на променливи на състоянието
        (*Asynchronous State-Message Communication*)
      - Синхронна комуникация
        (*Synchronous Communication*)
        - Обмен на командни съобщения
          (*Command Message Communication*)
        - Клиент-сървър взаимодействие
          (*Client-Server Interaction*)
    - Оповестяване на събития с векторни семафори
      (*Event Notification via Vector Semaphores*)
      - Неблокиращо оповестяване на събития
        (*Non-Blocking Event Notification*)
      - Блокиращо оповестяване на събития
        (*Blocking Event Notification*)
- Канално ниво
  (*Data Link Layer*)
- Физическо ниво
  (*Physical Layer*)

Фиг. 3. 3. Нива на комуникационния протокол

3.3.1 Слой за обмяна на сигнали

Функционалните блокове си комуникират чрез обмяна на *сигнали*. Те представляват наименовани съобщения. За осъществяване на комуникацията ФБ използват стандартни комуникационни обекти – *сигнални драйвери*, дефинирани в този слой. Главната идея на сигналните драйвери е осигуряване на прозрачност на съобщенията и скриване на данните (data hiding). Изходният сигнален драйвер има задачата да композира съобщението в изходен буфер (прозрачно за приложението) и да го изпрати, а входният сигнален драйвер получава съобщението от буфера-приемник и го декомпозира в компоненти на локалната база данни.

За тази цел сигналният драйвер използва *дескриптор на комуникационната променлива*, който се съдържа в структурата с данни на обекта. В настоящата система сигналните драйвери са реализирани като клас от функционални модули, известни също като вх./изх. драйвери за отдалечена комуникация (вж. Приложение 1). Сигналните драйвери могат да бъдат извиквани от приложните задачи, т.е. нишките на ФБ. Алтернативно те могат да бъдат извиквани от вх./изх. задачи, изпълнявани при осъществяване на реалновремеви транзакции.

Сигналните драйвери могат да бъдат параметризирани така, че да отчитат специфичните изисквания на комуникационните транзакции, например асинхронно глобално оповестяване (broadcast), клиент-сървър взаимодействия и т.н. Според това те извикват съответните комуникационни примитиви на *HARTEX* , които се използват за трансфер на съобщението от изходния към буфера-приемник на взаимодействащите блокове [11][12][14][71].

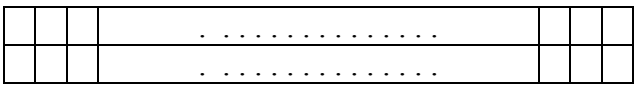
3.3.2 Слой на взаимодействие между нишки

Комуникационните примитиви на *HARTEX* съставляват слоя на взаимодействие между нишки. Последният е съставен от два поднива – слоя за обмяна на съобщения и по-долното ниво – слоя за оповестяване на събития (фиг. 3.2).

Поднивото за оповестяване на събития е използван качествено новият механизъм с векторни семафори и семафорни сигнализиращи операции, в рамките на два типа взаимодействия – блокиращо и неблокиращо оповестяване на събития [14][71]. Оповестяването на събития се реализира за *константно* време (независимо от броя на оповестяваните задачи) като резултат на използването на булевите операции, което е присъщо на векторните семафори.

3.3.2.1 Блокиращо оповестяване на събития

Блокиращото оповестяване на събития е изпълнено от примитивата *signal_and_resume (semaphore, tasks)*, (фиг. 3.5), която се използва за сигнализиране на една или няколко задачи, специфицирани от *tasks* аргументния вектор, за настъпване на събитие. Побитовите операции правят възможно едновременното вдигане на множество флагове за събитие във векторния семафор и едновременното “активиране” на задачите, чакащи на семафора. Горната операция се комбинира с блокиращата *wait* примитива, (фиг. 3.6).

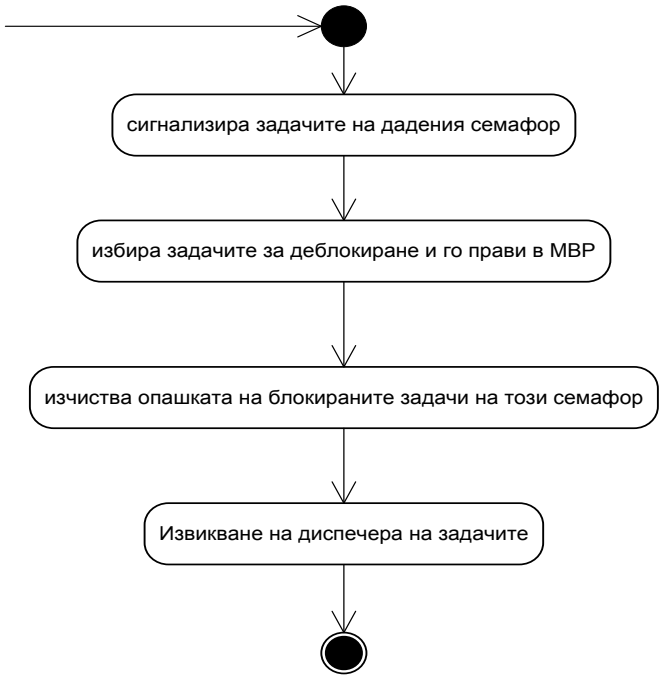


Семафорни флагове(flags)

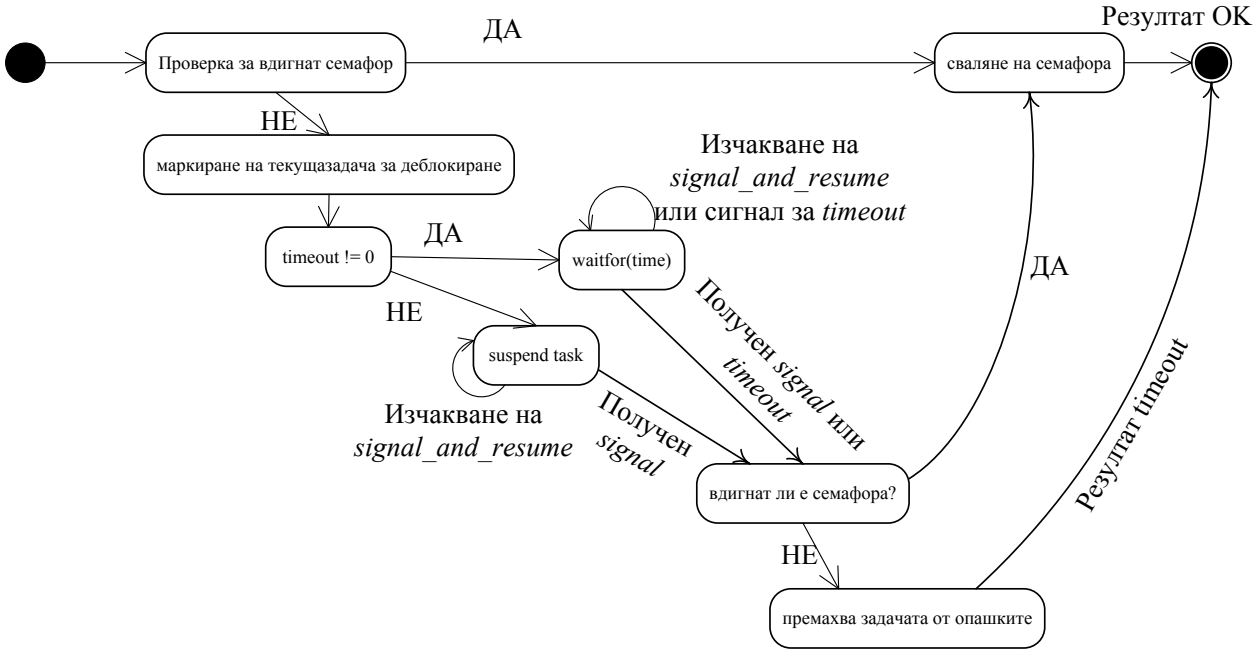
Вектор с чакащи задачи(queue)

Фиг.3.4. Структура на блокиращ векторен семафор

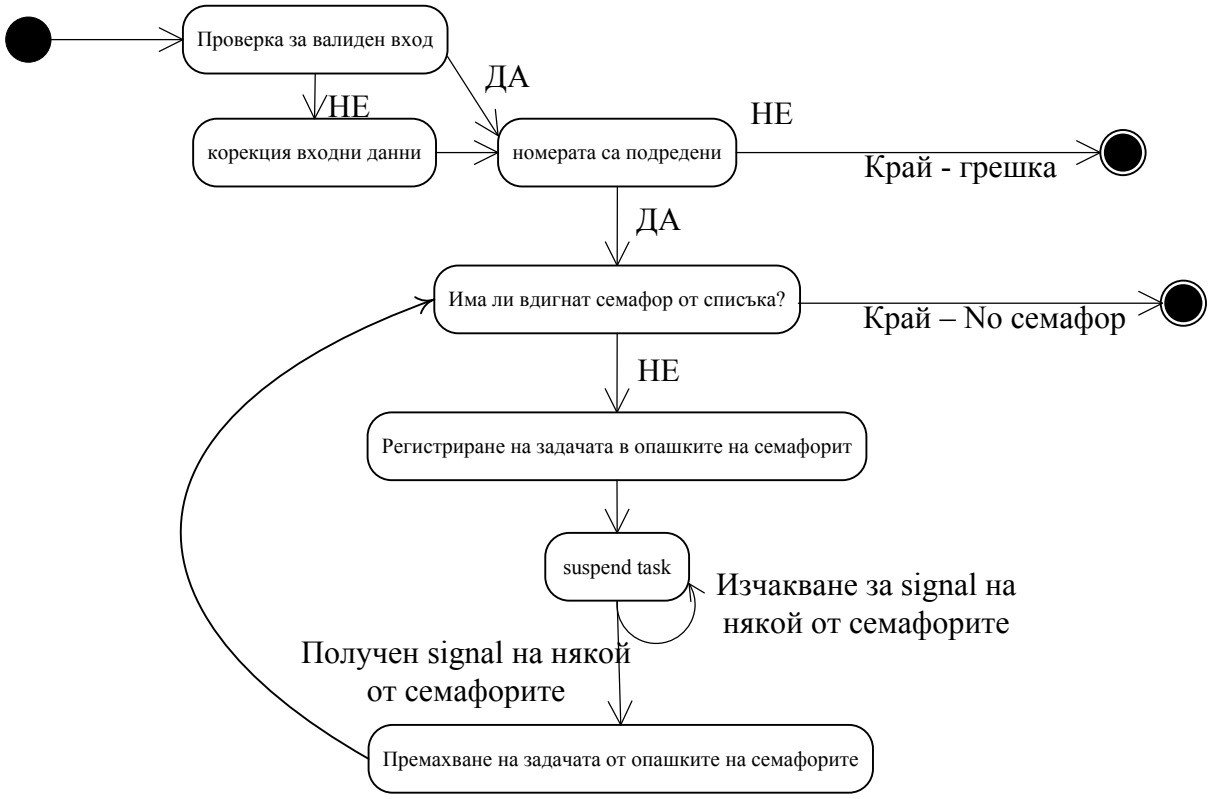
На всяка задача е присвоен отделен семафорен флаг (например бит *i*), съответстващ на нейната битова позиция във вектора на активните задачи (*active tasks vector - ATV*). Чакащите задачи се регистрират динамично в допълнителен двоичен вектор, включен в структурата от данни на семафора. Побитовата обработка позволява едновременно вдигане на множество флагове в един векторен семафор, и едновременното събуждане на сигнализирани задачи чакащи на този семафор. Съответните алгоритми са дадени по-долу във вид на state-chart диаграми. Съответните им псевдокодове са показани в Приложение 1.



фиг. 3.5. Команда „**signal_and_resume”**



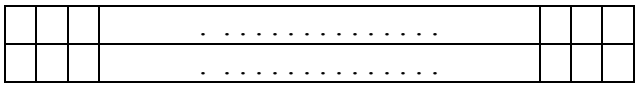
фиг. 3.6. Команда „**wait”**



фиг. 3.7. Команда „**wait_multiple”**

3.3.2.2 Неблокиращо оповестяване на събития

Неблокиращото оповестяване на събития може да бъде реализирано с примитивата *signal_and_release (semaphore, tasks)*, която се използва за сигнализиране на една или няколко задачи за настъпване на събитие, и за генериране на заявки за изпълнение на подмножество от освободени (*released*) задачи; *released* $\subseteq$ *tasks.* Заявяваните задачи са статично регистрирани в този вектор по време на инициализацията на системата. Тази операция е комбинирана с неблокиращата *test_and_reset* примитива.



Семафорни флагове (flags)

Вектор със задачи, подлежащи на заявяване (released)

Фиг.3.8. Структура на неблокиращ векторен семафор

Поради простотата им, алгоритъмът на тези две операции е даден по-долу във вид на псевдокод.

```
//Команда, която сигнализира и заявява за изпълнение (ако
// е необходимо) набор от твърди реално-времеви задачи
signal_and_release( semaphore, tasks )
{
        MAS[semaphore].flags = MAS[semaphore].flags | tasks;
        //сигнализира задачите на дадения семафор
         ATV = ATV | MAS[semaphore].released;
         //Заявяване на набор задачи
}

//Проверява дадения семафор за текущата задача и го нулира, ако е вдигнат
test_and_reset( semaphore )
{
        if( MAS[semaphore].flags[current_task] == 1 ){
          //Вдигнат ли е текущия семафор?
          MAS[semaphore].flags[current_task] = 0;//Да⇒нулиране
                return 1;//Да,семафора беше вдигнат
        } else
                return 0;//Не,семафора е свален
}
```

И двата типа операции за семафорно сигнализиране се изпълняват за константно време, независимо от броя на участващите задачи. Векторните семафори и съответните примитиви съставят основния слой за оповестяване на събития, който се използва за реализация на горния слой за обмяна на съобщения на комуникационния протокол на *HARTEX*.

Векторните семафори и асоциираните примитиви съставят основния слой на синхронизация на задачите, които се използва за оповестяване на настъпването

на събития за по-горния *слой за обмяна на съобщения*, в рамките на различни типове взаимодействие като асинхронна комуникация, а също и синхронна – съобщение-потвърждение, клиент-сървър. Следователно глобалното (broadcast) съобщение се изпълняват също за константно време, без значение на броя на получателите [14][71].

3.3.3 Поднивото за обмяна на съобщения

Поднивото за обмяна на съобщения се състои от комуникационни примитиви, които са проектирани за прозрачна обмяна на съобщения между буфера източник и буфера приемник на взаимодействащите си функционални блокове (и съставящи нишки). Те биват извиквани с имената на комуникационните променливи, с използване на адресиране по подразбиране на съобщенията (*адресиране по съдържание*).

Така формулираните задачи се решават с помощта на три вида системни таблици: таблица за предаване на съобщенията - *Message Transmission Table* (MTT), таблица за оповестяване на получателя - *Receiver Notification Table* (RNT) и таблица за получаване на съобщенията - *Message Reception Table*(MRT). Така чрез тези таблици се реализира механизъм за възприемане на съобщенията по тяхното съдържание (*acceptance filtering*) в контекста както на локални, така и на отдалечени взаимодействия между ФБ.

Въз основа на горния механизъм могат да бъдат реализирани трите вида взаимодействия между ФБ, дефинирани в потребителското ниво на протокола.

Допълнително тези таблици се съпровождат от битов вектор с големина, равна на броя на съобщенията, разпространявани в системата. Тази таблица – *Association Table* (AT) съдържа маркер в полето, отговарящо на идентификатора на всяко съобщение, което е включено в MRT.

Message Transmission Table (MTT)			
Mesasage_Identifier	Source_Address	Message_Length	RNT_Index

Receiver Notification Table (RNT)		
Message_Identifier	Blocking/Non-blocking_Interaction	Receiver_Tasks

Message Reception Table (MRT)				
Message_Identifier	Receiver_Task	Source_Address	Destination_Address	Message_Length

Посочените комуникационни таблици се дефинират за всеки възел от мрежата. В този контекст, последователността на комуникационната транзакция се изпълнява по следния начин:

- Предаването на съобщението започва с извикването на съответната примитива на *HARTEX*, например *broadcast (var_name, deadline)*, където *var_name* е идентификатора на изпращаното съобщение и се използва за намиране на съответстващ запис в *MTT*, след което съобщението се поставя в опашката за изпращане през мрежата.
- При локално взаимодействие *var_name* съвпада с идентификатор в *RNT*, а задачите-получатели се оповестяват чрез сигнализиране на семафор за пристигане на съобщения *(message arrival semaphore - MAS)*. Това е векторен семафор, съществуващ за всяко получено в даден възел съобщение и номерът му се подразбира от индекса на съответния запис в *RNT*. Записите в *RNT* са подредени по намаляващ приоритет. Така, номерата на семафорите отговарят на приоритетното подреждане на получаваните съобщения, като най-малкия номер съответства на съобщението с най-висок приоритет.
- В зависимост от типа на взаимодействието, операцията за сигнализиране на семафор може да доведе до промяна на състоянието на задача, както следва:

Получаващата задача се заявява за изпълнение (ако не е била активна при пристигането на съобщението), когато участва в неблокираща асинхронна комуникация и е указано извикване на задачата. Тази операция може да се извърши едновременно за множество получаващи задачи чрез примитивата *signal_and_release (semaphore, tasks)*, където получателите се посочват с битовия вектор *Receiver_Tasks,* намиращ се в съответния запис на *RNT*, а заявяваните получатели се посочват от допълнителен семафорен вектор. Посочената процедура също се изпълнява за константно време, независимо от броя на участващите задачи.
Получаващата задача се събужда (ако е спряна при пристигането на съобщението), когато участва в блокиращо взаимодействие, т.е. изпращане-потвърждение или клиент-сървър. Този преход се реализира чрез примитивата *signal_and_resume (semaphore, tasks)*, където получателя се посочва от полето *Receiver_Tasks* на съответния запис в *RNT*.

- Накрая, ако са посочени едновременно отдалечена и локална комуникация, съобщението се поставя в опашката за изпращане и същевременно се сигнализират локалните получатели. Този случай е възможен при комуникация един-към-много, реализирана от примитивата *broadcast*.

Комуникацията тип *broadcast*, може да бъде сведена до предаване до няколко получателя (*multicast*), ако се използва техниката за филтриране на приеманите съобщения (*acceptance filtering*), която е типична за мрежи, поддържащи адресиране на съобщения по съдържание (*content-oriented message addressing*). Филтрирането на приеманите съобщения се използва при отдалечени входящи съобщения, чиито идентификатори се анализират за да се реши дали да се приеме или да се игнорира съобщението. Това може да бъде направено по софтуерен начин чрез съдържанието на локалните таблици *AT или RNT,* а при някои мрежи (например *CAN*), филтрирането на приеманите съобщения може да се поддържа от хардуера на комуникационния адаптер.

Когато се приеме съобщението, комуникационния контролер записва съдържанието му в локален буфер-посредник *(proxy source buffer).* Пристигането на съобщението се сигнализира чрез прекъсване към главния процесор. Програмата за обработка на прекъсването сигнализира задачите-получатели и евентуално генерира заявки за изпълнение или събужда получателя, в зависимост от типа на комуникацията и текущото състояние на получателя.

Комуникационната транзакция завършва с активирането на задачата-получател, която изпълнява операция за получаване на съобщението. Неблокиращата операция открива идентификаторите на получените съобщения като изпълнява процедурата *test_and_reset* върху подмножество от семафорите за пристигане на съобщение. Идентификаторите се получават от съответните записи в *RNT*. Тази техника е реализирана в примитивата *receive_multiple(var_names)*, използвана за получаване на множество съобщения в контекста на конкурентно изпълнявани асинхронни транзакции.

Блокиращата операция за получаване, чака подмножество от съобщения и намира идентификатора на сигнализираното съобщение с най-висок приоритет, чрез блокираща семафорна примитива, т.е. *wait_ multiple (first, last)*. Последната връща номера на семафора за пристигане на съобщение, който се използва като индекс към съответния запис в *RNT*. Такъв е случаят с операции като *accept()* използвана за реализиране на синхронни двупосочни транзакции. Алтернативно, блокираща операция изпращане/приемане като *request ( )* може да бъде спряна на конкретен семафор за пристигане на съобщение, докато чака обратното съобщение, т.е. задачата знае предварително точно кое съобщение чака.

Идентификаторите на съобщението и на задачата-получател се използват за намиране на съответен запис в *MRT* и съобщението *автоматично* се копира от буфера-източник(или буфера-посредник) в буфера-получател, при условие че адресите за източник и получател се различават и съобщението не е празно (*Message_Length* $\neq 0$). Горната последователност се изпълнява за всеки идентификатор на съобщение което се получи, ако са повече от едно. Накрая, операцията за получаване на съобщение връща на извикващата задача идентификаторите на получените съобщения. Получените идентификатори могат да бъдат използвани за достъп до текущото състояние на разпределено приложение и да се избере подходяща реакция в рамките на получаващата задача. Те могат да бъдат използвани също така и за диагностика на работното състояние на мрежовите възли и съответните функционални блокове участващи в разпределената транзакция.

(1) Асинхронно взаимодействие производител-потребител :

Това е основният и най-мощен способ за взаимодействие между ФБ, който се използва предимно за обмен на сигнали между динамични подсистеми. По същество, предложеният асинхронен механизъм е еквивалентен на т. нар. “state message communication”, който се счита за най-подходящ при реализация на системи за управление.

Горното взаимодействие се реализира в общия случай по схемата „един към много”. За целта се използват неблокиращи команди от вида:

sender: *... broadcast (var_name, deadline) ...*

receiver(s): *... receive_multiple (var_names) ...*

При това, в зависимост от параметризирането на MTT, RNT и MRT на съответните възли, могат да бъдат реализирани множество различни варианти на взаимодействие:

- според начина на предаване на съобщението:
  - локална комуникация – с или без копиране на съобщението в буфера приемник
  - отдалечена комуникация
  - комбинирана (локална и отдалечена) комуникация
- според взаимодействието с нишката-приемник:
  - Със заявяване на нишката-приемник (ако същата е в неактивно състояние). Този вариант е подходящ за асинхронни, събитийно-активирани РКСУ. Той се използва при синхронни системи, когато в рамките на последователности от ФБ имаме активиране от типа „предшестване-следване".
  - Без заявяване на нишката приемник. Този вариант се използва при синхронни (time-driven) РКСУ. При тях нишката-приемник се активира от местен синхро-сигнал или от синхронизиращ съобщение (synchro-message). Тук се предполага, че входното съобщение пристига преди края на предходния период, при което КСУ функционира като конвейер съгласно принципа resource decoupling.
- според броя на участниците в транзакцията:
  - един към един (unicast)
  - един към много (multicast, broadcast)
  - много към един (или много към много).

Последният случай се характеризира с множество конкурентно изпълнявани транзакции, адресирани към един и същи функционален блок (или нишка-приемник). Това може да доведе до приемането на част от максимално възможния набор променливи (n от k) , $0 \leq n \leq k$. Идентификаторите на реално получените променливи се връщат към нишката от командата за приемане на съобщенията receive_multiple (var_names).

(2) Синхронно взаимодействие „съобщение-потвърждение" (обмяна на команди):

Този вид взаимодействия са еднопосочни и служат за осъществяване на вертикални взаимодействия. Най-често тези взаимодействия са двуточкови – примерно изпращане на команда, от операторска станция, за смяна на режим, или за смяна на параметри, на обект от системата, но е възможна реализация на такава комуникация от тип "един към много" като командата се приема от повече възли в системата. Осъществява се с помощта на команди от вида:

sender: *... send_wait (var_name, timeout) ...*

receiver: *loop*

var_name = *receive ( )*

```
.....                                   // process message denoted by var_name
            confirm (var_name)
        endloop
```

Командата *send* изпраща командното съобщение – по същество *send е* еднотипна с командата *broadcast* с тази разлика, че тя участва в блокораща комуникация и при локално взаимодействие се грижи за отблокиране на нишката приемник; *receive ()* връща на приемащата нишка идентификатора на полученото съобщение;

В зависимост записите в *MTT*, *RNT* и *MRT* на съответните възли могат да бъдат реализирани следните различни варианти на взаимодействие:

според начина на предаване на съобщението:

отдалечена комуникация – с копиране на съобщения от предаващия в приемащия възли;
локална комуникация – с копиране на съобщението в буфера приемник, като един и същи буфер се използува като източник и приемник във всеки един от взаимодействащите блокове;
локална комуникация –без копиране на съобщението, при което се използва един и същи буфер в рамките на ФБ-инициатор на транзакцията.

- според броя на участниците в транзакцията:
    - един към един (unicast)
    - един към много (multicast, broadcast).

(3) Взаимодействие „клиент-сървър”

Взаимодействието „клиент-сървър” е вариант на двупосочна комуникация, който се реализира с помощта на блокиращи команди от вида:

```
client:  ... request (var_name, timeout ) ...

server:  loop
             var_name = accept ( )
             .......        // process request denoted by var_name
             reply (var_name)
         endloop
```

Командата *request* изпраща съобщение-заявка и блокира инициатора на транзакцията за интервал timeout; *accept ()* връща на приемащата нишка идентификатора на полученото съобщение; *reply(var_name)* връща отговор с изискваната променлива на нишката-инициатор, в резултат на което същата се деблокира. При липса на отговор за време по-голямо от зададеното с *timeout,* нишката се деблокира от подсистемата за измерване на времето.

Следва да се отбележи, че при това взаимодействие деблокирането на нишката-инициатор е индикация за завършване на транзакцията, което може да бъде нормално или аварийно (timeout).

При това характерът на комуникацията дава възможност един и същи буфер да бъде използван както за съобщението-заявка, така и за отговора при всеки един от ФБ, които участват в транзакцията.

В този контекст вариантите на „клиент-сървър” комуникацията са три:

- отдалечена – с копиране на съобщенията и в двете посоки на предаване;
- локална – с копиране на съобщението от буфера-източник в буфера-приемник, като един и същи буфер се използва в комуникиращите си нишки и като източник, и като приемник;
- локална комуникация –без копиране на съобщението, при което се използва един и същи буфер в рамките на ФБ-инициатор на транзакцията.

Адресирането по съдържание следва от обектния модел СУПП (ASPS), в който функционалните блокове си взаимодействат с обмяна на сигнали („наименовани съобщения”) (виж по-горе). Тази особеност представлява основното различие от съществуващите ядра, които традиционно използват явно адресиране на съобщенията, например явно наименуване на приложен обект или комуникационен обект като порт, обект-съобщение, цикличен асинхронен буфер и т.н. [111][115]. Експлицитната комуникация води до липса на гъвкавост при проектиране, особено в случаите на директно наименоване на задачи /обекти [40]. Този ефект се редуцира в известна степен в системите, използващи индиректно наименоване - комуникационен порт и обекти-съобщения. Очевидното оптимално решение на горните проблеми се осигурява от адресирането по съдържание, което дава гъвкави и наистина отворени разпределени системи, както в предложения модел СУПП.

3.3.4 Сигнално подниво

Сигнално ниво се състои от определен набор сигнални драйвери. Тук те са посочени само като типове и имена. Подробно описание на сигналните драйвери е дадено в Приложение 1.

Видове сигнални драйвери*:*

3.3.4.1 Драйвери за асинхронна комуникация “производител-потребител”

Изходен драйвер - Изходния драйвер за излъчване на съобщение в рамките на асинхронно взаимодействие производител-потребител е S_OUT (*Signal_varible_OUT*). Драйверът капсулира в себе си композирането на изходящото съобщение от *n* входни променливи, копирането му в изходящия буфер и изпращането му с командата на операционната система *broadcast(var_name,deadline).*

Входен драйвер - Приемане на съобщение в рамките на асинхронно взаимодействие производител-потребител - S_IN (*Signal_varible_IN*). Драйверът скрива в себе си командата на операционната система *receive_multiple*(*var_names*) за приемане на всички постъпили съобщения, декомпозира ги на съставни променливи и попълва стойностите им в съответната изходна аргументна зона на модула. Входове за модула са разширените идентификатори на всички приемани съобщения – те са структура, съдържаща две полета: идентификатор на съобщението и брой изходни променливи, на които се декомпозира. Изходът *flags* (*flags = var_names*) е резултатът от командата *receive_multiple*(*var_names*) и по

същество е масив от идентификаторите на получените съобщения. Като вътрешни променливи модулът пази адресите на съответните входни буфери на съобщенията.

3.3.4.2 Драйвери за синхронна комуникация

Изходен драйвер - Изходния драйвер за синхронен обмен на команди осигурява композирането на командното съобщение и изпращането му с командата на ОС *send(var_name).* Приема като входен аргумент идентификатора на изпращаното съобщение и променливите, съставящи командата.

Входен драйвер - Входният драйвер C_IN е предназначен за прочитане и декомпозиране на получената команда на съставни променливи. Входен аргумент е идентификатора на командното съобщение.

Входен драйвер за приемане и оповестяване на команди - Драйверът за оповестяване на команди C_NF (Command_NotiFication) капсулира блокиращата примитива на ОС *receive*( ), която връща като резултат идентификатора на най-приоритетното пристигнало съобщение. Драйверът сигнализира за получена команда чрез вдигане на съответен флаг – променлива от своята изходна аргументна зона и нулира останалите. Входни аргументи са идентификаторите на приеманите команди.

3.3.4.3 Драйвери за комуникация тип „клиент-сървър”

Драйвер за клиентска заявка - Драйверът съдържа блокиращата командата на ОС *request*(*var_name, timeout*), с която се изисква дадена променлива. При получаване на желаната променлива, изпълнението продължава с декомпозирането ѝ на съставящи и записа им в изходната аргументна зона. Ако заявеното съобщение не пристигне в рамките на *timeout* период ОС деблокира задачата.

Драйвер за нотификация на постъпила заявка - Драйверът R_NF (Request_NotiFication) служи за оповестяване на получена заявка в рамките на двупосочното двуточково „клиент-сървър” взаимодействие. Той е напълно аналогичен на C_NF с тази разлика, че командата *receive*( ) се замества от *accept*( ).

Драйвер за отговор на клиентска заявка - Драйверът R_OUT(Replay_variable_OUT) осигурява композирането и изпращането на отговор на клиентска заявка. Съдържа командата на ОС *replay*( ), която изпраща съобщението-отговор към клиента.

Резултати:

- Направен е анализ на подсистемите на ОСРВ. Предложен е обектен подход при разглеждането и композицията ѝ .
- Предложен е формален модел за описание на комуникационните транзакции на РКСУ.
- Предложена е структура на системните команди, която гарантира константно време за изпълнение.
- Предложена е структура на системните услуги, които елиминират наличието на опашки.

- Предложена е структура на комуникационната подсистема и протокол, базирани на контекстно-ориентирано адресиране.

Гл. 4. Интерпретация на модела на управляващата програма

В глава 4 се разглежда организацията на управляващата подсистема - базовата библиотека на ФМ и двата основни интерпретатора – на автоматен граф (state-graph) и на трансформационен (сигнален) граф (sequencer interpreter), а така също реализацията на предикатните изчисления на базата на BDD (binary decision diagram). Разгледана е методика за синтезиране на съставни ФМ.

4.1. Организация на управляващата подсистема

РСУ е сложна техническа система, която може да се разгледа като композиция от взаимодействащи си подсистеми. В рамките на компонентния подход, всяка подсистема се реализира чрез един или няколко обекта – така наречените функционални блокове. На свой ред взаимодействията между функционалните блокове съответстват на първичните взаимодействия (например обмяна на сигнали), дефинирани в неформалната функционална спецификация на РКСУ.

В този контекст програмната спецификация на системата включва три взаимно свързани аспекта:

- Спецификация на функционалните блокове
- Спецификация на взаимодействията между функционалните блокове (в статика)
- Спецификация на поведението на РКСУ (в динамика)

4.2. Структурна спецификация на стандартни програмни обекти

Формалните модели, разгледани в предишните глави са използвани при създаването на метод за компонентно проектиране на програмното осигуряване за РКСУ, които се характеризира със следните по-важни особености:

Системата се изгражда въз основа на библиотеки от стандартни програмни компоненти със стандартна вътрешна структура в рамките на йерархия от прости и съставни обекти. В това число базови ФМ, съставни ФМ, функции (activities) и ФБ.

Софтуерните компоненти си взаимодействат чрез йерархия от комуникационни и синхронизационни протоколи, осигуряващи безопасно и предсказуемо поведение в контекста на система с отворена архитектура.

По конкретно, в системите изградени на базата на ASPS модела, се използват следните видове модели:

- Функционални модули (ФМ)
- Съставни функционални модули (СФМ)
- Функции (activities, процеси)
- Функционални блокове (ФБ).

При изграждане на различни видове обекти са използвани следните принципи на компонентно програмиране – капсулиране, агрегиране и асоциация, при което се цели емулиране на интегрални схеми и структури от интегрални схеми при изграждане на приложните системи. По-специално:

- Базовите (библиотечни) приложни функции се капсулират в ФМ;
- Екземпляри на функционални модули се агрегират в съставни функционални модули, като получената конструкция може евентуално да

бъде капсулирана като черна кутия, която е външно подобна на прост ФМ;

- Съставните функционални модули са компоненти за изграждане на т.н. функции (activities) на функционалните блокове, т.е софтуерни конструкции (схеми), генериращи сигнали на едни или други външни или таймерни събития;
- Функциите се капсулират във функционални блокове, като последните представляват софтуерни еквиваленти на интегрални схеми с голяма степен на интеграция.

Функционалните блокове се асоциират едни с други в рамките на определени конфигурации, които външно приличат на структури от интегрални схеми с висока степен на интеграция.

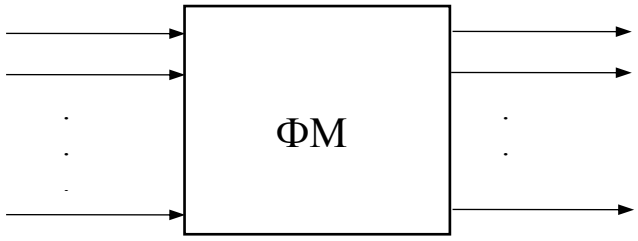
По-долу са описани накратко различните функционални обекти, като се засягат и техни особености, които не са включени в гл. 2.

4.2.1. Видове програмни обекти

➢ *Функционални модули*

Функционалните модули са стандартни компоненти, използвани при изграждането на функциите (activities), капсулирани в ФБ. Чрез тях се реализират специфични функции на управлението, като:

- Различни видове драйвери за локален и отдалечен (комуникационен) вход/изход.



Фиг.4.1. Функционален Модул (ФМ)

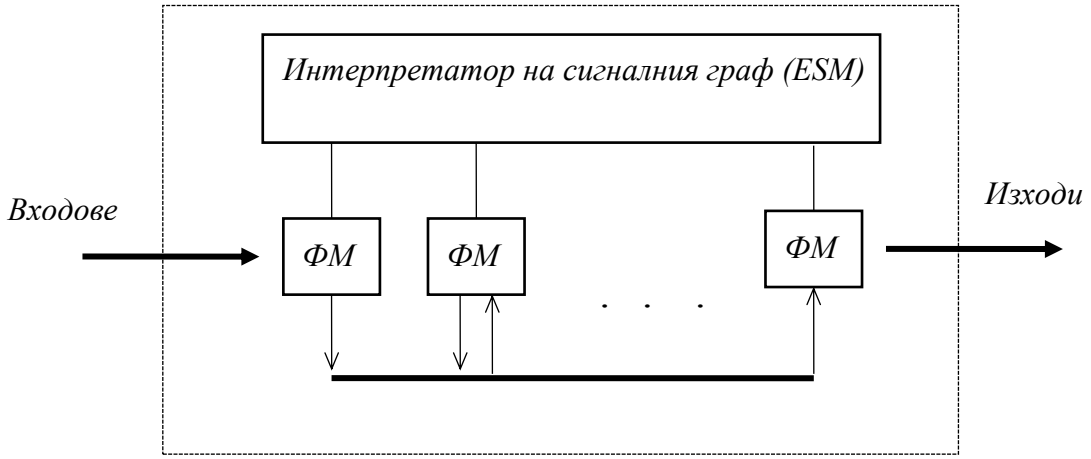
- Предварителна обработка на информацията, получавана от обекта: линеаризиране и преобразуване в технически единици, цифрова филтрация, различни видове проверки (за отклонение от технологична и аварийна граници), проверка на градиента и др.
- Конвенционални алгоритми за управление, включително различни модификации на класическия ПИД регулатор.
- Сложни алгоритми за управление в контекста на системи, описани с апарата на пространството на състоянията. Например наблюдател на състоянието, регулатор с обратна връзка по състояние или изход, адаптивни и самонастройващи се регулатори.
- Функционални модули, необходими за реализация на дискретни контролери, например генератор на двоични управляващи сигнали и последователности, компаратори, таймери, броячи на събития и др.
- Спомагателни модули като например модели на динамични звена, нелинейни елементи, суматори и умножители, генератори на сигнали и шумове и т.н.

Функционалните модули представляват обекти от най-ниско ниво, които се характеризират със стандартна вътрешна структура. Функционалният модул се състои от три компонента:

- Реентрантна функция (реализирана основно на език С);
- Известен брой скаларни и/или векторни входове и изходи;
- Структури с данни, състоящи се от аргументи (параметри), а така също и адресни дескриптори, които се използват за достъп до клетки от паметта, съдържащи аргументи и входно/изходни величини.

Входно/изходните връзки и конкретните стойности на аргументите на ФМ се определят при конфигурирането на приложната система.

Както беше отбелязано, ФМ се използват при конфигурирането на съставни функционални модули, чрез които се реализират по-сложни трансформации на сигналите, например формиране на управляващо въздействие при управление в затворен контур, цифрова обработка на сигнали и др. Съставните ФМ се описват чрез така наречения сигнален граф, или граф на потока на сигналите, който задава неявно последователността на тяхното изпълнение (глава 2). ФМ се извикват от стандартна служебна програма, наречена интерпретатор на сигналния граф (Execution Sequence Manager - ESM). Конструкцията на съставния функционален модул е дадена на фиг. 4.2.

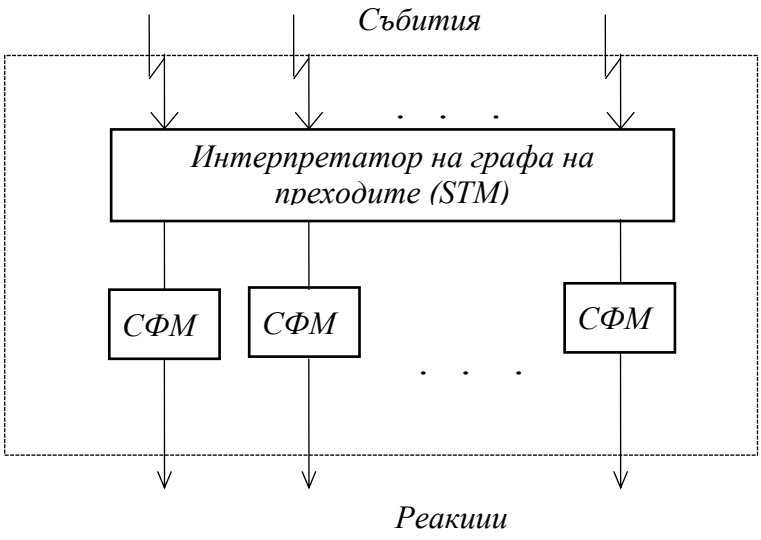


Фиг. 4.2. Съставен функционален модул

➢ *Функции (activities)*

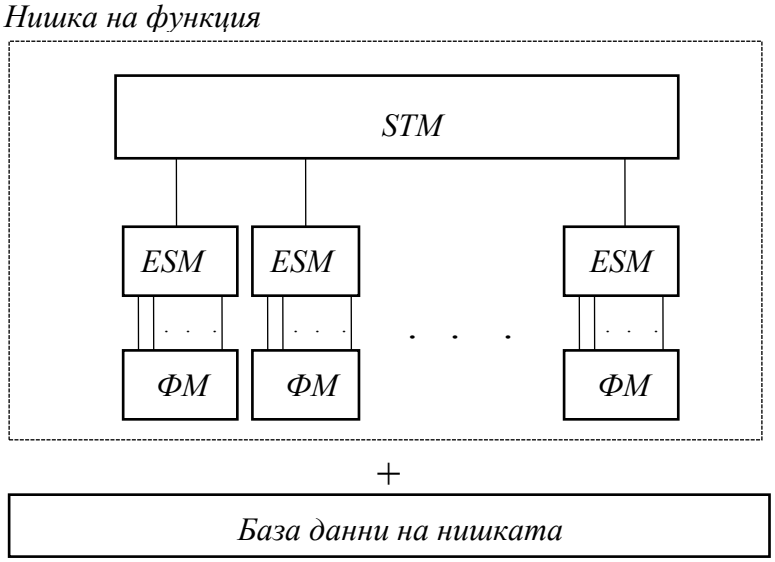
Функционалните блокове съдържат една или няколко функции, чието поведение се описва с формалния модел, разгледан в гл. 2 – така наречения хибриден автомат.

Поведението на хибридния автомат се описва чрез графа на преходите на автомата, който се интерпретира от служебна програма – интерпретатор на



Фиг. 4.3. Нишка на функция

графа на преходите - ИГП (State Transition Manager - STM). Последният се активизира от времеви или външни събития, което води до определяне на



Фиг.4.4.

текущото състояние на автомата и реакцията, свързана с това състояние, чрез която се реагира на възникналото събитие. Самата реакция се реализира чрез съставни функционални модули (СФМ), които се активизират от ИГП (STM) (фиг.4.3).

Съответно СФМ генерира един или няколко сигнала като реакция, които се дефинират чрез съответните сигнални графи. В крайна сметка се поражда софтуерна конструкция (фиг. 4.4), от която се вижда, че функцията може да

бъде представена като съвкупност от две обособени части - процесна нишка (нишка на функцията) и база данни на функцията.

Процесната нишка (activity thread) се състои изцяло от стандартни и обикновено резидентни компонентни (при малки системи – ROM-резидентни), показани на фигурата, докато базата данни на нишката съдържа всички структури с данни, зависещи от конкретното приложение.

- Граф на преходите на функцията (activity)
- Сигнални графове на съставните функционални модули
- Аргументна зона на ФМ и т.н.

По такъв начин една функция може да бъде реконфигурирана чрез модифициране на базата данни, вместо чрез традиционното препрограмиране, което очевидно е неприложимо, когато е необходимо оперативно реконфигуриране на приложната система.

Изпълнението на изчислителните процеси в този случаи се осъществява чрез интерпретиране на съдържанието на стандартните структури с данни, описващи графа на преходите и свързаните с него сигнални графи на нишката. Това може да доведе до различни видове поведение в зависимост от приложния контекст:

- В непрекъснати системи, спецификацията на функцията (activity) задава поредица от изчисления, т.е. сигнални трансформации, чрез които се формира управляващото въздействие за съответния режим на работа (автоматично, ръчно и др.). Поредицата от изчисления се активизира периодично от таймерни събития и се изпълняват изцяло в рамките на текущия такт на дискретизация.
- В дискретни системи за управление спецификацията на функцията (activity) задава поредици от дискретни състояния и управляващи сигнали, които се изпълняват като стъпки на управлението в рамките на поредица от тактове на управляващия автомат. В този случай изчислителният процес се активизира от периодично постъпващи таймерни събития (при синхронен автомат) или от спонтанно постъпващи външни събития (при асинхронен автомат).

Двата вида поведение могат да бъдат комбинирани при реализирането на хибридни системи за управление. Например многорежимни регулатори, сложни дискретни контролери, контролери за комбинирани (batch) процеси.

➢ *Функционални блокове*

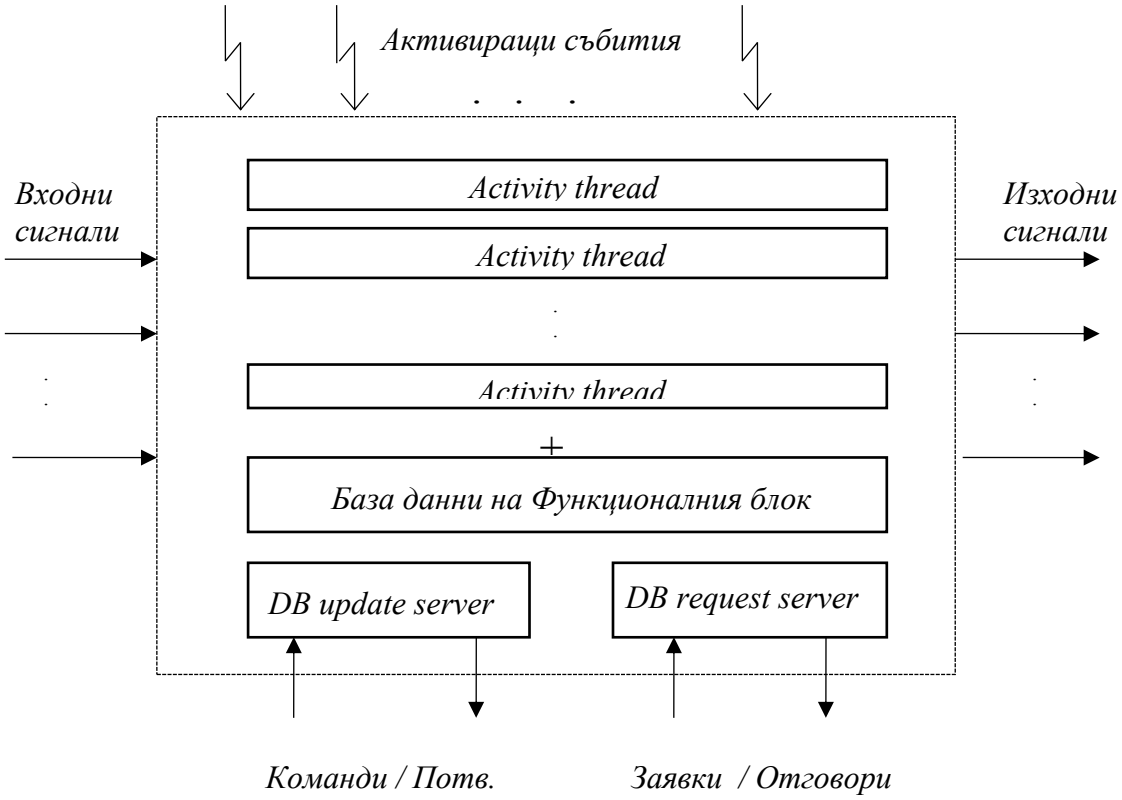
Функционалните блокове (ФБ) представляват софтуерни интегрални схеми със стандартна вътрешна структура (виж фиг. 4.5). Във всички ФБ се капсулират известен брой функции (activities), които в крайна сметка се реализират като процесни нишки (activity threads), взаимодействащи с базата данни на функционалния блок. Освен това във ФБ могат да се съдържат някакви спомагателни процесни нишки, като интерпретатор на отдалечените команди, сървър на отдалечената база данни и др.

Базата данни на ФБ се използва за спецификация на съставните процесни нишки и на съставните функционални модули. Тя може да бъде представена като обединение на базите данни на отделните активности, от които е съставен функционалния блок. По такъв начин структурата на ФБ се определя от съдържанието на базата данни, в която може да бъде съхранявана библиотека

с функционални модули, при условие че е разработена стандартна организация на базата с данни (виж по-надолу).

Базата с данни има още една важна функция. Тя подържа комуникацията между процесните нишки на ФБ чрез общи структури с данни (защитени записи), на базата на специален протокол за управление на достъпа до защитените ресурси. Така наречения протокол за условно наследяване на приоритетните тавани на ресурсите (Conditional Sealing Priority Protocol), който се подържа от ОСРВ “HARTEX”.

Функционалните блокове могат да се разглеждат като приложни



Фиг. 4.5. Функционален блок (ФБ) - вътрешна структура

програмни обекти, реализиращи различни видове интелигентни устройства като: непрекъснати и дискретни контролери, интелигентни датчици и изпълнителни механизми и др.

Функционалните блокове могат да бъдат използвани за конфигуриране на разпределено приложение. За целта следват да бъдат дефинирани взаимодействията между тях, в това число статична дефиниция на връзките между ФБ и съответните съобщения (сигнали), което става с помощта на комуникационния граф на РКСУ (виж глава 3). Спецификация на взаимодействията в динамика, т.е. на разпределените транзакции и техните атрибути, като активиращи събития, задачи (фази) на транзакцията, параметри на задачите като период, отместване, времеви граници и т.н.

В следващия параграф е дадено описание на базата данни на функционалния блок в съответствие с направеното по-горе уточняващо разглеждане.

4.2.2. База данни на функционален блок

Описаните по-горе и в глава 2 графове (граф на преходите и граф на сигналните трансформации) имат свое машинно представяне. Това представяне е в табличен вид и включва следните видове таблици:

- Таблица на изпълняваните задачи (Task Execution Table)
- Таблица на графа на преходите (State Transition Graph Table - STG)
- Таблица на състоянията (State Specification Table -SST)
- Таблица на сигналните графи (Signal Flow Graph Table - SFG)
- Таблица на ФМ (Function Module Instance Invocation Table)
- Таблица на В/И дескриптори на екземплярите на ФМ (Function Module Instance I/O Descriptor Table)
- Таблица на аргументната зона на екземпляр на ФМ (Functional Module Instance Argument Table)

Подробно описание на тези таблици е направено в следващите параграфи

4.2.2.1. Таблица на изпълняваните задачи (Task Execution Table)

T_1 ->	Thread Identifier	
	Initial STG index	
	Initial SST index	Task execution block – T_1
	Current state	
	Default/explicit	
⋮	⋮	⋮
T_n ->		
		Task execution block – T_n

Табл. 4.1. Таблица на изпълняваните задачи

Тази таблица съдържа изходни данни, необходими за изпълнението на задача/нишка (task/thread) на ОСРВ. Често тя се описва със съответния граф на преходите (STG). Нейните полета са:

- Идентификатор на задачата (Thread Identifier)
- Начален адрес на фрагмент на STG таблицата, в който се съдържа графа на преходите (STG) на задачата (Initial STG index)
- Начален адрес на фрагмент на SST таблицата, в който се съдържа описание (спецификация) на всяко едно от състоянията (Initial SST index)
- Текущото състояние на задачата (Current state)
- Default / explicit entry (DEE) – флаг, който индицира начина на стартиране на задачата
  a) от началното псевдосъстояние a_0, последвано от нормална серия от състояние/реакция (DEE = ∅);

b) от не-начално състояние, задачата се активизира от външна задача, т.е при преход към вътрешно елементарно състояние на сложно серийно състояние (DEE=1).

В последния случай се изпълнява реакцията, асоциирана с текущото състояние, след което флагът DDE се нулира.. По-нататък се активира нормалната поредица от състояния / реакции, съгласно стандартния механизъм за интерпретиране на STG и останалите таблици (виж по нататък).

4.2.2.2. Таблица на графа на преходите (State Transition Graph Table - STG)

В тази таблица се съдържат графите на преходите на отделните задачи (STG)

STG_1	← Initial_STG (T_1)
STG_2	← Initial_STG (T_2)
⋮	← Initial_STG (T_3)
STG_n	← Initial_STG (T_n)

Табл. 4.2. Таблица на графа на преходите

	Predicate var address	N / T_0	Successor $_0$	N / T_1	Successor $_1$	
Initial (T) -->	x_1^0		$a/x(x_1^0=0)$		$a/x(x_1^0=1)$	
	x_2^0		$a/x(x_2^0=0)$		$a/x(x_2^0=1)$	BDD (Fa_0)
	⋮		⋮		⋮	
	$x_{r_0}^0$		$a/x(x_{r_0}^0=0)$		$a/x(x_{r_0}^0=1)$	
	⋮		⋮		⋮	
Index Fa_k-->	x_1^k		$a/x(x_1^k=0)$		$a/x(x_1^k=1)$	
	x_2^k		$a/x(x_2^k=0)$		$a/x(x_2^k=1)$	BDD (Fa_k)
	⋮		⋮		⋮	
	$x_{r_k}^k$		$a/x(x_{r_k}^k=0)$		$a/x(x_{r_k}^k=1)$	
	Predicate var address	N / T_0	Successor $_0$	N / T_1	Successor $_1$	BDD (Fa_0)

Initial (T) -->	x_1^0		$a/x(x_1^0=0)$		$a/x(x_1^0=1)$	
	x_2^0		$a/x(x_2^0=0)$		$a/x(x_2^0=1)$	
	$\vdots$		$\vdots$		$\vdots$	
	$x_{r_0}^0$		$a/x(x_{r_0}^0=0)$		$a/x(x_{r_0}^0=1)$	
	$\vdots$		$\vdots$		$\vdots$	
Index Fa_k-->	x_1^k		$a/x(x_1^k=0)$		$a/x(x_1^k=1)$	BDD (Fa_k)
	x_2^k		$a/x(x_2^k=0)$		$a/x(x_2^k=1)$	
	$\vdots$		$\vdots$		$\vdots$	
	$x_{r_k}^k$		$a/x(x_{r_k}^k=0)$		$a/x(x_{r_k}^k=1)$	

Табл. 4.3. Блок от STG – двоична диаграма на преходите на едно състояние

На свой ред всеки STG блок се състои от определен брой булеви диаграми (BDD – binary decision diagrams) за оценяване на предикатите, чрез които се избира състоянието-наследник на текущото състояние.

При това

$$\forall a_i / i = 0 \div k : F_{a_i} \leftrightarrow BDD(F_{a_i}) .$$

В *BDD* са използвани следните определения:

- *Адрес на булев операнд(БО)* (Predicate var address)= адрес на източника на булева променлива (най-често изходния байт на ФМ, в който е записано значението на операнда)

  N/T = $\begin{cases} \text{T - терминален връх на BDD} \\ \text{N - нетерминален връх на BDD} \end{cases}$

- Successor 0/1 (x_j^i) - наследник при БО $x_j^i =$ true, съответно false.

  индекс на БО – при T/N = 0 (нетерминален връх)
  индекс на състоянието при T/N =1 (терминален връх).

- *Индекс на БО* – номер на ред от BDD , където се намира следващ БО, тестван в рамките на текущата BDD;

- *Индекс на състоянието* – номер на състоянието, респективно индекс на съответния ред от SST блок, описващ семантиката на състоянията, респективно на избраното състояние.

Забележки:

- Входните сигнали (аналогови, дискретни и двоично-кодирани) се снемат от ФМ - драйвери, В/И локални и комуникационни (remote I/O) драйвери.

- Вътрешни събития, сигнализирани чрез семафори, също се снемат чрез ФМ, които капсулират съответните функции на ОСРВ (wait , test_and_reset и др.)
- Изчислимите логически условия се формират от изчислителни и/или релационни ФМ.

В резултат на това се <u>унифицира</u> начина за посочване на булевия операнд на базата на адрес и номер на бит (конкретно представяне на абстрактния Булев тип - чрез индекс в битов масив), независимо дали е вътрешно или външно, изчислимо и др. (виж по-горе).

Логическите условия се формират от изчислителни вериги, тоест последователности от ФМ (SFG), зададени в предикатната част на SFG таблицата. Самите предикати се изчисляват при обработката на съответната BDD на STG таблицата.

4.2.2.3. Таблица на състоянията (State Specification Table -SST)

Тази таблица се състои от блокове (по-един за всяка задача), като във всеки такъв блок се съдържа описание (спецификация) на състоянията на задачата и протичащите в тях реакции.

SST block		
	SST_1	← Initial_ SST (T_1)
	SST_2	← Initial_ SST (T_2)
	⋮	
	SST_n	← Initial_ SST (T_n)

Табл. 4.4. Таблица на състоянията - SST

Всеки такъв блок има следния вид:

	Reaction Specification				Successor States	
	R_type	Reaction/Task(s)	Period	Deadline	P_index	Fa_index
a_0 →					$P(a_0)$	Fa_0
a_1 →					$P(a_1)$	Fa_1
	…	…	…	…	…	…
a_k →					$P(a_k)$	Fa_k

Табл. 4.5. Блок от таблица на състоянията

Спецификацията на състоянието се състои от две части:

- Спецификация на пораждащите реакции / задачи (*Reaction Specification*)
  R_type – задава вида на реакцията , както следва:
  0 – нулева реакция със спонтанен преход към следващото състояние
  1 – еднократна реакция със спонтанен преход към следващото състояние
  2 – еднократна реакция със събитийно-активиран преход към следващото състояние

3 – периодична реакция със събитийно-активиран преход към същото / следващото състояние от таймерно / външно събитие

4 – външна реакция, реализирана от дъщерна задача или набор от такива задачи, заявявани/ изпълнявани в текущото състояние

Reaction / Tasks

Reaction: индекс на управляващ оператор, т.е номер на началния ред на SFG, в който е описан операторът (примерно X_i)

Или

Tasks – булев вектор, който задава подмножеството от задачи , подлежащи на активиране (виж гл. 4).

Period/deadline – времеви параметри на реакцията за съответното състояние (*mode*)- ако реакцията е вътрешна. При външна реакция тези параметри следва да бъдат детерминирани по аналогичен начин за пораждащите дъщерни задачи (т.е на друго място в същата тази таблица).

Горната информация се използува от STM при изпълнението на задачата, като интервалите се отчитат от момента на заявяване, който се фиксира в TCB на заявената задача (т.е таймерите се инициализират от STM).

- Спецификация на наследниците (*Successor States*), в това число:

  Индекс на предиката (*P_index*)

  Номер на начален ред на SFG, в който се описва поредицата от изчисления, необходими за формиране на съответните булеви променливи (този SFG се намира в предикатната част на SFG таблицата, така наречената *Predicate Execution Table*)

  Индекс на BDD (*Fa_index*)

  Съответства на изображението на текущото състояние F_{a_i} .

Ако *next_BDD_index* = *Inial_SFG*, това е индикация за връщане в началното състояние $\Rightarrow$ *current_sate* = $\varnothing$.

4.2.2.4. Таблица на сигналните графове (Signal Flow Graph Table - SFG)

Това представлява таблицата на сигналните графове. Тук се включват графовете на предикатите, самите реакции и на съставните функционални модули.

Индекс P_1 →	SFG(P_1)	Predicate Processing
P_2→	SFG(P_2)	
	….	
P_m→	SFG(P_m)	
Индекс Y_1→	SFG(Y_1)	Reaction Execution
Y_2→	SFG(Y_2)	
	…	
Y_2→	SFG(Y_n)	
Индекс CFM_1→	SFG(CFM_1)	CFM Specification
CFM_2→	SFG(CFM_2)	
	…	
CFM_r→	SFG(CFMr)	

Табл. 4.6. Таблица на сигналните графове

Всеки сигнален граф има следната структура:

	FMI No	Next (M) index	
Индекс Y_1→			NeM_o^i
	….	….	
	break		
Индекс NeM_1^I→			NeM_i^i
	…	…	
	break		
	…	…	
Индекс NeM_{ki}^I→			NeM_{ki}^i
	…	…	
	break		

Табл. 4.7. Структура на сигнален граф

Интерпретацията на сигналния граф става по следния алгоритъм:

- Достъпът до съответните SFG, т.е. до съответния фрагмент от таблицата се осъществява чрез номера на реакцията Y_i (или чрез съответния начален индекс, съответстващ на Y_i).

- Обработката му завършва след изпразване на опашката на готовите за изпълнение ФМ, която се формира и обслужва от *Execution Sequence Manager.*
- Ако даден ФМ е терминален, то съответното поле *Next* (*FM*) съдържа нулев индекс (*NULL*).
- Изображението FM_0 съответства на всички входни ФМ, т.е. ФМ подлежащи на активиране при стартиране на последователността от ФМ (това са модулите наследници на един фиктивен начален модул M_0, респ. FM_0)
- SFG таблицата се използва за активиране на ФМ в нужната последователност според принципа "поток на сигналите". Информационните (сигналните) връзки между ФМ се задават чрез адресните указатели, съдържащи се във *FM invocation* блоковете (таблица 4.8, 4.9).
- SFG графите се изграждат въз основа на два вида модули.
  1. изпълнителни модули
  2. модул за управление на потока на данните, в това число:
     - превключвател (switch)
     - комутатор (multiplexor)

  По този начин могат да се реализират алтернативни последователности от ФМ, зависещи от изчисляеми логически условия (например признак за невалидност на входния сигнал, генериран от АЦП модула и др. такива), т.е. условия, които се формират по време на изчисленията и не се отчитат при изчисляване на предикатите на SFG таблицата.

Изпълнението на ФМ се разрешава при нулиране на *activation counter*. Изпълнението на всеки модул-предшественик води до декрементиране на АС (независимо от броя на подаваните сигнали от предшественика). Това означава, че началната стойност на АС е равна на броя на ФМ-предшественици (*predessor count*) в SFG.

Забележка: Не се считат за предшественици модули, принадлежащи на отделни сигнални вериги (със своя динамика на активиране и изпълнение), които предоставят допълнителни сигнали – напр. сигнали с характер на параметри – виж примерите в гл. 5.

4.2.2.5. Таблица на ФМ (Function Module Instance Invocation Table)

Таблицата на ФМ дава едно обобщено описание на функционалните модули, използвани в системата. Това описание се конкретизира посредством други две допълнителни таблици, описани в по-надолу.

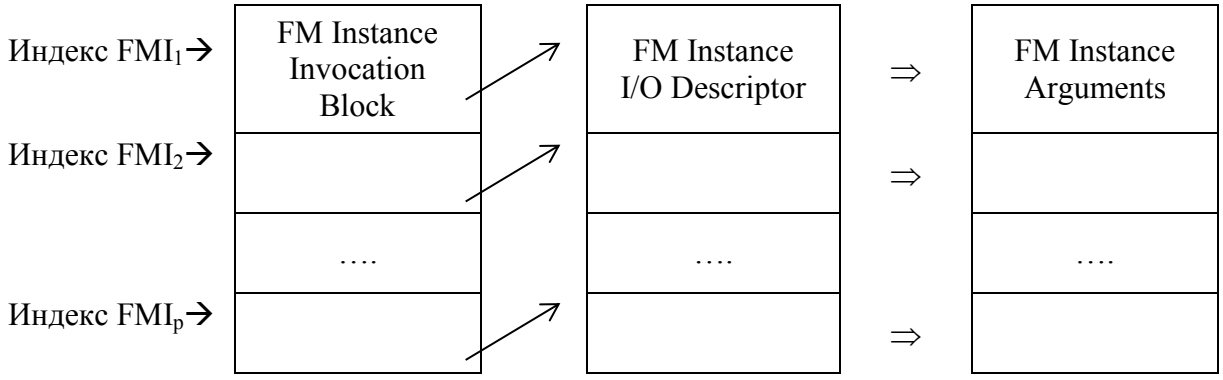


Табл. 4.8. Таблица на функционалните модули

Спецификацията на таблицата на ФМ е следната

FMI_1 index→	FM Identifier
	Activation Counter (AC)
	Initial AC Counter
	Resource No
	Basic / Composite
	Input Count
	Output Count
	Internal Variables Count
	I/O descriptor index
	…..
FMI_n index→	

Табл. 4.9. Структура на блок от таблицата на функционалните модули

Спецификацията на полетата в таблицата е дадена по-долу:

- Идентификатор на ФМ (FM Identifier). ФМ се идентифицира с два числени атрибута със следното формално описание (instance count | catalogue ID)
  - номер на ФМ (т.е. на екземпляра от даден тип - instance count) - поле в таблицата на сигналния граф
  - идентификатор (catalogue ID)– каталожен номер на функцията. Служи за индекс в ТСА (таблица на стартовите адреси) на модулите. Когато модулът е съставен, този номер е празен (NULL).
- Активиращ брояч (Activation Counter)
- Начална стойност на активиращия брояч (Initial AC Counter)
- Номер на ресурса, които се използва от съответния функционален модул (Resource No)
- Тип на модула (Basic / Composite)– прост/съставен функционален модул
- Брой на входни променливи (Input Count)
- Брой на изходни променливи (Output Count)
- Брой на вътрешните променливи (Internal Variables Count)
- Указател към таблицата с В/И дескриптори (I/O descriptor index).

Така в тази таблица се съдържа информация за това какви модули се използват, какви функции имат и спецификация на променливите, с които работят.

4.2.2.6. Таблица на В/И дескриптори на екземплярите на ФМ (Function Module Instance I/O Descriptor Table)

За всеки вход/изход на ФМ се дефинира:

- Тип на променливата
- Размерност (*rows* × *columns*)
- Свързване (*external/local*)
    - към изхода на друг модул
    - към собствен (параметричен) вход, т.е. към клетка (*n*) от собствения АБ
- Адрес на променливата
    - адрес на входа според вида на свързване – адрес на изх. клетка на модула-източник, респ. адрес на клетка от собствения аргументен блок.
    - адрес на изхода – адрес на изходната клетка от собствения аргументен блок (*local*) или адрес на входната клетка на модул-приемник (*external*).

	Variable Table	External/Loc	Address	Rows	Columns	
I/OP index→						I_1
2 →						I_2
3 →	…	…	…	…	…	…
p →						I_p
1 →						O_1
2 →						O_2
	…	…	…	…	…	…
q →						O_q

Табл. 4.10. Таблица на В/И дескриптори на ФМ

4.2.2.7. Таблица на аргументната зона на екземпляр на ФМ (Functional Module Instance Argument Table)

Аргументната зона е разделена на три части.

- Входна зона
- Изходна зона
- Зона на вътрешните променливи

Inputs
Output
Internal Variables

Табл. 4.11. Таблица на аргументната зона на ФМ

4.2.2.8. Служебни програми.

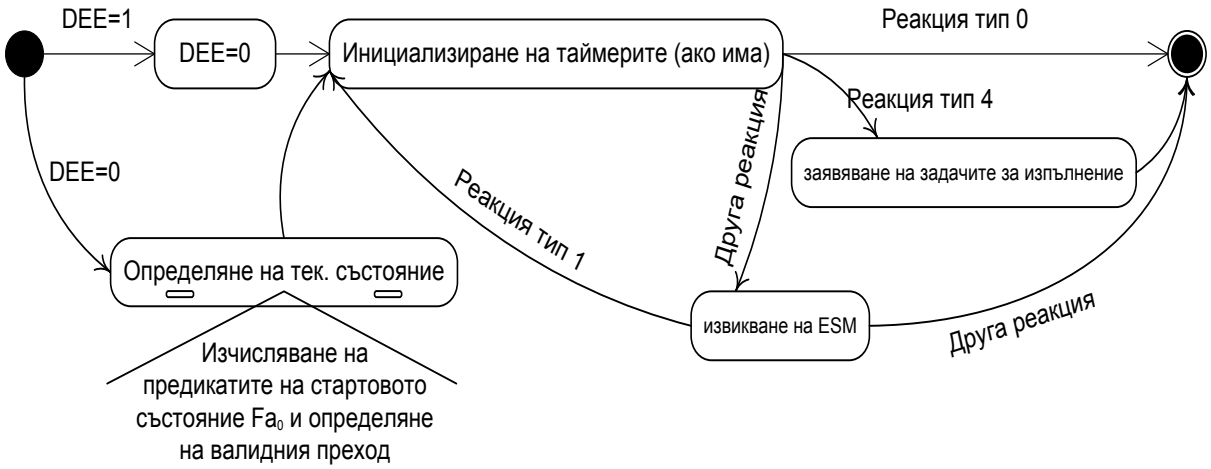
В тази част са описани по-подробно двете споменати по-нагоре служебни програми „Интерпретатор на графа на състоянията” и „Интерпретатор на сигналния граф”. Дадени са техните диаграми на състоянията и коментар по отношение начина на работа. Псевдо-кодовете им са дадени в Приложение 1.

➢ *Интерпретатор на графа на състоянията (State Transition Manager)*

Интерпретаторът на графа на състоянията представлява отделна задача на операционната система. При своята работа, както беше споменато той използва таблицата на графа на състоянията (STG) и таблицата на състоянията (SST). На фиг. 4.6. е показана диаграмата на състоянията му. Работата на STM може да се опише по следния начин:

Фиг. 4.6. Интерпретатор на графа на състоянията – диаграма на състоянията

- В началото се проверява стойността на флага *DEE* . Неговата стойност показва дали работата започва от началното състояние или от някакво текущо състояние, определено от полето *current State* (таблицата на изпълняваните задачи). Ако флагът е вдигнат, то той се сваля и работата започва от *current State,* като се пристъпва към изпълняване на това текущо състояние. Ако флагът не е вдигнат, се стартира процедура за определянето на текущото състояние
- Определянето на текущото състояние стартира, като се изчисляват предикатите, необходими за определянето на валиден преход. Това става с помощта на разгледания по-долу ESM. Следва да се извършат необходимите проверки за установяването на новото състояние. Това представлява интерпретация на графа на преходите и определяне на състоянието, което се записва в полето *current State* (таблицата на изпълняваните задачи). След тези действия се преминава към интерпретация на действията, предвидени в това състояние.
- Интерпретацията на текущото състояние по същество е изпълнение на сигналния граф, асоциран със съответното състояние. Предварително се извършват няколко други операции. Започва се с инициализирането



на таймерите за *deadline* и *period*. Следва проверка за типа реакция:

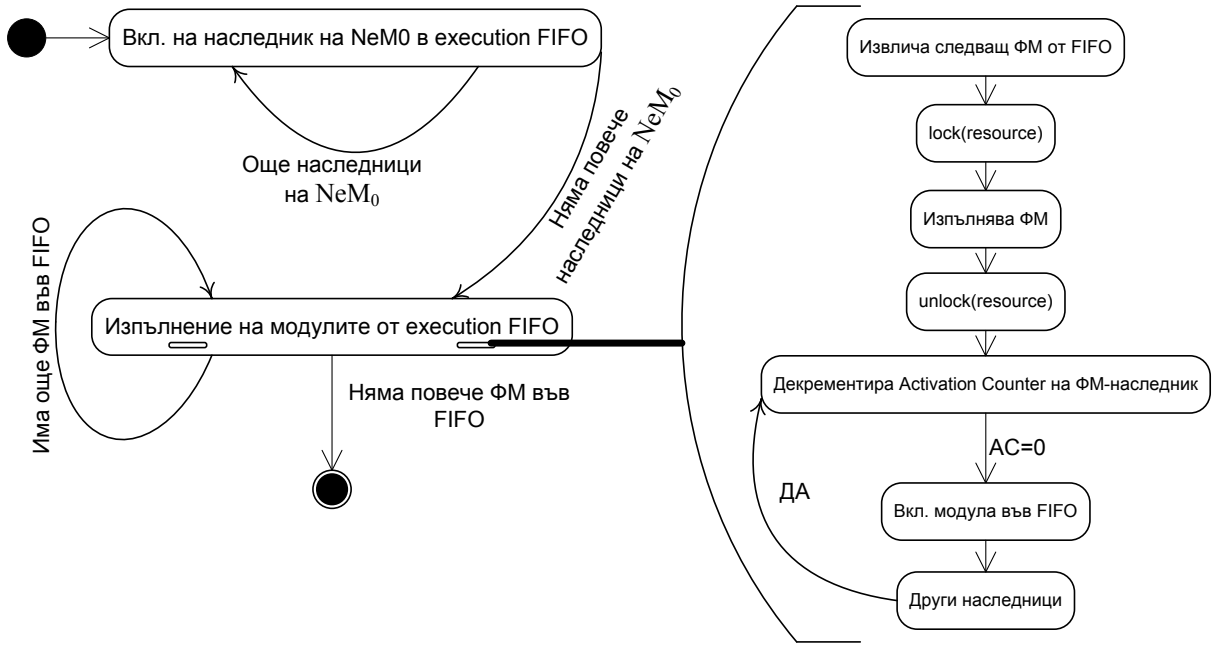
- R_type = 4 - това е външна реакция, реализирана от дъщерна задача или набор от такива задачи, заявявани/ изпълнявани в

текущото състояние. В този случай новата задача се заявява за изпълнение с помощта на командата на операционната система *enqueue*.

- R_type $\neq$ 4 - В този случаи направо се преминава към извикването на Интерпретатора на сигналния граф (Execution Sequence Manager). Ако реакцията е била R_type = 1, понеже този тип реакция предполага спонтанен преход към следващото състояние след завършване на работата по текущото, се преминава към новото състояние и цялата процедура започва наново.

➢ *Интерпретатор на сигналния граф (Execution Sequence Manager).*

Тази служебна програма представлява интерпретатор на сигналния граф на системата , се грижи за последователностна на изпълнение на функционалните модули. Тя пряко използва таблицата на сигналния граф (SFG) и таблицата на ФМ.



Фиг. 4.7. Интерпретатор на сигналния граф – диаграма на състоянията

Интерпретаторът на сигналния граф осигурява изпълнението на ФМ в последователността, зададена от сигналния граф. Последователността на работа е следната (фиг. 4.7):

Всички модули, които са начални за сигналния граф, се включват опашката за изпълнение (FIFO буфер). Основният цикъл се свежда по последователно извличане на функционални модули от FIFO и тяхното изпълнение. Преди изпълнението на всеки модул се заключва ресурса, който ще бъде зает от него (ако има такъв), а след приключването на ФМ същия се отключва. След изпълнението на функционалния модул се декрементират активиращите броячи на всички негови наследници. Ако за някой от тези ФМ бъде нулиран активиращият му брояч, той се включва в опашката за изпълнение. Работата на интерпретатора продължава до изчерпване на буфера.

4.3. Библиотеки на функционалните модули

Библиотеката на функционалните модули представлява набор от подпрограми, реализиращи типови функции на цифровите системи за управление. С помощта на ФМ могат да бъдат изграждани управляващи програми с различно предназначение.

Библиотека на функционалните модули за примерна РКСУ:

- Драйвери за физически вход/изход
  - ADC - аналогов входен драйвер
  - DAC – аналогов изходен драйвер
- Драйвери за отдалечен вход/изход
  - S_OUT
  - S_IN } асинхронна комуникация
  - C_OUT
  - C_NF } обмен на команди
  - C_IN
  - R_IN
  - R_NF } клиент-сървър комуникация
  - R_OUT
- Приложни функционални модули
  - LIN - линеаризация
  - FILT – експоненциален филтър
  - LC – проверка за долна и горна граници
  - GC – градиентна проверка
  - SPG – генератор на задаващо въздействие
  - PID – ПИД регулатор
  - LIM - ограничител по изход
  - ENC - флагов компаратор
- Функционални модули за сигнализация
  - S_R – сигнализация и заявяване на задача (signal_and_release)

Функционалните модули имат типова програмна конструкция, която се определя от унифицираната структура на данните, използвани от тях.

Функционалния модул се извиква с един единствен параметър - указател към ред от FM Instance Invocation Table. От този адрес с помощта на съответните отмествания могат да бъдат получени адресите на останалите дескриптори, респективно на клетките, съдържащи съответните адресни константи (указатели).

4.3.1. Приложни функционални модули

Тук са разгледани само по един модул от базовите типове. Спецификация на реализираните ФМ е включена в Приложение 1.

4.3.1.1. LIN – Функционален модул реализиращ линеаризация на технологичната величина.

Модулът е предназначен да преобразува показанието на датчика, като на изхода на модула технологичната величина е в линейна скала.

Алгоритъмът за работа е следния

$$y\sim(k) = a*(y(k) - y(k-1)) + b$$

- входни параметри

  y(k) - технологичната величина на входа на модула.

  a,b - коефициенти на линеаризация

- изходни параметри

  y~(k) - технологичната величина след линеаризацията

- работни параметри

  y(k-1) - стойност на технологичната величина в предния такт.

- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
Double	External	y(k)	1	1	I_1
Double	External	a	1	1	I_2
Double	External	b	1	1	I_3
Double	Local	y~(k)			O_1
Double	Local	y(k-1)	1	1	V_1

4.3.1.2. PID – Функционален модул реализиращ пропорционално-интегрален-диференциален закон за регулиране(ПИД регулатор)

Алгоритъм за работа

u(k)=(Kp+Ki+Kd)*eps(k) - Kd*eps(k-1) + V(k-1) +Uman
eps(k) = r(k) - y(k)

$$V(k-1) = Ki * \sum_{i=0}^{k-1} eps(i)$$

- входни параметри
  y(k) - технологичната величина на входа на модула.
  r(k) - заданието на регулатора
  Uman - ръчно задание
  Kp,Ki,Kd -коефициенти на ПИД регулатора

- изходни параметри
  u(k)- технологичната величина след линеаризацията

- работни параметри
  Eps(k-1) -грешката от миналия такт
  V(k-1) - интегрална компонента в закона на управление в предишния такт

- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
double	External	y(k)	1	1	I_1
double	External	r(k)	1	1	I_2
double	External	Uman	1	1	I_3
double	External	Kp	1	1	I_4
double	External	Ki	1	1	I_5
double	External	Kd	1	1	I_6
double	Local	u(k)	1	1	O_1
double	Local	Eps(k-1)	1	1	V_1
double	Local	V(k-1)	1	1	V_2

4.3.2. Драйвери за физически вход/изход

- ADC - Функционален модул реализиращ аналогово-цифрово преобразувана
  - входни параметри (АЦП)
    chanel - номер на канал на АЦП
    y(k) - технологичната величина след преобразуването
  - Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
int	Local	chanel	1	1	I_1
double	Local	y(k)	1	1	O_1

- DAC – Функционален модул реализиращ цифрово-аналогово преобразуване
  - входни параметри
    chanel - номер на канал на АЦП
    y(k) - технологичната величина
  - Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
Int	Local	chanel	1	1	I_1
Double	External	y(k)	1	1	I_2

Резултати:

- Предложена е идеология за реализация на съставни (композитни) ФМ
- Предложена е структура на системните таблици и структура на интерпретивните програми за автомати и трансформационни структури
- Предложена е структура на основните системни програми, адекватна на табличното представяне на структурата на ФБ
- Обоснован е и е предложен набор от ФМ, които осигуряват основно множество от библиотечни програми, поддържащи както непрекъснати, така и дискретни системи, вкл. и специализирани комуникационни и други поддържащи елементи

Гл. 5. Програмен генератор

В глава 5 се разглежда структурата и начина на функциониране на програмен генератор на РКСУ, отговарящ на формулираните в гл. 2. изисквания.

Програмният генератор (ПГ) е основен елемент от средата за проектиране на разпределени компютърни системи за управление.

ПГ има две части:

- генератор на управляващи програми (конфигуратор на РКСУ)
- система за оперативно конфигуриране и визуализация.

5.1. Изисквания към програмния генератор

Изискванията към програмния генератор като елемент от средата за проектиране на РКСУ бяха формулирани в гл. 2. Те са следните:

5.1.1. Възприема спецификацията на системата за управление.

5.1.2. Възприема физическата структура на системата и трансформира спецификацията до физическата конструкция.

5.1.3. Поражда таблици с параметри за различните елементи на средата (framework).

5.1.4. Прави тестове на конкретни физически елементи от системата, с цел установяване на времената на изпълнение на програми от възлите и комуникационните подсистеми.

5.1.5. На базата на (5.1.2,3,4) се прави анализ на диспечируемостта на системата.

5.1.6. Определя структурите с данни, до които има конкурентен достъп. Определя тези данни като ресурси и включва информация за това към таблиците на средата с цел автоматична защита на общите данни в “lock-unlock” от елементите на средата.

5.1.7. Определя данните с конкурентен достъп между елементите на средата с поведение на задача (task) на ОС и тези с поведение на ISR (interrupt service routines), формира информация за това за средата с цел защита на конкурентните данни в участъци със забранени прекъсвания.

5.1.8. ПГ възприема изискванията за тактове на дискретизация, максимално време за изпълнение на ФБ, на отделните сигнални трансформации

5.1.9. ПГ поражда физическа структура на системата със следните форми на активизиране на процесите

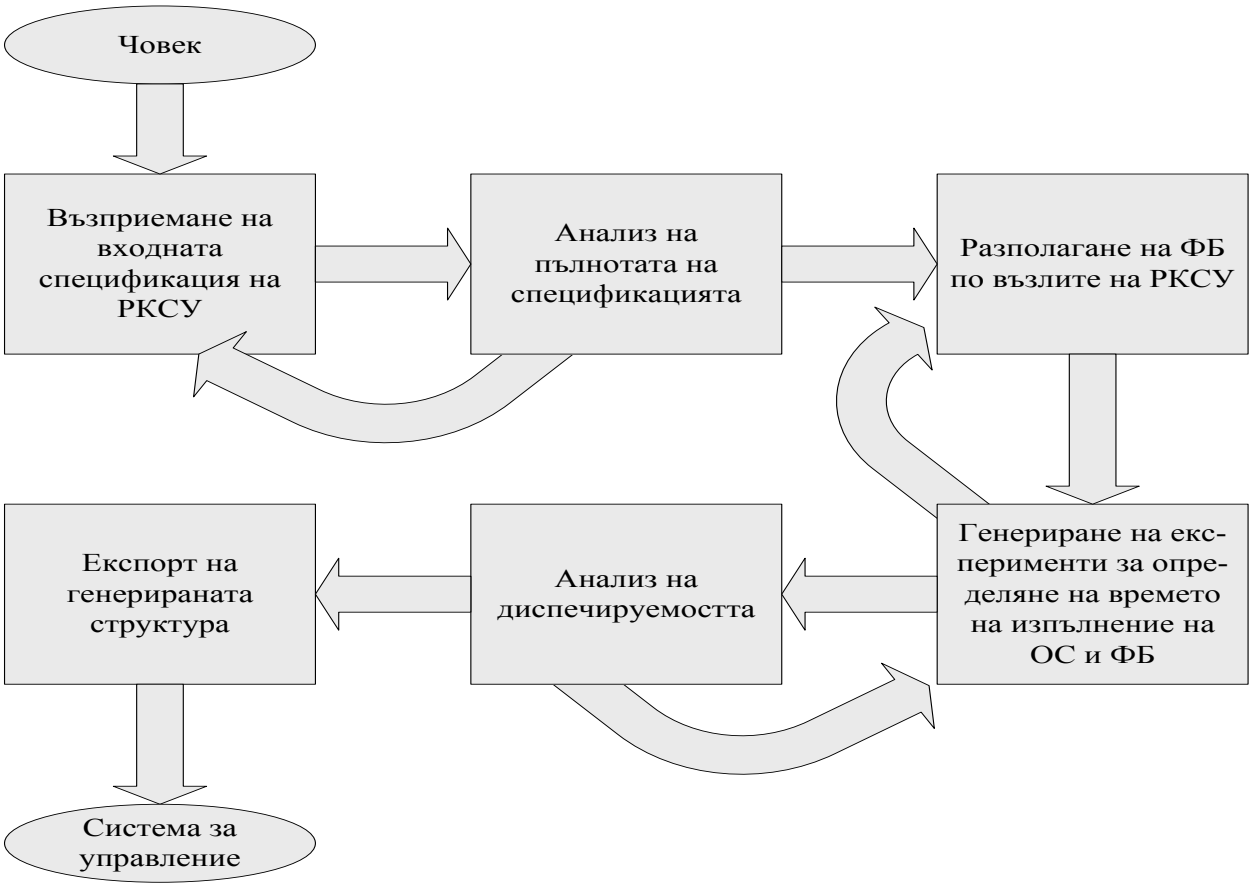
- периодично
- периодично с отложено активизиране
- периодично за първия и по събитие за следващите от трансформационната верига
- периодично в последователни тактове за задачите от една трансформационна верига (конвейерно активиране).

5.1.10. ПГ определя за всеки възел от физическата мрежа колко пъти ще се заяви като отделна задача интерпретатора на графа на състоянията. Определя приоритетите на тези заявки, ΔT и др. свързани с това елементи на конфигурацията на ОСРВ.

Конфигураторът на РКСУ е предназначен да възприема входната спецификация на РКСУ, да я анализира за валидност и да я превърне в набор от таблици (оперативна база данни), предназначена за експортиране към управляващата система. Освен това той изпълнява редица други задачи, свързани с разполагането на конструираната КСУ върху реалната разпределена

компютърна система; при наличие на връзка с управляващата система - генериране на команди за измервания на времето за работа (execution time) на елементи от операционната система, комуникационната подсистема и отделните ФБ и елементи от тях. На тази база се извършва анализ на диспечируемостта на РКСУ. При липса на връзка с управляващата система, анализ на диспечируемостта се извършва на базата на външни данни от експерименти. Генерираната таблична структура на (Р)КСУ се експортира в подходящ файлов вид или при наличие на връзка с реалната система – направо към нея за инициализация на управляващите компютри.

Общата структура на ПГ е показана на фиг. 5.1.



Фиг. 5.1. Обща структура на ПГ

5.2. Етапи на конфигурирането на РКСУ

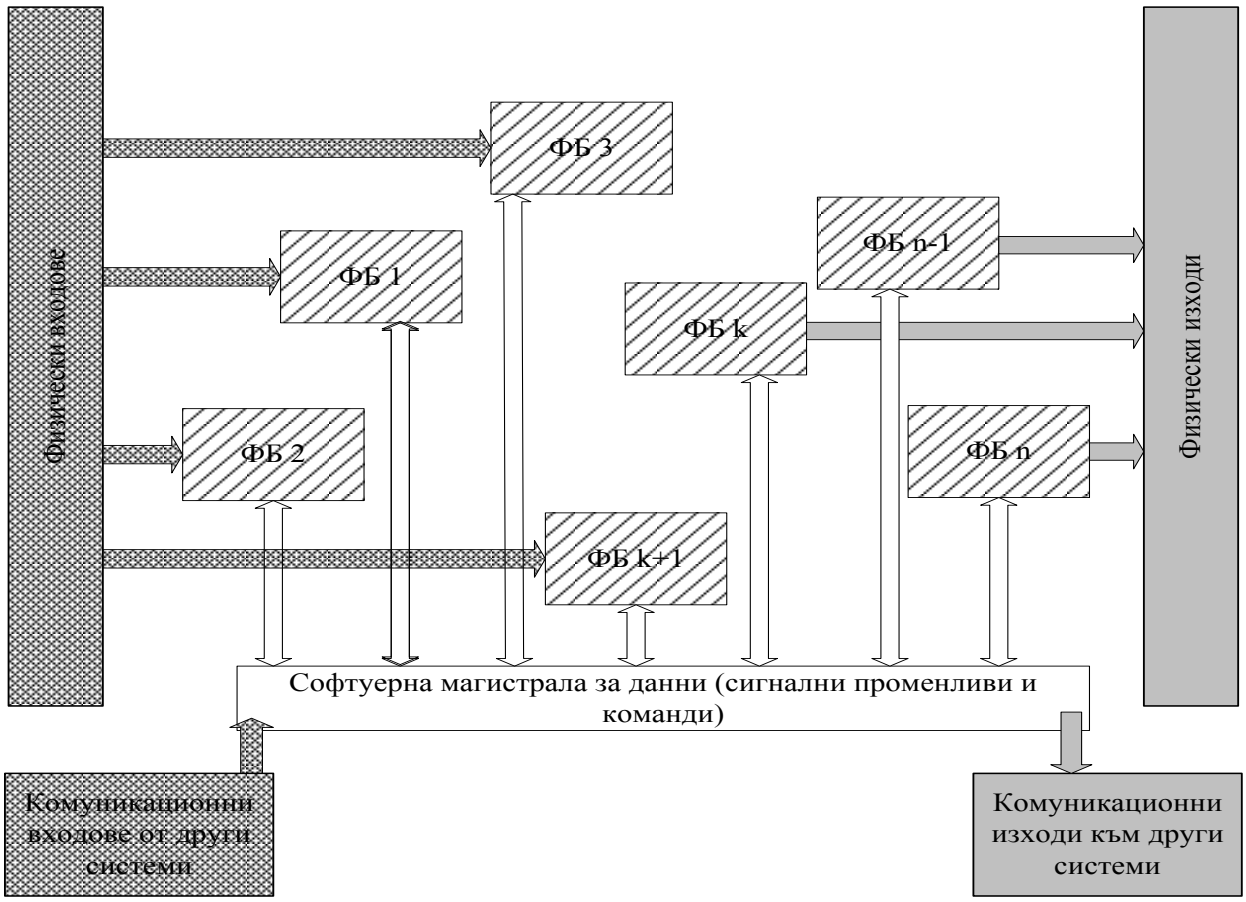
Главен проблем при конфигурирането на разпределена система за управление се явява необходимостта на ранен етап от създаването на системата да започне обвързване на решението с физическата ѝ структура. Това обикновено е причинено от изискването за обвързване с физическите входно/изходни устройства. Предлаганата структура на ПГ е освободена от този недостатък. При конфигурирането на системата началният модел се базира на идеята за конструиране на виртуална еднокомпютърна система за управление. Процесните (физически) връзки са регламентирани само като тип и наименование (пр. целочислена, „Положение по X”). Това позволява няколко възможности:

- структурата на системата за управление се създава без отчитане на неудобствата, които създава компютърната мрежа при този процес;
- създадената система е инвариантна на компютърната мрежа, върху която ще бъде наложена;

- осигурява субективното предимство да не натоварва ненужно инженера по автоматика с относително непривични за него дейности като разпределяне на изчислителни процеси в мрежа от компютри и пр.

При този процес на конфигуриране се поражда обобщена структура на системата за управление, представена като набор взаимодействащи си функционални блокове. ФБ, които обменят помежду си сигнали (системни съобщения). Както беше посочено в гл. 2 и 3, ФБ получава входна информация от физически входни устройства и от други ФБ и предава информация към физически изходни устройства и други ФБ. Местоположението на ФБ в мрежата е инвариантно спрямо логиката на реализираната КСУ. Това местоположение започва да оказва влияние едва когато се премине към физическа реализация.

Структура на КСУ, създадена по предложения начин, е показана на фиг.5.2.

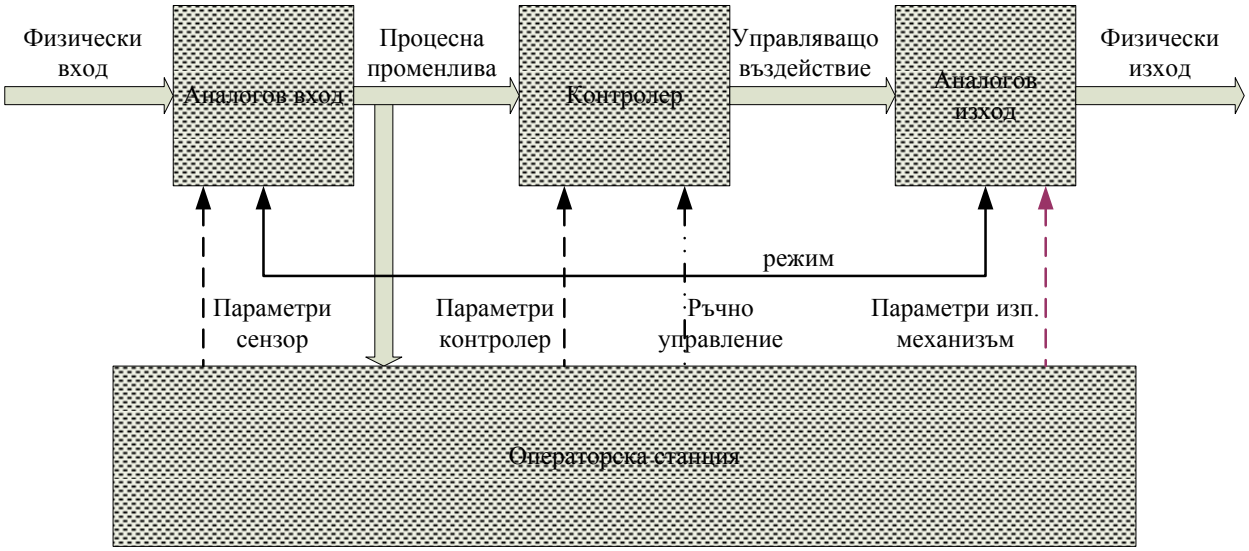


Фиг. 5.2. Обща структура на КСУ

Създаването на структурата на КСУ е на два етапа:

- Логическо дефиниране на входовете и изходите на (Р)КСУ. На началния етап на проектирането това е само дефиниране на имената и типовете им, подобен на вече показаната декларация на процесни връзки във ФБ – напр. „Температура/реално” или „Стоп/логически”. Входовете и изходите, които свързват (Р)КСУ с други системи се описват по същия формален начин, както физическите връзки. Асоциирането им с комуникационните входове/изходи на системата става на по-късни следващи етапи.
- Конфигуриране на функционалните блокове, които изграждат КСУ (или използуване на вече готови библиотечни ФБ) и тяхното параметризиране. Конфигуриране на входно/изходните връзки на функционалните блокове

Пример за подобен формален подход при структурирането на КСУ е показан на фиг. 5.3.



Фиг. 5.3. Формално конструиране на КСУ от функционални блокове

Неформално описание на конфигурирането на един ФБ може да бъде направено по следния начин:

- декларираме на събитията, които трябва да бъдат отработвани от функционалния блок (т.е. определяне на събитията, на които се реагира) – това са сигнали за изминали интервали от време и модификация на определени входни променливи.
- Конструиране на нишка на функция от ФБ, реализираща реакцията на всяко събитие.

Асоциирането на събитие със съответната нишка на функция е формален механизъм от вида:

„изтекъл интервал от 2 ms” → „Управление по оси X и Y”.

Физическата форма на тази асоциация зависи от операционната среда на управляващата система. Подобно на гл. 3 и 4, по-надолу в тази глава ще бъде разгледана версия на ПГ, съобразена с ОСРВ HARTEX. Възможно е адаптиране и към други операционни среди.

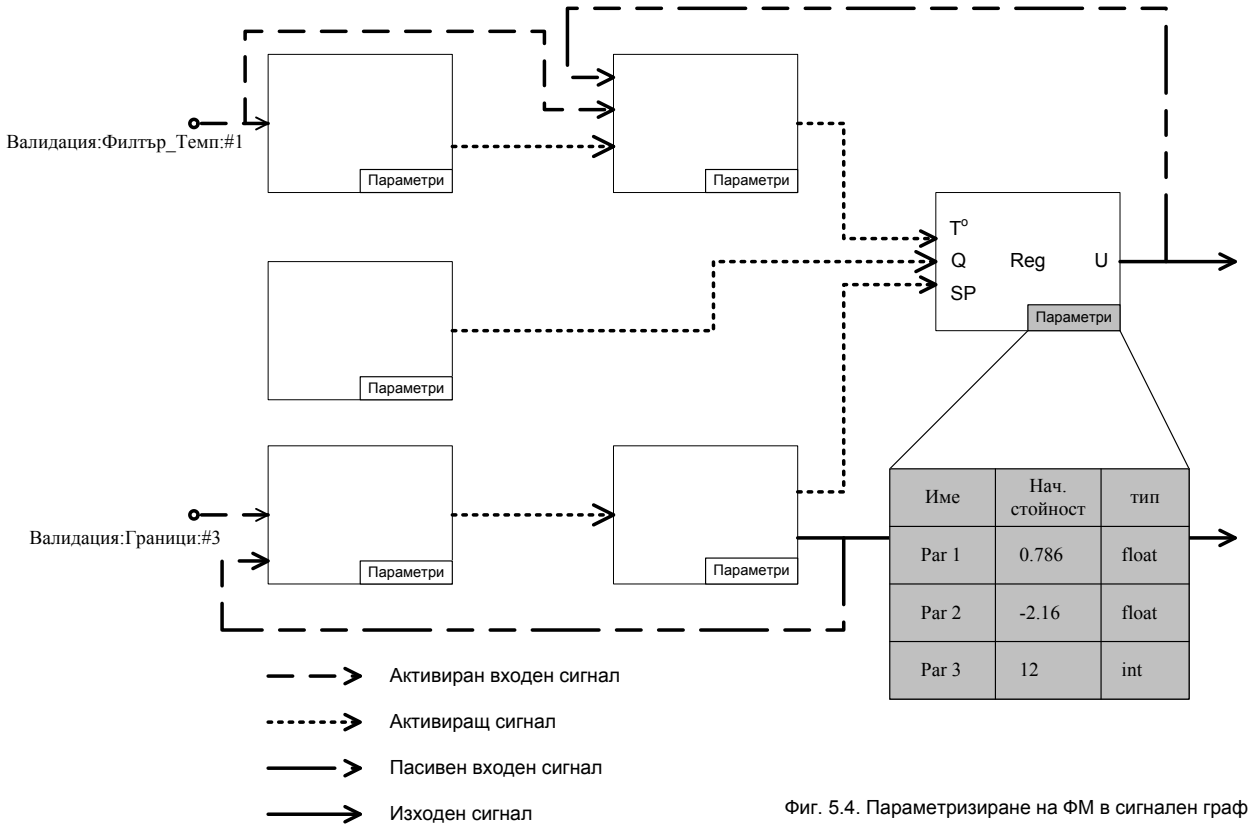
Конструирането на функционалната нишка (activity thread) е процес, при който се специфицира графа на реакцията и сигналните трансформации, асоциирани с отделни възли от него. Процедурата не започва по вече известните начини „отдолу-нагоре”, а в обратната посока – „отгоре-надолу”. Дефинират се възлите в графа. Всеки възел има уникално име (имената са произволен текст на български или английски език). Той отговаря на определено състояние на управляващия автомат.

За всяко състояние се определя какво действие ще се извършва в него – сигнална трансформация (една или няколко) или изпълнение на подчинен граф (виж гл. 4). След това се определят неговите състояния -наследници. Дефинират се предикатите, определящи валидността на даден преход. На този етап предикатите се дефинират само като конюнкции от формални величини (без посочен източник). Имената на формалните величини се задават по същия начин

като имената на възлите в графа. При необходимост се включва и логическата променлива „tick”. Тук се има предвид наличие на постъпил сигнал от източник на синхросигнал (когато активиращото събитие на нишката е периодичен сигнал). Както беше разгледано в гл. 4, това позволява реализиране на синхронно-асинхронни графове (виж типовете реакции).

- Когато действието, реализиращо функционалността на един възел (състояние), е подчинен граф, неговото конструиране протича по същия начин. Това позволява теоретически неограничена дълбочина на влагане на графовете един в друг. Реално ограничение е само обема на паметта на системата за управление.
- Когато действието, реализиращо функционалността на един възел, е сигнална трансформация (една или няколко), се конфигурира съответната поредица (поредици) от функционални модули (ФМ), които реализират сигналната трансформация (трансформации). Както беше посочено в гл. 4, това е идентично на създаване на съставен функционален модул.

Конфигурирането става, като се задава набор от ФМ, техните параметри и информационни връзки. Един пример е показан на фиг. 5.4.



Фиг. 5.4. Параметризиране на ФМ в сигнален граф

Както беше посочено в гл. 2 и 4, изпълнението на сигналните трансформации става по принципа „поток на данните” (data flow). Това налага различаването на няколко типа входни сигнали:

- Активиран – това са данни, чието значение се счита за вече готово за употреба.
- Активиращ – това са данни, които са готови за употреба след изпълнение на ФМ, който го генерира.
- Пасивен – това са данни, които се счита, че имат някакво значение, но чието изменение не се контролира в тази трансформация.

Освен това входовете и изходите биват:

- Външни – получават (предават) данни от (към) ФМ извън тази трансформация или са свързани към входове (изходи) на ФБ;
- Вътрешни – свързват вътрешни за трансформацията ФМ

Един ФМ от веригата може да бъде изпълнен при следните условия:

- Когато няма входове
- Входовете му са от тип „активиран” или „пасивен”
- Когато „активиращи”-те му входове вече са били инициализирани (т.е. ФМ, на които те са изходи, вече са изпълнени).

Една по-сложна дефиниция на потока на данните позволява да се определи, че потокът им може да бъде променлив в различни активизирания на сигналната трансформация и за един ФМ може да се дефинира, че е готов за изпълнение, ако един от определен набор входове е активизиран (виж гл. 4). Допълнително разширение е възможността да се управлява кой от този набор входове да се счита за активизиращ в определен момент (т.е. да се игнорират останалите, независимо дали има поток от данни към тях).

Най-пълно (и най-сложно за формално определяне, формална верификация и реализация) е решението, при което един ФМ може да има входове, които са активиращи и такива, които образуват група от активиращи входове (един от много) с управление на избора на активиращ вход.

При специфицирането на потока на данните се контролират типовете на входовете и изходите при съответните връзки.

При случай, когато към един възел на графа има асоциирани повече от една сигнална трансформации, те се считат за независими помежду си, т.е. редът на изпълнението им е без значение.

Конфигурирането на всички сигнални трансформации, асоциирани с възлите на графа на съответната нишка, завършва с асоциирането на формалните аргументи на предикатите на преходите с конкретни източници на булеви величини. Това могат да бъдат входове на ФБ от булев тип или съответно – булеви изходи на ФМ от различни сигнали трансформации.

5.3. Верификация на графовете на нишка на функция от ФБ (activity thread)

След специфицирането на всички елементи на графа се верифицира структурата му. Тази верификация е от няколко различни типа.

- Верификация на предикатите на преходите.

Наборът от предикати, определящи разрешенията за преход от даден възел към някой от неговите наследници трябва да бъде пълен и непротиворечив. Това означава, че при произволен набор от значения на операндите на предикатите, един и само един предикат може да има резултат „Истина”. Анализът се извършва, като се построява дървото на преходите на изчисление на предикатите (построява се BDD). За целта се използува подход, разгледан в [120].

На този етап се откриват и асоциират и всички операнди на предикатите, които не са били асоциирани със съответни източници на булеви сигнали.

- Верификация на структурата на графовете на нишка на функция

Верификацията на графовете на нишката е свързано с два анализа – на графа на състоянията и на графовете на сигналните трансформации, асоциирани

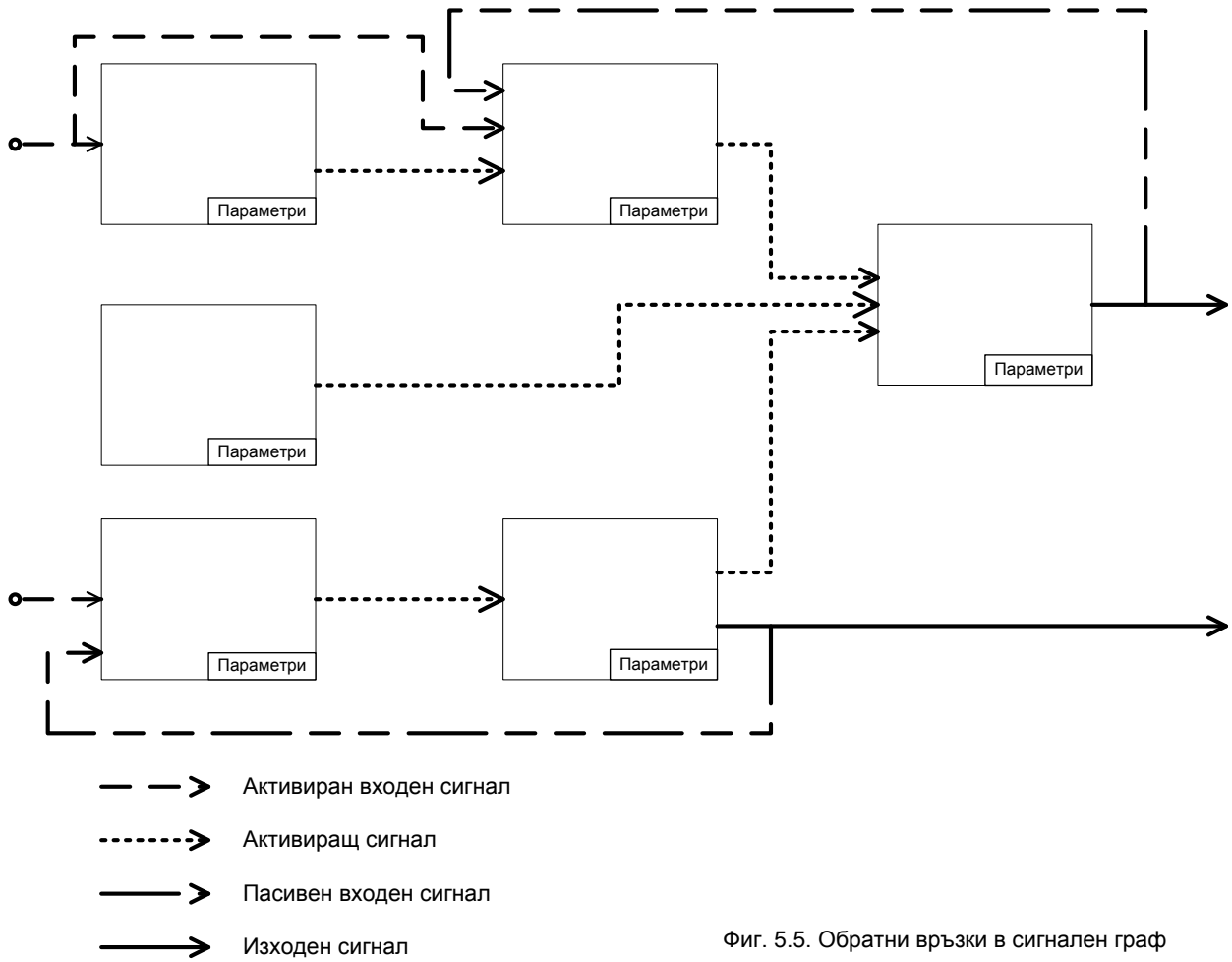
с отделните възли на графа на състоянията, като и за двата типа графове е недопустимо наличието на цикли от определен вид.

За графа на състоянията е недопустимо наличие на цикли, при които всички преходи са *асинхронни* (*асинхронен* е преход, чието изпълнение не зависи от тактов “synchro” сигнал и не е отбелязан като отложен преход). Това ограничение се налага с цел да не може да се получи безкраен и непрекъснат цикъл на изпълнение (оттам и монополизиране на процесора). Анализът се прави, като се използва модификация на базовия алгоритъм на Флойд за намиране на най-къс път в граф. Описания на модификацията е разгледано по-надолу в тази глава.

5.4. Верификация на сигнална трансформация

Верификация на сигналната трансформация е процес, при който се проверява валидността (непрекъснатостта) на сигналните вериги. Това става чрез следната процедура:

- Всеки вход на ФМ трябва да бъде свързан към съответен източник на сигнал, като типовете на обменяните данни трябва да са идентични;
- Всеки вход на ФБ трябва да бъде асоциран към операнд на предикат в графа на състоянията и/или свързан към вход на ФМ от поне една сигнална трансформация;
- Всеки изход на ФБ трябва да е свързан към изход на ФМ от една и само една сигнална трансформация;
- При наличие на информационни цикли в сигнален граф, те трябва да включват връзка от пасивен тип между поне два ФМ от цикъла, т.е. няма възможност от безкрайно ре-инициализиране на изпълнението ма ФМ. Този анализ се извършва отново на базата на алгоритъма за намиране на цикли в граф на базата на модифицирания алгоритъм на Флойд. Пример е разгледан на фиг. 5.5.



Фиг. 5.5. Обратни връзки в сигнален граф

Началната стойност на всеки изход за всеки ФМ може да бъде определена при конфигурирането на съответната сигнална трансформация, в която е включен съответния ФМ.

Тези процедури по верификация се повтарят за всяка нишка на функционалния блок.

След верифициране на елементите на ФБ, се валидират неговите входове и изходи. Всъщност това означава, че се установява наличието на връзка между всеки вход на ФБ и вход на ФМ (един или няколко) от вече специфицираните сигнални трансформации.

5.5. Свързване на вътрешните и външните източници на данни

Последен процес от вътрешната спецификация на ФБ е свързване на формалните източници на активиращи събития за функционалните нишки към изходи на ФМ (за тези, събития, чиито източници са вътрешни за ФБ).

Този подход за конфигуриране се прилага за всички планирани ФБ на (Р)КСУ.

Последното действие от формалното конфигуриране на системата за управление е свързване на входовете и изходите на ФБ помежду им и към физическите входове и изходи на КСУ. Както беше посочено по-горе, на този етап физическите входове са само дефинирани формално чрез двойката <име|тип> и не са локализирани физически.

При тази процедура на свързване се провежда и валидиране на връзките по тип за всяка двойка изход-вход. За всяка данна се посочва периода ѝ на

генериране и времето ѝ за валидност (deadline). Това е необходимо за последващите анализи на диспечируемостта на комуникацията.

5.6. Модификация на алгоритъма на Флойд за установяване на наличието на цикли в граф на състоянията и граф на сигнална трансформация

Базовата версия на алгоритъма на Флойд е предназначена за намиране на най-кратките пътища между всеки два възела в един граф. Алгоритъмът работи еднакво добре за ориентирани и неориентирани графове. Работи се с матрица на връзките, като стойностите на всеки елемент a_{ij} показват теглото на пътя от възел *i* до възел *j*. Тук графът на преходите и сигналният граф са ориентирани.

Модификацията, която се предлага тук се състои в следното:

- За тегло на дъгите се приема 1 – поради това, че дъгите нямат собствено тегло извън контекста „има/няма връзка".
- За ограничител на теглото на пътя (т.е. „липса на път") при вече направеното ограничение на теглото на дъгите се приема W=N+1, където N е броя на възлите на графа. Това допускане е справедливо, защото не може да има прост път, който включва повече от един път всеки възел, а те са N на брой.
- В резултат от работата на основната версия на алгоритъма на Флойд по главния диагонал на матрицата на връзките се получават нулеви стойности, т.е. пътят от един възел до самия него е 0. Всъщност, при анализ на решението се вижда, че в една негова междинна стъпка по главния диагонал се получават стойности, които съответстват на най-малките тегла (дължини) на цикли, на които принадлежи съответният елемент a_{ii} . Това произлиза от оригиналното доказателство на метода, който по същество е спускане в дълбочина от елемент *i* до срещане на търсения елемент *j*, като не се прави изключение за точката на тръгване. Така, ако се елиминира изкуственото маркиране на дължините на пътищата a_{ii} като нулеви, се получава възможност за анализ на графа за наличие на цикли.
- Анализът за наличие на цикли се основава на следното:
    - Всеки елемент a_{ii}, който има стойност, различна от N+1 (т.е. ограничителя за празен път), участвува в поне един цикъл.
    - Елементите a_i и a_j участват в един и същи цикъл, ако a_{ij} и a_{ji} са ненулеви, т.е. има пътища между двата елемента и в двете посоки.

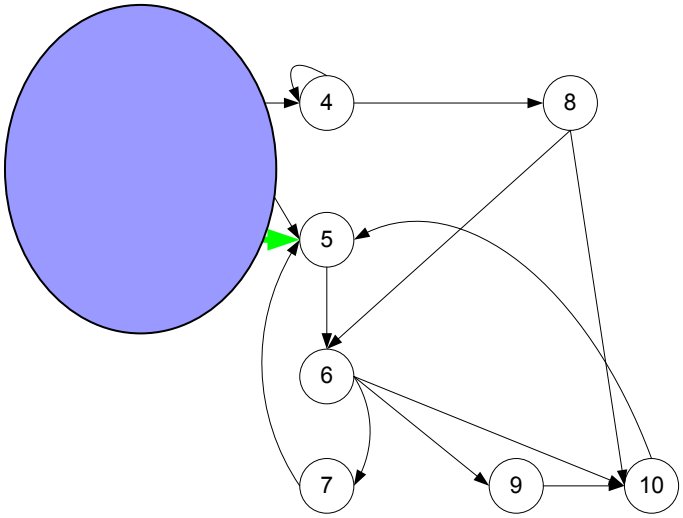
Извод – в анализирания граф (на преходите или на сигнална трансформация) има поне един цикъл, ако има елементи, разположени на главния диагонал, които имат минимални дължини на пътищата до самите себе L<N+1.

- При необходимост може да се направи пълно изследване дали всеки елемент, който участва в цикъл, не участва в повече от един цикъл. Може и да се определи точно с кои елементи в кои цикли участва. Този анализ са основава на посочените по-горе базови елементи на анализа. С оглед на факта, че за системата за управление е необходимо само да се знае, че **няма** цикли и като се има предвид, че сложността на този анализ е от порядък $O(n^2)$, този анализ се прави само като специална функция на ПГ .
- Общата сложност на алгоритъма на Флойд е $O(n^3)$. Това налага предварително да се направи опит за минимизиране на анализирания граф.
    - *Минимизиране на графа на преходите*: от оригиналния граф на преходите се премахват всички възли, преходите към които са

маркирани като синхронни или с отложено активизиране (т.е. изпълнявани в последователни тактове на работа на управляващата система). Не може да се прави предварително предположение за обема на минимизацията на графа, защото тя зависи от конкретната конфигурация, но в случая на чисто синхронен автомат оригиналният граф се свежда по празен граф. Сложността на тази минимизация е *O(n),* което оправдава нейното приложение.

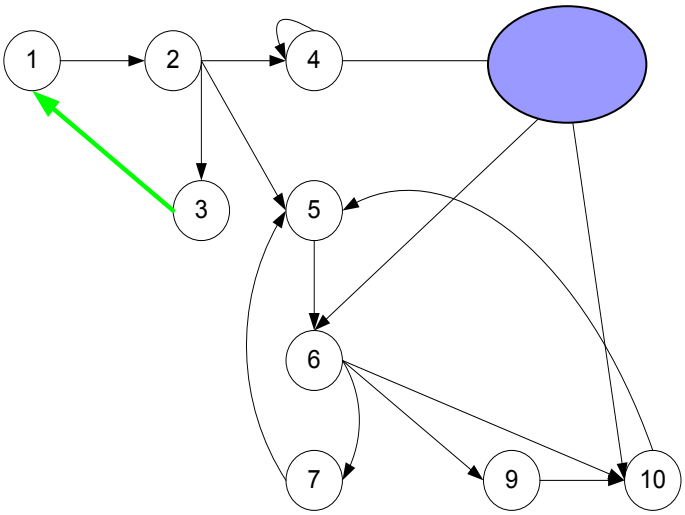
- *Минимизиране на сигнална трансформация*: при сигналните трансформации пасивните връзки се елиминират от графа. Независимо колко реални активиращи връзки има между два ФМ, връзката между тях се редуцира до единична. Сложността на тази минимизация е *O(n).*

Пример на установяване на наличието на цикли в графа на преходите е показано на фиг. 5.6 а и 5.6 б.



*	1	2	2	2	3	4	3	4	4
*	*	1	1	1	2	3	2	3	3
*	*	*	*	1	2	3	*	3	3
*	*	*	1	3	2	3	1	3	2
*	*	*	*	3	1	3	*	2	2
*	*	*	*	2	3	1	*	1	1
*	*	*	*	1	2	3	*	3	3
*	*	*	*	2	1	2	*	2	1
*	*	*	*	2	3	4	*	4	1
*	*	*	*	1	2	3	*	3	3

Фиг. 5.6 а. Граф на състоянията



3	1	2	2	2	3	4	3	4	4
2	3	1	1	1	2	3	2	3	4
1	2	3	3	3	4	5	4	5	5
*	*	*	1	3	2	3	1	3	2
*	*	*	*	3	1	3	*	2	2
*	*	*	*	2	3	1	*	1	1
*	*	*	*	1	2	3	*	3	3
*	*	*	*	2	1	2	*	2	1
*	*	*	*	2	3	4	*	4	1
*	*	*	*	1	2	3	*	3	3

Фиг. 5.6 б. Граф на състоянията с допълнителен цикъл

Разполагане на функционалните блокове по възлите на РКСУ

При всяка разпределена компютърна система възниква въпроса с разполагане на изчислителните компоненти по възлите на системата. Когато системата работи в реално време, проблемите се усложняват от два фактора:

- Променливите на състоянието имат определено време на валидност, което повишава изискванията към комуникационната подсистема с цел доставяне на съобщенията навреме;
- Времето за реакция на определени външни и вътрешни събития в общия случай е ограничено – типична работа в реално време.

Процедурата по разполагане на ФБ включва следните два начални етапа:

- Асоциране на формално дефинираните входове и изходи на РКСУ към физически входове и изходи, разположени в съответните възли. За всеки вход и изход се параметризира и реалния му източник – устройство (вх./изх. карта, номер на канал от нея, диапазон и пр).
- Конфигуриране на комуникационните входове и изходи на РКСУ. Има се предвид тези входове и изходи, които свързват системата с други системи извън обхвата на генерираната РКСУ. При направеното в началото на тази работа предположение, че се използува комуникация тип CAN, независимо от реалната физическа среда, асоциирането на посочените входове и изходи с комуникационен поток е достатъчно. Единственото допълнително изискване е асоциране на реални CAN идентификатори към формалните им имена. Това може да бъде направено ръчно или автоматизирано. Ръчното присвояване на идентификатор е твърде неспецифична задача за специалист по автоматика и е непрепоръчително. По-нататък в тази глава ще бъде предложен метод за автоматично присвояване на идентификатори на комуникационните обекти.

След като бъдат специфицирани общосистемните входове и изходи, се поставя въпроса за физическото разполагане на ФБ по възлите. Целта на процеса е постигане на интегрирана диспечируемост на РКСУ, т.е. диспечируемост на изчислителните процеси във всеки възел и на потока от съобщения между тях.

Най-елементарното на пръв поглед решение се свежда до следното:

- ФБ с най-дълги изчисления да се разположат във възли с най-висока производителност.
- Да се минимизира трафика по свързващата мрежа.

В повечето случаи този начален подход е недостатъчен. Това се определя от факта, че ако се следва първото предложение, всички ФБ ще бъдат разположени в един възел. Това удовлетворява в много отношения и второто изискване, но се пораждат следните проблеми:

- Даже и при положение, че в избраният възел може да бъдат разположени всички изчислителни процеси, това определено е неоптимално, защото остава неизползувана изчислителна мощност в другите възли. Освен това може да се стигне до изчислително претоварване на възела.
- Разкъсва се връзката с физическите входове и изходи на РКСУ, разположени в други възли и ФБ в конкретния възел.

На практика опитите да се минимизира комуникационния трафик водят до разполагане на много ФБ в един възел, което затруднява диспечируемостта на процесите. Обратно – при разпределяне на процесите по множество възли се повишава комуникацията между тях. Така наличието на две взаимно-

противоречиви изисквания значително затруднява процедурата по разполагането на ФБ по възлите на мрежата. Втори проблем е, че много често не се разполага с точни данни за реалната производителност на възлите и пропускателната способност на комуникационната подсистема. В условия на търсене на оптимално решение се поражда задача за максимизиране на сложна функция.

Облекчаващо е, че не се търси оптимално решение в смисъл на максимизиране на критерий „производителност на цялата РКСУ", а само конфигурация, която е диспечируема, т.е. отговаря на ограниченията по време (deadlines) за сигнали (данни) и реакции. Търсенето на субоптимално решение може да бъде направено по следния разгледан по-долу начин.

В общия случай един ФБ не може да бъде разположен във възел, в който са разположени всички физически входове и изходи, към които е свързан (поради елементарното съображение, че те може да не са групирани в един възел). Това предполага наличието на възможност физическите входни данни от един възел да се изпращат по комуникационната подсистема на РКСУ. По същия начин данните за извеждане през физическите изходи трябва да може да се получават чрез комуникационната подсистема. Този проблем е решен със съществуването на входен и съответно изходен обобщен драйвер (вж. гл. 3,4), които интегрират специфичните входни и изходни драйвери и от гледна точка на всеки възел работят като системни предварително конфигурирани ФБ. Това позволява асоциирането им с комуникационната подсистема да бъде на същия принцип, както на обикновените ФБ. Описаната по-горе процедура по свързване на формалните входове и изходи на РКСУ с физическите представлява на практика и процедурата по конфигуриране на входните и изходни обобщени драйвери.

5.7. Процедура по разполагане на ФБ по възлите на РКСУ

- Начална стъпка - всеки ФБ се разполага във възел, към който има най-много физически връзки, т.е. получава и предава най-много данни към/от физическите драйвери.
- Функционални блокове, които нямат физически вход/изход, се разполагат по останалите възли равномерно. При наличие на предварителна информация за времето за изпълнение на отделни ФБ в специфични възли, се провежда предварителен анализ на диспечируемостта на възлите и ФБ се разполагат според тези резултати.
- На базата на вече направеното конфигуриране на входно/изходните връзки между ФБ се изготвя подредена таблица на съобщенията. Подредбата е по принципа „най-малкото време за валидност първо" (nearest deadline first). При съобщения с еднакви времена по-високо в таблицата се записва съобщение, което е регистрирано за получаване от процеси с по-малки времена за очакване (deadlines).
- Идентификаторите на съобщенията се асоциират според тази таблица. Така първите съобщения в нея получават най-висок приоритет. Всяко съобщение има уникален идентификатор. След това начално разполагане се изготвя каталог на всички данни, които трябва да преминат през комуникационната система, т.е. има обмен на данни между възли. За всеки възел се изготвя таблица на входните и изходните съобщения. Входната таблица служи за описаната в гл. 3 процедура на „разпознаване по съдържание" (acceptance filtering).

При наличие на началната информация за разполагане на ФБ по възлите на РКСУ и за конфигурацията на комуникацията следва процедура по експериментиране за определяне на реалната производителност на възлите

(време за изпълнение на всеки ФБ) и пропускателната способност на комуникационната подсистема.

5.8. Процедура по установяване на времето за изпълнение на ФБ и пропускателната способност на комуникационната подсистема.

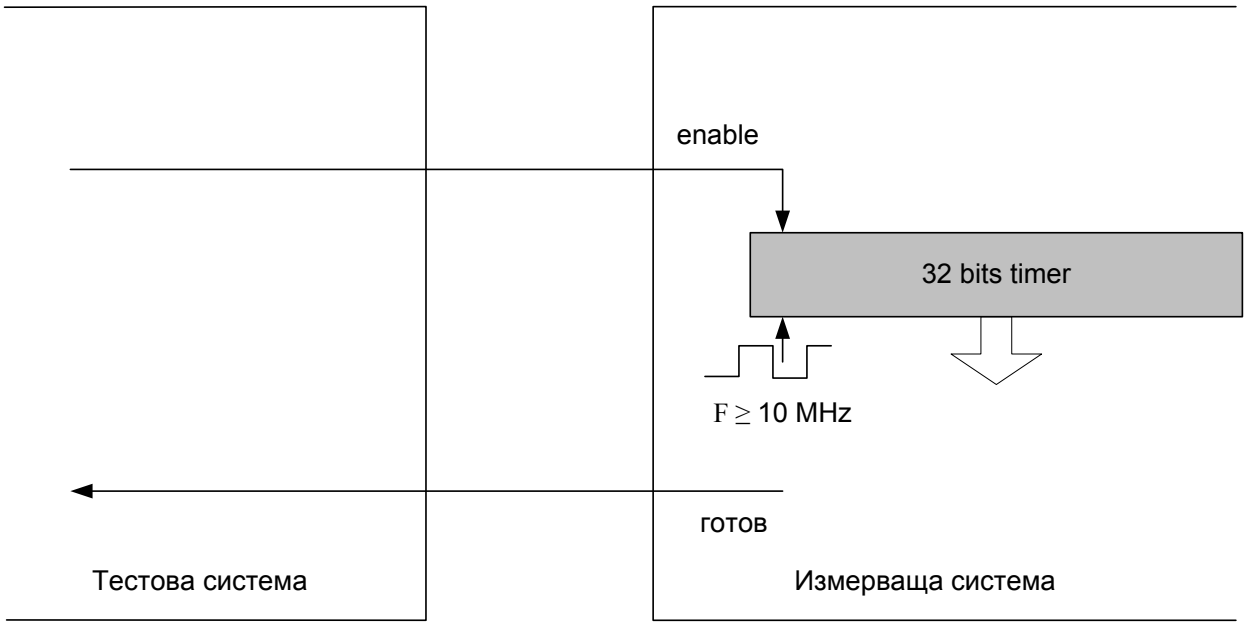
В областта на компютърния анализ въпроса за намиране на времето за работа (execution time) на програма е един от най-неприятните поради факта, че това време се влияе от голям и често пъти неопределен брой фактори. Освен видимата част, свързана с брой на изчисленията, цикли и разклонения съществува напълно необхващаема част от обръщения към библиотечни подпрограми, системни обръщения, влияние на кеш-паметта и конвейерната структура на процесора, прекъсвания и пр. Разглеждането, което ще бъде направено тук, се отнася до много частен случай на програми с предварително известно поведение. Даже и при тази ситуация ще бъдат направени редица допускания и приети ограничения с цел гарантиране на поведението на КСУ.

Всички разглеждания се правят при предположението, че използваната ОСРВ има детерминирано време за изпълнение. Това изискване се покрива напълно от ОСРВ HARTEX [71]. Това изискване е важно, защото премахва неопределеността при системните обръщения.

В общия случай отделните нишки се изпълняват физически от отделни нишки на ОСРВ. Тук възниква двойна употреба на термина „нишка”. Както беше разглеждано по-горе, „activity thread” е логическа конструкция, която описва реакцията на ФБ на определено събитие („стимул”), докато нишката на ОСРВ е просто самостоятелно диспечируема програмна единица. Ситуацията, когато една функционална нишка може да се асоциира към една програмна нишка на ОСРВ е много благоприятно, тъй като улеснява както анализа, така и реализацията на КСУ. ОСРВ HARTEX отговаря на това изискване. Това обстоятелство позволява анализа на ФБ да се разглежда разделно (всяка нишка поотделно). Тази декомпозиция намалява общата сложност на тестовете и анализите.

Времето за изпълнение на една нишка се намира чрез следната експериментална постановка.

- Експериментът се извършва на компютър, който е идентичен с този, на който реално ще бъде изпълнявана. Идентичността е както по отношение на хардуера, така и по отношение на инсталирания софтуер.
- Компютърът е екипиран с дискретна периферия, чрез която се управлява времеизмерващата апаратура.
- Времеизмерващата апаратура е разположена в отделен компютър. Тя включва многоразряден таймер (32 бита) с висока разрешаваща способност (0.1 μs или по-добра). Принципна схема на връзката между тествания компютър и измервателната среда е показана на фиг. 5.7.



Фиг. 5.7 Схема на таймер и връзка с тествания компютър

Тестващия софтуер работи по следната програмна схема:

start timer
repeat N times
    tested block
stop timer
read timer – T_1
start timer
repeat N times
stop timer
read timer – T_2
$T = (T_1 - T_2)/N$

Тази схема позволява да се намери осредненото време за изпълнение на даден програмен фрагмент, като при това се елиминира времето за работа на измерващата програма.

Нишката няма поведение на статичен програмен фрагмент. Това изисква усъвършенстване на процедурата за измерване на времето. Тази процедура включва следните етапи:

- Намиране на всички фрагменти от функционалната нишка, които се изпълняват асинхронно, т.е. при едно активизиране. За тази цел се изследва графа на нишката, като точките на разделение са възли, в чиито предикатни изрази участва синхро-маркера (tick). Получава се набор от фрагменти, чието време за изпълнение трябва да бъде намерено.
- За всеки фрагмент се построява пълното дърво на връзките в него. (Всички пътища са ациклични). За всеки път се формира набора от стойности на операндите на предикатните му изрази, за да се постигне гарантирано преминаване по него – на базата на BDD.
- За всеки фрагмент се извършват следните действия:
    - Реинициализират се таблиците, съдържащи двоичните диаграми на преходите (BDD), за да сочат към източници на логически операнди с гарантирана стойност, което гарантира преминаване по желана част от дървото на графа на нишката с начало в съответния възел.
    - Входната точка на графа се установява на желания възел.

- Изпълнява се процедурата по измерване на времето, посочена по-горе.
- Измереното време се запазва за по-нататъшни анализи.

- След изследване на времената за изпълнение на всички клонове от дървото на един фрагмент, най-голямото измерено време се приема за време за изпълнение на този фрагмент.

Графът се изследва фрагмент по фрагмент.

- Най-голямото от всички времена за изпълнение на фрагментите на нишката се приема за време за изпълнение на тази функционална нишка.

Тази оценка в редица случаи е песимистична, защото може да бъде направена на базата на времето за изпълнение на части от графа, които не се включват в нормалния режим на работа. За да се избегне това и за да се намали влиянието на песимистичните оценки, а и от чисто конструктивна гледна точка, е разумно конструирането на функционалните нишки да включва само елементи от стандартните реакции на съответните външни събития. При всички случаи обаче получената оценка дава възможност в последствие да се прави анализ на диспечируемостта на ФБ (или блокове), разположени в един възел.

Измерването на пропускателната способност на комуникационната мрежа изисква по-сложни експерименти. При всички разглеждания се предполага, че се работи с мрежа с детерминирано поведение, т.е. не е възможно възникване на неограничено във времето очакване за достъп до мрежата. Това предполага приоритетно подреждане на изходящите съобщения и детерминиран достъп до мрежата. Освен това се изисква и постоянно време за изпълнение на процедурата по разпознаване на входящите съобщения (acceptance filtering). При това положение общото транспортно време на едно съобщение е функция на времето за изчакване за предаване във възела t_d, времето за работа на комуникационната подсистема на възела t_{out}, времето за физически транспорт t_{tr}, времето за работа на комуникационната подсистема на възела-приемник t_{in}, време за предаване на съобщението до неговия получател във възела-приемник t_{rec}. Обект на това измерване е времето от момента на подаване на заявка за изпращане на съобщение до неговото получаване в точката за транспортиране до очакващата го програма $t = t_d + t_{out} + t_{tr} + t_{in}$. Пример за такова измерване и получените резултати са посочени в Приложение 2 към работата.

Както е посочено в [181], диспечируемостта на CAN мрежа може да бъде компрометирана при съответното неподходящо поведение на апаратния CAN-драйвер. Тъй като тук не се разискват специализирани изследвания по мрежи за реално време, ще приемем, че мрежовия хардуер има адекватно поведение и не индуцира неограничена приоритетна инверсия.

Транспортното време t_{tr} се приема, че започва от момента на подаване на композираното съобщение към комуникационния хардуер до активизиране на програмата за обработка на прекъсването (ПОП) в приемника на съобщението. Това време е извън всякакъв програмен контрол и действието може да се счита за атомарно от гледна точка на анализа. Измерванията се организират, като приемникът и предавателят са оборудвани с дискретна изходна периферия, която управлява стартирането и спирането на измервателната система, разположена в самостоятелен компютър.

Измерванията се правят по предложената по-горе схема с цел елиминиране на времето за работа на измерващите програми. Оценката, която се получава, е приблизителна, защото не позволява отчитането на закъснението за стартиране на програмата за обработка на прекъсването, породено от потенциално активни прекъсвания с по-висок приоритет. Може да бъде направен анализ и на това отлагане, но тук ще бъде направен компромис. Компромисът се базира на това, че времето за работа на програмите на комуникационната подсистема е съществено по-голямо от времето за работа на стартовите ПОП на по-високо приоритетните източници и оказва по-малко влияние от тях. Така се получава оценка без да се включва времето за изчакване на съобщението в мрежовия адаптер за физически транспорт. Поради различните методи за работа на различните мрежи е по-удачно това време да се счита като част от t_d.

Времето за изчакване на съобщението t_d включва времето от регистрирането му до подаването му към комуникационния хардуер. Това време може да се разгледа като сума от две времена – време за включване в изходящата опашка и пасивно изчакване за транспортиране. Времето за изчакване за транспортиране зависи от типа на мрежата (CAN, Profibus, RT Ethernet) и може да бъде оценено, като песимистичната оценка се добавя към измерените в идеални условия времена. Този тип оценки са известни [26][99] [100][180].

Времето за работа на комуникационната подсистема t_{out} зависи от начина за предаване на съобщението към нея и неговото регистриране за изпращане. Според типа на поддържаната изходна опашка времето за включване в нея може да бъде константно (идеален случай) и функция на метода за сортиране на опашката. При относително ограничен брой на изходящите съобщения (до няколкостотин) се прилага стратегия с фиксиран брой буфери за изходящи съобщения – по един за всяко съобщение и времето за регистриране на съобщението е константно. Това е справедливо допускане, защото за РКСУ е предварително известен броя на входящите и изходящи съобщения за всеки възел и той нормално не е „прекалено” голям. Общото време за работа на тази подсистема се измерва по вече посочения по-горе начин за измерване на времето за работа на програмен фрагмент.

Времето за работа на комуникационната подсистема на приемника t_{in} зависи основно от времето за извършване на разпознаването на съобщението за приемане (*acceptance filtering*) и времето за разполагане на съобщението за транспортиране до неговите приемници. Филтрирането на съобщенията при „малък” и предварително известен брой на съобщенията може да се извършва за фиксирано време на базата на битов вектор с големина, равна на броя на съобщенията в системата, в който са маркирани тези съобщения, които се приемат в съответния възел. Намирането става на базата на константно по време изчисление, базирано на идентификатора на съобщението, за намиране търсения бит в битовия вектор.

N_{byte} = MessageID/8

N_{bit} = MessageID%8.

Номерът на бита служи за намиране на маската на съответния бит от константен масив с маски. Номерът на байта служи за намиране на байта в масива, емулиращ битов вектор. Така след намиране на нужния байт и маска се проверява дали съобщението с идентификатор MessageID е част от входните съобщения на възела.

Регистрирането на съобщението за предаване до приемниците му най-често е функция на услугите, предоставени от ОС, като за разгледания случай на ОСРВ HARTEX, се поддържа оповестяване на базата на множествена паралелна сигнализация чрез битови семафори (виж гл. 3 и 4). Това време се измерва по същия начин, както и t_{out}.

Така общото време за комуникация се получава на базата на сумирането на тези времена. Времената за предаване на съобщения от един към друг възел се измерват за всяка двойка възли. Оценките за времената на изчакване на съобщенията за предаване се оценяват за всяко съобщение поотделно. Така за всяко съобщение се получава оценка дали е диспечируемо в рамките на системата. При извършване на реконфигурация на системата, ако не е променян типа и броя на възлите, се налага само извършване на преоценка на времената за изчакване на съобщенията и повторно изчисление на оценките за пълните времена за транспортирането им и оттам – тяхната диспечируемост.

Този подход позволява анализ на диспечируемостта на програмите, изпълнявани в отделните възли и диспечируемостта на мрежата. Предложеното решение предполага наличието на допълнителна апаратура за извършване на измерванията, което в определен смисъл затруднява приложението ѝ . Въпреки това тя е била приложена успешно [99]. При наличие на сравнително еднотипен хардуер броят на отделните експерименти се намалява значително, тъй като измерванията за бързодействие на основните подсистеми на ОСРВ ще важат за голям брой възли.

5.9. Реализации на системи за управление на базата на предложения формален и програмен модел

5.9.1. Система за управление на климата в тъкачен цех – завод „Дунавска коприна” Русе.

5.9.2. Система за управление на плазмена разкроечна машина за ламарина – ЗММ София.

5.9.3. Широкоформатен текстилен плотер – „Български разкроечни машини” ООД

5.9.4. Система за управление на парафинови вани – „Хънитек” ООД

5.9.5. Система за управление на машина за рециклиране на ротори на постоянно-токови електродвигатели – „M&T 2001” Limited

5.9.6. Симулатор на пристанищен кран – „Бесттехника” ООД

5.9.7. Система за управление на машина за обработка на колбаси – „Хънитек” ООД.

Резултати:

- Разгледана е структурата на програмен генератор на РКСУ.
- Предложена е система за формално структуриране на КСУ.
- Реализирана е начална структура на система за формална верификация на генерираните управляващи структури.
- Предложена е система за разполагане на КСУ върху реалната компютърна мрежа. Изградена е система за провеждане на off-line експерименти с цел определяне на диспечируемостта на генерираната система.

Резултати от направените изследвания и авторски претенции:

1. Направен е литературен обзор, изследване и анализ на:

- модели за формално описание и реализация на компютърни системи.
- модели за описание на системи за управление във връзка с моделите за описание на компютърни системи.
- начините за формално описание на разпределени компютърни системи и на разпределени компютърни системи за управление.
- програмни езици, програмни генератори и среди за разработка на РКСУ.

На база на направените изследвания са формулирани недостатъци на наличните системи и средства и посока на работа за преодоляване на част от тях.

2. Предложен е нова семантика на интерпретираните автомати чрез модел за формално описание на КСУ, който позволява описанието на хибридна разпределена (и съсредоточена) КСУ на базата на разширен автомат на Мур. Чрез предложения подход се предлага подобрен начин за елиминиране на проблема с наличието на доминиращ математически апарат – еднакво се представят както непрекъснати, така и дискретни системи.

3. Предложен е формален модел за описание на комуникационните транзакции на РКСУ, позволяващ изграждане на структурата на системата с доминиращо отчитане на изискванията за функциониране на управляващото устройство на системата за управление. Във връзка с това е предложена нова структура на комуникационната подсистема и протокол, базирани на контекстно-ориентирано адресиране.

4. Разработена е програмна среда за реализация на управляващи системи на базата на предложената системна структура. За целта е предложена структура на системните таблици и структура на интерпретивните програми за автомати и трансформационни структури.

5. Предложена е методика и структура на програмното осигуряване и апаратурата за целите на измервания, необходими за верификация на времевата валидност на генерираните управляващи структури.

6. Формулирани са основните изисквания към елементите на системата за управление и програмния генератор, необходими за реализирането на предложения формален модел. Направена начална реализация на програмен генератор за конфигуриране на управляващи системи съгласно предложения формален модел.

1. Abramsky S., S. J. Gay and R. Nagarajan. “Interaction Categories and the Foundations of Typed Concurrent Programming.” In: Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School (M. Broy, ed.). NATO ASI Series F, Springer-Verlag, 1995.
2. Abu-Ghazaleh N., P. Alexander, D. Dieckman, R. Murali, and J. Penix, “Orbit — A Framework for High Assurance System Design and Analysis,” University of Cincinnati, TR 211/01/98/ECECS, 1998.
3. Agha G. A, “Concurrent Object-Oriented Programming,” Communications of the ACM, 33(9), pp. 125-141, 1990.
4. Agha G. A., “Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems,” in Formal Methods for Open Object-based Distributed Systems, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.
5. Agha G. A., “Actors: A Model of Concurrent Computation in Distributed Systems”, MIT Press, Cambridge, MA, 1986.
6. Agrawal G., B. Chen, W. Zhao and S. Davari. “Guaranteeing synchronous message deadlines with the timed token medium access control protocol”. IEEE Trans. on Comput., vol. 43, No 3, Mar. 1994.
7. Ahuja S., N. Carreiro, and D. Gelernter, “Linda and Friends,”Computer, Vol. 19, No. 8, Aug. 1986, pp. 26-34.
8. AIRES: Automatic Integration of Reusable Embedded Software, http://kabru.eecs.umich.edu/aires.
9. Allen R. and D. Garlan, “Formalizing Architectural Connection,” in Proc. of the 16th International Conference on Software Engineering (ICSE 94), May 1994, pp. 71-80, IEEE Computer Society Press.
10. Andrews D., D. Ince. “Practical Formal Methods with VDM”, McGraw Hill Pub., Ed., pp. 450, 1991.
11. Angelov C. K. and P. I. Zhivkov. “A design method for distributed computer control systems”. 11th Int. Conf. on Systems Science 'Systems Science XI', Wroclaw, Poland, Sept. 1992.
12. Angelov C. K. and V. E. Georgiev. “An interconnection network for hard real-time distributed systems”. Proc. of the IFIP International Symposium on Network Information Processing Systems NIPS '97, Sofia, Bulgaria, Oct. 1997.
13. Angelov C. K. et al. “A software system for low - cost process automation”. Prepr. of the 3rd IFAC Symposium on Low - Cost Automation, Vienna, Austria, Sept. 1992.
14. Angelov C. K., I. E. Ivanov and P. T. Petkov. “A generalized approach to program generation for process control applications”. Prepr. of the 18th IFAC Workshop on Real - Time Programming, Bruges, Belgium, June 1992.
15. Angelov C. K., I. E. Ivanov and V. E. Georgiev. “A software architecture for distributed hard real-time computer control systems”. Proc. of the National Conference on Automation and Information Technology "Automation and Informatics '97", Sofia, Bulgaria, Oct. 1997.
16. Angelov C. K. and I. E. Ivanov, “Formal Specification of Distributed Computer Control Systems (DCCS). Specification of DCCS Subsystems and Subsystem Interactions”. Proc. of the International Conference “Automation & Informatics’2001”, May 30 - June 2, 2001, Sofia, Bulgaria, vol. 1, pp. 41-48.
17. Ангелов Х. К. “Софтуерна архитектура за разпределени управляващи системи”. Национална конференция “Автоматика и информатика’94”, София, ноември 1994.
18. Audsley N. C., A. Burns and A. J. Wellings. “Deadline - monotonic scheduling theory and application”. Control Eng. Practice, vol. 1, No 1, 1993.

19. Audsley N. C., A. Burns, M. F. Richardson and A. J. Wellings. “Hard real - time scheduling: the deadline monotonic approach”. Proc. of the 8th IEEE Workshop on Real - Time Operating Systems and Software, Atlanta GA, USA, 1991.
20. Audsley N. C., A. Burns, M. Richardson, K. Tindell and A. J. Wellings. “Applying new scheduling theory to static-priority preemptive scheduling”, Software Engineering Journal, Sept. 1993.
21. Audsley N. C., R. I. Davis and A. Burns. “Mechanisms for enhancing the flexibility and utility of hard real - time systems”. Proc. of the IEEE Real - Time Systems Symp., 1994.
22. Babb M. “Fast computers open the way for advanced controls”. Control Engineering, Jan. 1991.
23. Baker T. and O. Pazy.” Real - time features for ADA 9X”. Proc. of the 12th IEEE Real - Time Systems Symp., 1991.
24. Baker T. P. “A stack - based resource allocation policy for real - time processes”. Proc. of the 11th IEEE Real - Time Systems Symp., 1990.
25. Baker T. P. “Stack - based scheduling of real - time processes”. J. of Real - Time Syst., vol 3, No 1, 1991.
26. Balarin F. et al., “Hardware-Software Co-Design of Embedded Systems: The POLIS Approach”. Norwell, MA: Kluwer., 1998.
27. Bender K. (Ed.)” PROFIBUS - the fieldbus for industrial automation”. Prentice - Hall, 1992
28. Bennet S. “Real - time computer control: an introduction”. Prentice - Hall, 1994
29. Benveniste A. “Synchronous languages provide safety in reactive system design”. Control Engineering, Sept. 1994.
30. Benveniste A. and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems”, Proceedings of the IEEE, vol. 79, no. 9, 1991, pp. 1270-1282.
31. Benveniste A. and K. Aström, “Meeting the Challenge of Computer Science in The Industrial Applications of Control: an Introductory Discussion”, IEEE Trans. on Aut. Control, vol. 38, No 7, July 1993.
32. Benveniste A. and P. Le Guernic, “Hybrid Dynamical Systems Theory and the SIGNAL Language,” IEEE Trans. On Automatic Control, vol. 35, no. 5, pp. 525-546, May 1990.
33. Berard B., M. Bidoit et al., “Systems and Software Verification. Model-Checking Techniques and Tools”, Springer, 2001.
34. Berry G. and G. Gonthier, “The Esterel synchronous programming language: Design, semantics, implementation,” Science of Computer Programming, vol. 19, no. 2, pp. 87-152, 1992.
35. Berry G., “Real-Time programming: Special purpose or general purpose languages,” in Information Processing, Ed. G. Ritter, Elsevier Science Publishers B.V. (North Holland), vol. 89, pp. 11-17, 1989.
36. Berwanger J. et al. “FlexRay–the communication system for advanced automotive control systems”. In SAE 2001 World Congress, Society of Automotive Engineers, Detroit, MI, April 2001. Paper number 2001-01-0676.
37. Bhattacharyya S. S., P. K. Murthy and E. A. Lee, “Software Synthesis from Dataflow Graphs”, Kluwer Academic Publishers, Norwell, Mass, 1996.
38. Biemans F. P. M., C.A. Vissers. “A Systems Theoretic View of Computer Integrated Manufacturing”; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 390 – 410.
39. Bihari T. E. and P. Gopinath. “Object-oriented real-time systems: concepts and examples”. IEEE Computer, Dec. 1992.
40. Bittici U. S. “Computer Integrated Manufacturing Systems: Status and Trends”, Control and Dynamic Systems; Vol. 60; 1994; pp. 1-31.

41. Boasson M. “Control Systems Software”. IEEE Trans. on Automatic Control, vol. 38, No 7, July 1993.
42. Bohms M., F.Tolman. “RIA: Reference Model for Industrial Automation”, Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 114 – 132.
43. Brookings K. R. “State logic: a paradigm for integrated control in the 90's”. Control Engineering, March 2, 1991.
44. Brunel J. Y., A. Sangiovanni-Vincentelli, and R. Kress, “COSY: A methodology for system design based on reusable hardware & software IP’s,” in Technologies for the Information Society, J.-Y. Roger, Ed. Singapore: IOS, June 1998, pp. 709–716.
45. Brunel J. Y., W. Kruijtzer, H. Kenter, F. Pétrot, L. Pasquier, E. de Kock and W. Smits, “COSY communication IP’s,” in Design Automation Conf. 2000, Los Angeles, CA, June 2000.
46. Bryant R., E. Clarke and D. Dill, “GSRC formal verification effort” in Annual GSRC Activities, San Jose, CA, Dec. 1999.
47. Bucci G., M. Campanai and P. Nesi, “Tools for Specifying Real-Time Systems”. Real-Time Systems, vol. 8, 1995, p.p. 117-172.
48. Burns A. “Preemptive priority - based scheduling: an appropriate engineering approach”, in Advances in Real - Time Systems, ed. S. H. Son, Prentice - Hall, 1994.
49. Burns A. “Scheduling hard real - time systems: a review”. Software Engineering Journal, vol. 6, No 5, May 1991.
50. Burns A. and A. Wellings, “Real-Time Systems and Programming Languages”, Addison-Wesley 2001.
51. Burns A. and A. Wellings. “Concurrency in ADA”. Cambridge University Press, 1995.
52. Burns A. and A. Wellings. “HRT - HOOD (TM): a structured design method for hard real - time ADA systems”. North - Holland, 1995.
53. Burns A. and A. Wellings. “HRT - HOOD: a design method for hard real - time systems” J. of Real - Time Systems, vol 6, No 1, 1994.
54. Burns A. and A. Wellings. “Real - time systems and their programming languages”. Addison - Wesley, 1990.
55. Burns A. and A. Wellings.” Analysis of hard real-time communications”. J. of Real-Time Systems, No 9, 1995.
56. Burns A., “The Event-Triggered Model of Computation”, Real-Time Systems Research Group, Department of Computer Science, University of York, http://www.cs.york.ac.uk/rts.
57. Burns A., K. Tindell and A. Wellings,” Effective analysis for engineering real-time fixed-priority schedulers”. IEEE Trans. on Soft. Eng., vol. 21 (1995), pp. 475-480.
58. CAN Application layer for industrial applications. CiA Draft Standards DS-201 ... DS-207, Feb. 1996.
59. CANopen. Communication profile for industrial systems (based on CAL). CiA Draft Standard 301, Oct. 1996.
60. CANopen. Device profile for I/O modules. CiA Draft Standard Proposal 401, Dec. 1996.
61. Carriero N. and D. Gelernter, “Linda in Context,” Comm. Of the ACM, vol. 32, no. 4, pp. 444-458, April 1989.
62. Caspi P., D. Pilaud, N. Halbwachs, and J. A. Plaice, “LUSTRE: A Declarative Language for Programming Synchronous Systems,” Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages, Munich, Germany, January, 1987.
63. Chatha A. “Fieldbus: the foundation for field control systems”. Control Engineering, May 1994.

64. Coulas M. F., G. H. Macewen and G. Marquis.” RNET: a hard real - time distributed programming system”. IEEE Trans. on Comput., vol C - 36, No 8, 1987.
65. Cox B. J. “Planning the software industrial revolution”, IEEE Software, Nov. 1990.
66. Crater K. C. “When Technology Standards Become Counterproductive” URL:”http://www.ctc-control.com/customer/elearning/whpapers/counter.asp”
67. Davis II J., M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, “Overview of the Ptolemy Project,” ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, July 1999.
68. Davis R. I., K. W. Tindell and A. Burns. “Scheduling slack time in fixed-priority preemptive systems”. Proc. of the IEEE Real - Time Systems Symp., 1993.
69. Davis R. I.. “Dual - priority scheduling: a means of providing flexibility in hard real - time systems”. Report YCS 230, Dept. of Computer Science, University of York (1994).
70. Dijkstra E., “Cooperating Sequential Processes”, in Programming Languages, E F. Genuys, editor, Academic Press, New York, 1968.
71. Dimov A, I.E. Ivanov and K. Milenkov, “Component-based approach for Distributed Hard Real-Time Computer Systems”, Information Technologies and Control, под печат.
72. Douglass B. P., “Real-Time UML”, Addison Wesley, 1998.
73. Douglass B.P., “Real-Time UML, Developing Efficient Objects for Embedded Systems”, Addison-Wesley, 1999.
74. Kopetz H. and G. Grundsteidl. “TTP - a protocol for fault - tolerant real - time systems”. Computer, Jan. 1994.
75. Faergemand O. and A. Olsen, “Introduction to SDL-92”, Computer Networks and ISDN Systems, vol. 26 (1994), pp. 1143-1167.
76. Felder M., D. Mandrioli and A. Morzenti. “Proving properties of real - time systems through logical specifications and Petri - net models”. IEEE Trans. on Soft. Eng., vol 20, No 2, Feb. 1994.
77. Ferrari A. and A. Sangiovanni-Vincentelli, “System design: Traditional concepts and new paradigms,” in 1999 Int. Conf. Computer Design, Austin, TX, Oct. 1999.
78. Gamma E., R. Helm, R. Johnson, and J. Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, 1994.
79. Gerold J. S. “Industrial PC ' s tackle tough terrain”. Control Engineering, Aug. 1994.
80. Gerold J. S. “PC software does windows for operator interface”. Control Engineering, Sept. 1993.
81. Gien M. “1996 technology forecast: embedded systems of the future. Real-time operating systems move to the open world”. Electronic Design, Jan. 8, 1996.
82. Girault A., B. Lee, and E.A. Lee, “Hierarchical Finite State Machines with Multiple Concurrency Models,” IEEE Trans. CAD Integrated Circuits and Systems, June 1999, pp. 742-760.
83. Gomaa H. “A software design method for distributed real - time applications”. The Journal of Systems and Software, vol. 9, No 2, 1989.
84. Gomaa H. “Software design methods for concurrent and real - time systems”. The SEI Series in Software Engineering. Adisson - Wesley, 1993.
85. Gruhler G., M. Tischer and L. Felbinger. “CANopen networks: implementation and configuration aspects”. Proc. of the 3rd International CAN Conference, Paris, France, 1996.
86. Halbwachs N., P. Caspi, P. Raymond, D. Pilaud, “The Synchronous Data Flow Programming Language LUSTRE,” Proc. of the IEEE, Vol. 79, No. 9, 1991, pp. 1305-1319.

87. Harbour M. G., M. H. Klein and J. P. Lehoczky. “Timing analysis for fixed priority scheduling of hard real - time systems”. IEEE Trans. on Soft. Eng., vol 20, No 1, Jan. 1994.
88. Harel D. “STATECHARTS: a visual formalism for complex systems”. Science of Computer Programming, vol 8, No 3, 1987.
89. Heimdahl M. P. E. and N. G. Leveson. “Completeness and consistency in hierarchical state-based requirements”. IEEE Trans. on Soft. Eng., vol 22, No 6, June 1996.
90. Henzinger T. A., “The theory of hybrid automata,” in Proceedings of the 11th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, 1996, pp. 278-292, invited tutorial.
91. Hoare C. A. R., “Communicating Sequential Processes,” Communications of the ACM, Vol. 21, No. 8, August 1978.
92. Huggard A. Client-server approach links distributed controls to desktop PC’s”. Control, Sept. 1994.
93. IAR visualSTATE® Concept Guide, IAR Systems, 1999.
94. Intec Control Corp. “Distributed architecture comes do automation software”. Control, Sept. 1994.
95. ISA / 94 Philadelphia: a focus on solutions. Control, June 1994.
96. ISA SP50 - 1990 - 389B Draft Standard. User Layer Technical Report, 1990.
97. Ishikawa Y., H. Tokuda and C. W. Mercer. “An object - oriented real - time programming language”. Computer, vol 25, No 10, Oct. 1992.
98. Isovic D. and C. Norström, “Components in Real-Time Systems”, URL: www.mrtc.mdh.se/publications/0376.pdf.
99. Ivanov I. E., “Schedulability of mixed Real-Time and general messages in heterogeneous type networks (CAN-TCP/IP)”, Microsoft Research Project, Project report, 2004
100. Ivanov I. E., K. Filipova, “Integrated scheduling of heterogeneous CAN and Ethernet-based hard Real-Time network”, Proc. of IEEE spring seminar 27th ISSE, Annual School Lectures, Bulgaria, 2004,vol. 24, pp.481-485
101. Ivanov I. E., V. Georgiev, “Formal models for system design”, Proc. of IEEE spring seminar 27th ISSE, Annual School Lectures, Bulgaria, 2004,vol. 24, pp. 564-568
102. Jobling C. P., P. W. Grant, H. A. Barker and P. Townsend. “Object - oriented programming in control system design: a survey”. Automatica, vol 30, No 8, 1994.
103. John K.H. and M. Tiegelkamp, IEC 61131-3: Programming Industrial Automation Systems, Springer, 2001.
104. Kagan H. “Objects in real – time”. Byte, Aug. 1992.
105. Kahn G., “The Semantics of a Simple Language for Parallel Programming,” Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.
106. Kamin S. “Research on Domain-specific Embedded Languages and Program generators” in Electronic Notes in technical Computer Science 14, 1998, URL: http://www.elsevier.nl/locate/entcs/volume14.html.
107. Kamin S. and Lars Clausen. “Dynamically reconfigurable software components”, Oct. 2001
108. Katcher D., H. Arakawa and J. Strosnider, “Engineering and Analysis of Fixed-Priority Schedulers”, IEEE Trans. on Soft. Eng., vol. 19 (1993), pp. 920-934.
109. Kiencke U. (Ed.). Proceedings of the 1st International Workshop on Open Systems in Automotive Networks, Karlsruhe, Germany, Oct. 1995 (посветен на сандарта OSEK).
110. Kiencke U. and K. J. Neumann. “OSEK/VDX - an open software architecture for communicating vehicle systems”. Proc. of the 3rd International CAN Conference, Paris, France, 1996.

111. Kienhuis B., E. Deprettere, K. Vissers, and P. van der Wolf, “An approach for quantitative analysis of application-specific dataflow architectures,” in 11th Int. Conf. Application-Specific Systems, Zurich, Switzerland, July 14–16, 1997.
112. Klein M. H., T. Ralya, B. Pollak, R. Obenza and M. G. Harbour. “A practitioner's handbook for real-time analysis: guide to rate-monotonic analysis for real-time systems”. Kluwer Academic Publishers, 1993.
113. Klein M., J. Lehoczky and R. Rajkumar. “Rate - monotonic analysis for real - time industrial computing”. Computer, Jan. 1994.
114. Kopetz H. and Günter Grünsteidl. “TTP—a protocol for fault-tolerant real-time systems”. IEEE Computer, 27(1):14–23, Jan. 1994.
115. Kopetz H. et al.” Distributed fault - tolerant real - time systems. The MARS approach.” IEEE Micro, Feb. 1989.
116. Kopetz H., “Real-Time Systems, Design Principles for Distributed Embedded Applications”, Kluwer Academic Publishers, 1997.
117. Kramer T. R. “A Framework for Control Architectures”, International Journal of Computer Integrated Manufacturing; Volume 11, Number 4, July-August 1998, pp. 347 - 363.
118. Kramer T. R., Senehi, M. K.. “Feasibility Study: Reference Architecture for Machine Control Systems Integration”,NISTIR 5297; National Institute of Standards and Technology; 1993.
119. Laddaga R., “Active Software,” position paper for the St.Thomas Workshop on Software Behavior Description, December, 1998.
120. Lamport L., “Time, Clocks, and the Ordering of Events in a Distributed System,” Communications of the ACM, Vol. 21, No. 7, July, 1978, pp. 558-565.
121. Lee E. A. and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,” IEEE Trans.CAD Integrated Circuits and Systems, Dec. 1998, pp. 1217-1229.
122. Lee E. A. and D. G. Messerschmitt, “Synchronous Data Flow,” Proceedings of the IEEE, September, 1987.
123. Lee E. A. and T. M. Parks, “Dataflow Process Networks,”, Proceedings of the IEEE, vol. 83, no. 5, pp. 773-801, May, 1995.
124. Lee E. A., “What’s Ahead for Embedded Software?” IEEE Computer, 2000 - 9, pp. 18-26
125. Lee E. and A. Sangiovanni-Vincentelli, “A unified framework for comparing models of computation,” IEEE Trans. Computer-Aided Design, vol. 17, pp. 1217–1229, Dec. 1998.
126. Lee I. and S. B. Davidson. “Adding time to synchronous process communications”. IEEE Trans. on Comput., vol. C - 36, No 8, Aug. 1987.
127. Lee I. and S. B. Davidson. “Protocols for timed synchronous process communication”. Proc. of the 7th IEEE Real - Time Systems Symp., 1986.
128. Lehoczky J. P. “Fixed priority scheduling of periodic task sets with arbitrary deadlines”. Proc. of the IEEE Real - Time Systems Symp., 1990.
129. Lehoczky J. P. and S. Ramos - Thuel. “An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive systems”. Proc. of the 13th IEEE Real - Time Systems Symp., 1992.
130. Lehoczky J. P., L. Sha and J. K. Strosnider. “Enhancing aperiodic responsiveness in a hard real - time environment”. Proc. of the 8th Real- Time Systems Symp., 1987.
131. Lehoczky J. P., L. Sha and Y. Ding. “The rate - monotonic scheduling algorithm - exact characterization and average-case behaviour”. Proc. of the IEEE Real - Time Systems Symp.,1989.

132. Lehoczky J. P., L. Sha, J. K. Strosnider and H. Tokuda. “Fixed-priority scheduling for hard real - time systems”. In Foundations of Real - Time Computing: Scheduling and Resource Management, A. van Tilborg and G. M. Koob, Eds., Kluwer, N. Y., 1991.
133. Leveson N., M. P. E. Heimdahl, H. Hildreth and J. D. Reese. “Requirements specifications for process control systems”. IEEE Trans. on Soft. Eng., vol 20, No 9, Sept. 1994.
134. Levi S. T. and A. K. Agrawala.”Real - time systems design”. Mc. Graw - Hill, 1990.
135. Lewis R., “Modeling Control Systems Using IEC 61499”, Institution of Electrical Engineers, 2001
136. Liao S., S. Tjiang, R. Gupta, “An efficient implementation of reactivity for modeling hardware in the Scenic design environment,” Proc. of the Design Automation Conference (DAC 97), Anaheim, CA, 1997.
137. Liu C. and J. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” JACM, vol. 20, pp. 46--61, January 1973.
138. Liu J. W. S., “Real-Time Systems”, Prentice-Hall, 2000.
139. Locke C. D. “Software architecture for hard real - time applications: cyclic executives vs. fixed priority executives”. Journal of Real - Time Systems, vol 4, No 1, 1992.
140. Loftis D. and W. Cawlfield. “Personal computers tackle the direct control stigma”. Control Engineering, March 1994.
141. Lyden W. P. “OOP's new role in software for control engineering”. Control Engineering, Mid - March, 1992.
142. Malcolm M. and W. Zhao. “The timed token protocol for real - time communications”. Computer, Jan. 1994.
143. Maraninchi F., “The Argos Language: Graphical Representation of Automata and Description of Reactive Systems,” in Proc. IEEE Workshop on Visual Languages, Kobe, Japan, Oct. 1991.
144. Marian N. and C. Angelov. “Compositional Verification of Embedded Systems Software”, CSI Seminar on Safety-Critical Software, Soenderborg, June 12, 2003.
145. Meinel. C. and Th. Theobald. “Algorithms and data structures in VLSI desing”, Springer-Verlag Berlin Heidelberg 1998.
146. Mercer C. and H. Tokuda,”The ARTS Real-Time Object Model”, Proc. of the IEEE Real-Time Systems Symposium, 1990, pp. 2-10.
147. MetaMatrix, “Model Driven Architecture” OMG 2001, URL: http://www.omg.org/mda/executive_overview.htm.
148. Morris H. M. “Signal conditioning is becoming I / O processing”. Control Engineering, Aug. 1994.
149. Nierstrasz O. et al., “A Component Model for Field Devices” URL: www.iam.unibe.ch/~scg/Archive/pecos/public_documents/Nier02aPecosModel.pdf.
150. ODVA Device Net™ Specification, vol. I: Device Net communication model and protocol; vol. II: Device Net device profiles and object library, Open Device Net Association, 1996.
151. Olsson G. and G. Piani. Computer systems for automation and control. 3d edition, Prentice - Hall, 2001
152. PECOS: Pervasive Component Systems, EU Project IST-1999-20398, http://www.pecos-project.org.
153. Pfeifer H. “Formal verification of the TTA group membership algorithm”. In Tommaso Bolognesi and Diego Latella, eds., Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000, pages 3–18, Pisa, Italy, Oct. 2000.
154. Pfeifer H., D. Schwier and F. W. von Henke. “Formal verification for time triggered clock synchronization”, in Charles B. Weinstock and John Rushby, eds., Dependable

Computing for Critical Applications—7, Volume 12 of IEEE Computer Society Dependable Computing and Fault Tolerant Systems, pages 207–226, San Jose, CA, Jan. 1999.
155. Pleinevaux P. and J. D. Decotignie. “Time - critical communication networks: fieldbuses”, IEEE Network, vol. 2, No 3, May 1988.
156. Poledna S., A. Burns, A. Wellings and P.Barrett, “Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems”, IEEE Trans. on Comp., vol. 49, No 2, Feb. 2000.
157. Pollard J. R. “Ladder logic remains the PLC language of choice”. Control Engineering, Apr. 1994.
158. Popovic D. “Distributed computer control for industrial automation”. Control Engineering, Aug. 1994.
159. Puschner P., G. Bernat and A. Wellings. “Making JAVA Hard Real-Time”. 2001
160. Quintero R., A. Barbera. “A Real-Time Control System Methodology for Developing Intelligent Control Systems”; NISTIR 4936; National Institute of Standards and Technology; October 1992.
161. Raji R. S. “Smart networks for control”. IEEE Spectrum, June 1994.
162. Rajkumar R. “Task synchronisation in real - time systems”. Kluwer Academic Publishers, 1991.
163. Rushby J.. “A comparison of bus architectures for safety-critical embedded systems”. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 2001. Available at http://www.csl.sri.com/˜rushby/papers/buscompare.pdf.
164. Sathaye S. S. and J. K. Strosnider. “A real - time scheduling framework for packet - switched networks”. Proc. of the 14th IEEE Conf. on Distributed Computing Systems, Poznan, Poland, 1994.
165. Sgroi M., L. Lavagno, and A. Sangiovanni-Vincentelli, “Formal models for embedded system design”, in IEEE Design Test Comput. (Special Issue on System Design), 2000.
166. Sha L. and J. B. Goodenough. “Rate - monotonic scheduling theory and ADA”. IEEE Computer, vol 23, Apr. 1990.
167. Sha L., M. H. Klein and J. B. Goodeneugh. “Rate - monotonic analysis for real - time systems”. In Foundations of Real - Time Computing: Scheduling and Resource Management, A. van Tilborg and G. M. Koob, Eds., Kluwer, N. Y., 1991.
168. Sha L., R. Rajkumar and J. P. Lehoczky. “Priority inheritance protocols: an approach to real - time synchronization”. IEEE Trans. on Comp., vol 9, No 9, 1990.
169. Sha L., R. Rajkumar and S. H. Son. “A real - time locking protocol”. IEEE Trans on Comp., vol 40, No 7, 1991.
170. Sha L., R. Rajkumar and S. Sathaye. “Generalised Rate-Monotonic Scheduling Theory: a Framework for Developing Real-Time Systems”, Proc. of the IEEE, vol. 82, 1994, pp. 68-82.
171. Shaw A. C., “Communicating real - time state machines”. IEEE Trans. on Soft. Eng., vol. 18, No 9, Sept. 1992.
172. Shaw M.,” Beyond Objects: A Software Design Paradigm Based on Process Control”, ACM Software Engineering Notes, vol. 20, No 1, Jan. 1995.
173. Shiu C. et al., “Specifying Reconfigurable Control Flow for Open Architecture Controllers”, Proceedings of 1998 Japan-USA Symposium on Flexible Automation, June 1998.
174. SINTEF Telecom and Informatics. “ProgGen: SDL transformation Tool”, 1998.
175. StateWORKS: Software as Engineering, URL: http://www.stateworks.com.
176. Strosnider J. K., T. Marchok and J. Lehoczky. “Advanced real - time scheduling using the IEEE 802. 5 token ring”. Proc. of the 9th IEEE Real-Time Symp., 1988.

177. Szyperski C. “Component Software: Beyond Object-Oriented Software”. Reading, MA: ACM/Addison-Wesley, 1998.
178. 223a. SystemC - http://systemc.org
179. Thane H., A. Pettersson and D. Sundmark. “The Asterix Real-Time Kernel”, URL: www.mrtc.mdh.se/publications/0349.pdf.
180. Thomesse J. P. et al. “Factory instrumentation protocol: model, products and tools”. Control Engineering, Sept. 1991.
181. Tindell K. W., A. Burns and A. J. Wellings. “Mode changes in priority pre-emptively scheduled systems”. Proc. of the 13th IEEE Real - Time Systems Symp., 1992.
182. Tindell K. W., H. Hansson and A. J. Wellings. “Analysing real - time communications: controller area network (CAN)”. Proc. of the 15th IEEE Real - Time Systems Symp., 1994.
183. Tindell K., A. Burns and A. J. Wellings.” Analysis of Hard Real-Time Communications”, Real-Time Systems, vol. 7, No 9, 1995, pp. 147-171.
184. Tindell K., A. Burns and A. Wellings. “Calculating controller area network (CAN) message response times”. Proc. of the IFAC Workshop on Distributed Computer Control Systems DCCS ' 94, Toledo, Spain, 1994.
185. Tindell K., A. Burns. “Guaranteeing message latencies on controller area network (CAN)”. Proc. of the 1st International CAN Conference, 1994.
186. Tokuda H., T. Nakajima and P. Rao. “Real - time Mach: towards a predictable real - time system”. Proc. of the USENIX Mach Workshop, 1990.
187. Van Bekkum J. “Open programmable controllers”. INTECH, June 1994.
188. Van Gompel D. “Device Net offers a new option for device networking”. Control Engineering, March, 1995.
189. Waldo J., “JiniTM Architecture Overview,” Sun Microsystems, URL: http://www.sun.com/products/jini, 1998.
190. Wang S. and K.G. Shin, “An Architecture for Embedded Software Integration Using Reusable Components”, Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, San Jose, CA., 2000.
191. Wang S. and K.G. Shin, “Generic Programming Paradigm for Open Architecture Controllers”, Proc. of the WAC 2000/Seventh International Symposium on Manufacturing with Applications (WAC), June 2000.
192. Warnock I. G. “Programmable controllers: operation and application”. Prentice - Hall, 1988.
193. Waterbury R. C. “Control software drives PC engines”. INTECH, June 1992.
194. Waterbury R. C. “Software trends lead to control forefront”. INTECH, June 1994.
195. 249a. Wellings A., L. Beus-Dukic and D. Powell. “Real-Time Scheduling in a Generic Fault-Tolerant Architecture, *Proc. of the IEEE Real-Time Systems Symposium*, 1998, pp. 390-398.
196. Wing J. “A specifier’s introduction to formal methods”, Computer, Sept. 1990.
197. www.rempli.org – Communication Over Power Lines.
198. Zagar K., M. Plesko, M. Sekoranja, G. Tkacik, A. Vodovnik. “The Control System Modeling Language”, ICALEPCS01, 2001.
199. Zeltwanger H. “An inside look at the fundamentals of controller area networks (CAN)”. Control Engineering, Jan. 1995.
200. Zimmermann U., M. Wenz, T. Längle and H. Wörn, “Describing Components and Configurations for Embedded Control Systems”, Proceedings of the 4th International Workshop on Computer Science and Information Technologies (CSIT'2002).
201. Zimmermann U., T. Längle and H. Wörn, “Online Software Reconfiguration in Embedded Real-Time Control Systems”, Proc. of the SCI 2002, vol. VI, pp. 375-379.
202. Z. Armstrong and Gray, “VHDL Design Representation and Synthesis”, Prentice-Hall, 2000.

П1. Приложение 1

1. Команди на ОСРВ HARTEX, реализиращи системни примитиви за векторно сигнализиране

1.1 Команда за сигнализиране и възобновяване изпълнението(ако е необходимо) на набор от меки реално-времеви задачи

```
//-----------------------------------------
// Използвани означения
// MAS( MessageArrivalSemafore ) – Масив от семафори
// сигнализиращи пристигане на съобщение
signal_and_resume( semaphore, tasks )
{
        MAS[semaphore].flags = MAS[semaphore].flags | tasks;
        //сигнализира задачите на дадения семафор
        //MAS – Message Arrival Semafore
        MBP = MBP | (MAS[semaphore].queue & tasks);
        //избира задачите за деблокиране и го прави в MBP
        MAS[semaphore].queue = 0;
        //изчиства опашката на блокираните задачи на този
        //семафор
        preempt();
 //Извикване на диспечера на задачите
}
```

1.2 Команда за блокиране на текущата задача до сигнализиране на даден семафор или до изтичане на зададено време

```
 wait( semaphore, timeout )
 {
  if( MAS[semaphore].flags[current_task] == 1 ){
        //Проверка за вдигнат семафор
        MAS[semaphore].flags[current_task] = 0;//Да-сваляне
        return OK;//Връща – семафора беше вдигнат
        } else {//Не–не е вдигнат
                MAS[semaphore].queue[current_task] = 1;
                //маркиране на текущ.задача за деблокиране
                if( timeout == 0 )//Ако няма timeout
                        suspend();
                        //"прекъсване" на задачата до
                        // извикване на signal_and_resume
                        // за тази задача и семафор
                else
                        waitfor( timeout );
                        //изчакване до извикване на
                        //signal_and_resume или изтичане на timeout

                if( MAS[semaphore].flags[current_task] == 1 )      { //вдигнат ли е семафора?
                  MAS[semaphore].flags[current_task] = 0;
                        //Да-сваляне
                        return OK;//Връща– семафора беше вдигнат
                } else {
```

```
                MAS[semaphore].queue[current_task] = 0;
                //НЕ – премахване на задачата от опашката
                //на чакащите на семафора

                return timeout_err;   //Връща– изтичане на timeout преди
                                      //сигнализиране на семафора
            }
        }
}
```

1.3 Команда за изчакване в текущата задача докато един от набор семафори не бъде вдигнат

```
wait_multiple( first, last )
{
    if( first < 0 )//Проверка за отрицателен първи семафор
        first = 0;//Отрицателен ⇒ първият е нула
    if( last >= MAS_Size )//Проверка невалиден последен семафор
        last = MAS_Size-1;//Невалиден ⇒ приема последния валиден номер
    if( first <= last )
    {   //Ако номерата са коректно подадени
        for( i=first; i<=last; i++ )//Обхождане на ∀ семафори в интервала [first,last]
            if( test_and_reset( i )) //текущ.семафор е вдигнат
                return i; //връщане на номера му

        while( true )
        {   //Старт на безкраен цикъл
            //никой семафор в [first,last] не е вдигнат
            for( i=first; i<=last; i++ )
                //Обхождане на ∀ семафори в интервала[first,last]
                MAS[i].queue[current_task] = 1;
                //Маркиране на тек. задача като
                //чакаща на индексирания семафор
            suspend();//Прекъсване на текущата задача до
                      //извикване на signal_and_resume
                      //за тази задача и някои от
                      //семафорите[first,last]
            for( i=first; i<=last; i++ )
                MAS[i].queue[current_task] = 0;
                //Премахване на тек. задача от чакащите
                //на индексирания семафор
        }
        //Повтаряне на цикъла за откриване на вдигнатия семафор
        for( i=first; i<=last; i++ )//Обхождане на ∀ семафори в интервала [first,last]
            if( test_and_reset( i )) //текущ.семафор е вдигнат
                return i; //връщане на номера му
    } else
        return invalid_semaphore_err;
    //Номерата на семафорите не са
    //коректни ⇒ връщане флаг за грешка
```

```
  }
```

1.4 Команда за сигнализиране и заявяване за изпълнение на набор от задачи

```
signal_and_release( semaphore, tasks )
{
        MAS[semaphore].flags = MAS[semaphore].flags | tasks;
        //сигнализира задачите на дадения семафор
          ATV = ATV | MAS[semaphore].released;
          //Заявяване на набор задачи
}
```

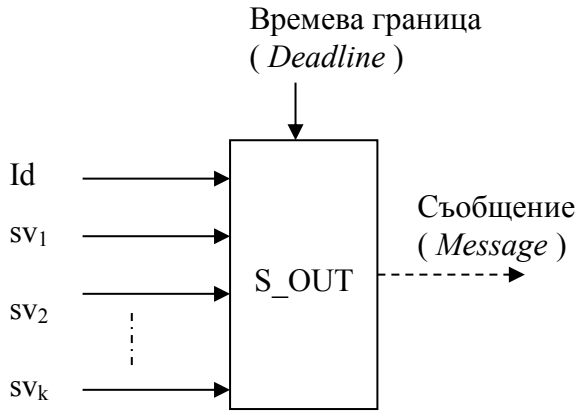
1.5 Команда за проверка на зададен семафор

```
//Проверява дадения семафор за текущата задача и го нулира, ако е вдигнат
test_and_reset( semaphore )
{
        if( MAS[semaphore].flags[current_task] == 1 ){
          //Вдигнат ли е текущият семафор?
          MAS[semaphore].flags[current_task] = 0;//Да⇒нулиране
                return 1;//Да, семафорът беше вдигнат
        } else
                return 0;//Не, семафорът е свален
}
```

2. Драйвери за асинхронна комуникация “производител-потребител”

2.1 Изходен драйвер

Изходният драйвер за излъчване на съобщение в рамките на асинхронно взаимодействие производител-потребител е S_OUT (*Signal_varible_OUT*). Драйверът капсулира в себе си композирането на изходящото съобщение от *n* входни променливи, копирането му в изходящия буфер и изпращането му с командата на операционната система *broadcast(var_name,deadline).*



● *Организация на структурите с данни на сигналния драйвер S_OUT*

Таблица на екземпляр на функционалния модул
(*S_OUT Function Module Instance (FMI) Invocation Block*)

pn = FMI_index

Идентификатор на ФМ (*FM Identifier*) = FM_S_OUT
Брояч на активиране (*Activation Counter (AC)*)
Начална стойност на AC (*Initial AC Value*)
Номер на ресурса(*Resource No*)
Прост / Композитен (*Basic / Composite*)
p - Брой входове(*Input Count*) = p
q - Брой изходи(*Output Count*) = 0
r - Брой вътрешни променливи(*Intern. Var. count*) = 0
Указател към таблицата на вх./изходните дескриптори (*I/O descriptor index*)
…..

Таблица-дескриптор на входовете и изходите на екземпляр на ФМ

(S_OUT Function Module Instance (FMI) – I/O Descriptor Block)

	Тип на променливата (Variable Type)	Външна/ Вътрешна (External/Local)	Адрес в арг.блок (Address)	Бр. Редове (Rows)	Бр. Колони (Columns)	
pIOD	…_TYPE		&sv_1			I_1
pIOD +1	…_TYPE		&sv_2			I_2
	…	…	…	…	…	…
pIOD + p-3	…_TYPE		&sv_n			
pIOD + p-2	LONG_TYPE		&DEADLINE	1	1	I_{p-2}
pIOD + p-1	INTEGER_TYPE		&ID	1	1	I_p

Аргументен блок на екземпляра *(FM Instance Argument Block)*

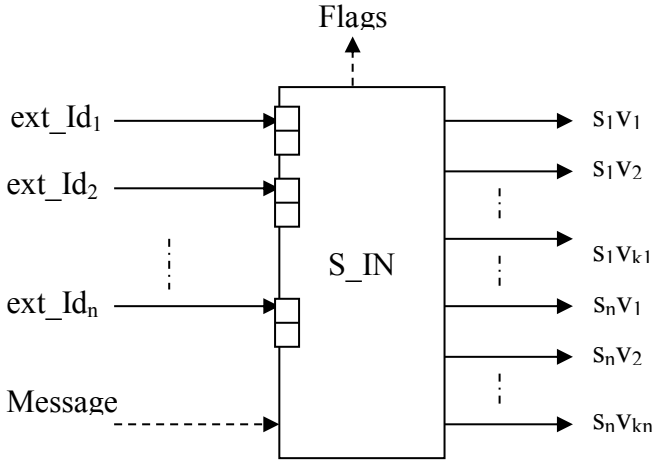
Inputs
Output

● *Псевдокод на сигналния драйвер S_OUT*

```
S_OUT( pn ) {

  p =  pn.inputCount;
  pIOD = pn.IODescIndex;
  &ID = (pIOD + p - 1).Address ;//ID на съобщ.
  &deadline = (pIOD + p - 2).Address;//deadline на съобщ.
  composeMessage();//композиране на съобщ.
  broadcast( ID, deadline );//разпространяване на съобщ.

}
```

2.2 Входен драйвер

Приемане на съобщение в рамките на асинхронно взаимодействие производител-потребител - S_IN (*Signal_varible_IN*). Драйверът капсулира в себе си командата на операционната система *receive_multiple*(*var_names*) за приемане на всички постъпили съобщения, декомпозира ги на съставни променливи и попълва стойностите им в съответната изходна аргументна зона на модула. Входове за модула са разширените идентификатори на всички приемани съобщения – те са структура съдържаща две полета: идентификатор на съобщението и брой изходни променливи, на които се декомпозира. Изходът *flags* (*flags = var_names*) е резултатът от командата *receive_multiple*(*var_names*) и по същество е масив от идентификаторите на получените съобщения. Като вътрешни променливи модулът пази адресите на съответните входни буфери на съобщенията.



● *Организация на структурите с данни на сигналния драйвер S_IN*

S_IN Function Module Instance (FMI) Invocation Block

FMI index

FM_Identifier = FM_S_IN
Activation Counter (AC)
Initial AC Value
Resource No
Basic / Composite
Input Count = p
Output Count = q
InternalVarCount = p
I/O descriptor index = pIOD
…..

S_IN Function Module Instance (FMI) – I/O Descriptor Block

	Variable Type	External/Local	Address	Rows	Columns	
pIOD	ext_ID_TYPE		&ID_1	1	1	I_1
pIOD +1	ext_ID _TYPE		&ID_2	1	1	I_2
	…	…	…	…	…	…
	ext_ID _TYPE		&ID_p	1	1	I_p
pIOD +p	…_TYPE		$\&s_1v_1$			O_1
	…		…	…	…	…
	…_TYPE		$\&s_1v_{k1}$			O_{k1}
	…_TYPE		$\&s_nv_1$			O_x
	…	...	…	…	…	…
	…_TYPE		$\&s_nv_{kn}$			O_{x+kn-1}
pIOD + p+q	BYTE_PNTR		&(&buff_1)	1	1	IV1
	...	...	...	1	1	
	BYTE_PNTR		&(&buff_p)	1	1	IVp

S_IN FM Instance Argument Block

Inputs
Outputs
Internal Variables

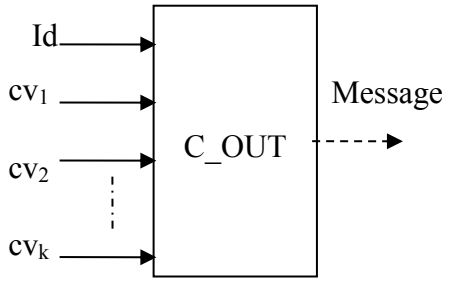
● *Псевдокод на сигналния драйвер S_IN*

```
void S_IN( pn ) {

     var_count = receive_multiple(varnames);
     for(i=0;i< pn.inputCount;i++){
          tempID = getNextExtendedID();
               //прочитане на поредно разширено_ID
          ID = tempID.ID;
               // прочитане на ID от  разширено_ID
          param_count = tempID2.paramCount;
               // прочитане на бр. съставни параметри
               // от разширено_ID
          for(k=0;k<var_count;k++){
               // търсене на ID в получените
               if(ID==varnames[k]) break;
          }
          if(k>=var_count) {
               //Не е намерено
               continue;//Връщане в началото на цикъла и
                        //продължение
          }
          buff = getInputBuffer(ID);
               // взимане вх. буфер за текущото ID
               //от аргументната зона вътрешни_променливи
          fillOutputZone();
               // декомпозиране на съобщ. и
               // попълване на съотв. изходна аргум.зона
     }
}
```

3. Драйвери за синхронен обмен на команди

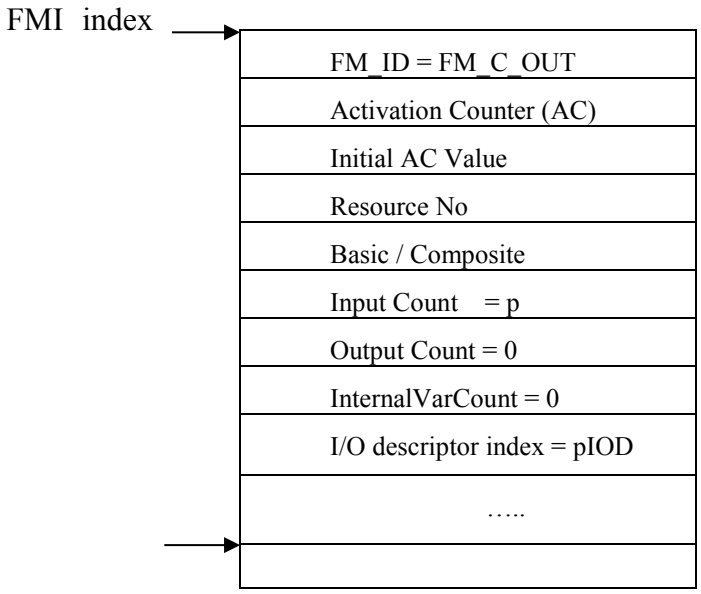
3.1 Изходен драйвер

Изходният драйвер за синхронен обмен на команди осигурява композирането на командното съобщение и изпращането му с командата на ОС *send(var_name).* Приема като входен аргумент идентификатора на изпращаното съобщение и променливите, съставящи командата.



● *Организация на структурите с данни на сигналния драйвер C_OUT*

C_OUT Function Module Instance (FMI) Invocation Block



C_OUT Function Module Instance (FMI) – I/O Descriptor Block

	Variable Type	External/Local	Address	Rows	Columns	
pIOD →	…_TYPE		&cv_1			I_1
pIOD +1 →	…_TYPE		&cv_2			I_2
	…	…	…	…	…	…
pIOD +p-2 →	…_TYPE		&cv_k			
pIOD +p-1 →	INTEGER_TYPE		&ID	1	1	I_p

C_OUT FM Instance Argument Block

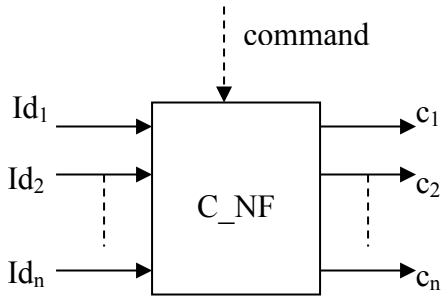
Inputs

● *Псевдокод на сигналния драйвер C_OUT*

```
C_OUT( pn ) {

  p =  pn.inputCount;//брой входове за модула
  pIOD = pn.IODescIndex;//Указател към вх/изх дескриптор
  &ID = (pIOD + p – 1).Address ;//ID на съобщ.
  composeMessage();//композиране на съобщ.
  send( ID );//разпространяване на съобщ.

}
```

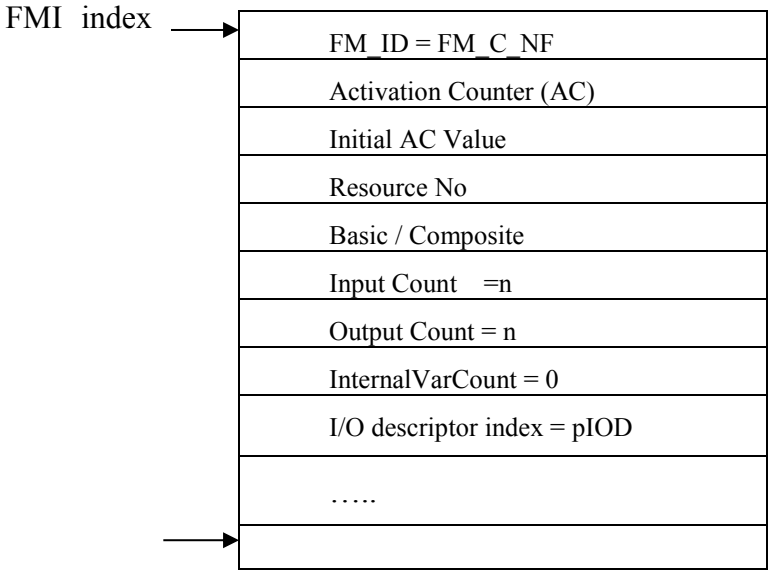
3.2 Драйвер за приемане и оповестяване на команда

Драйверът за оповестяване на команда C_NF (Command_NotiFication) капсулира блокиращата примитива на ОС *receive*( ), която връща като резултат идентификатора на най-приоритетното пристигнало съобщение. Драйверът сигнализира за получена команда чрез вдигане на съответен флаг – променлива от своята изходна аргументна зона, и нулира останалите. Входни аргументи са идентификаторите на приеманите команди.



● *Организация на структурите с данни на сигналния драйвер C_NF*

C_NF Function Module Instance (FMI) Invocation Block



C_NF Function Module Instance (FMI) – I/O Descriptor Block

	Variable Type	External/Local	Address	Rows	Columns	
pIOD →	LONG_TYPE		$\&ID_1$	1	1	I_1
	...	...	…	...	...	...
pIOD +n-2 →	LONG_TYPE		$\&ID_{n-1}$	1	1	I_{n-1}
pIOD +n-1 →	LONG_TYPE		$\&ID_n$	1	1	I_n
pIOD +n →	…_TYPE		&c_1			O_1
	…	…	…	…	…	…
pIOD +2n-2 →	…_TYPE		&c_(n-1)			O_{n-1}
pIOD +2n-1 →	…_TYPE		&c_n			O_n

C_NF FM Instance Argument Block

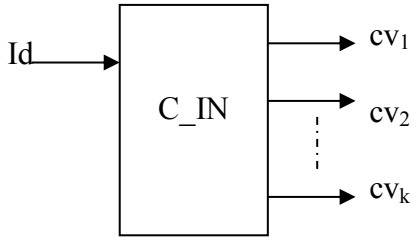
Inputs
Outputs

● *Псевдокод на сигналния драйвер C_NF*

```
void C_NF( pn ) {

     receivedID = receive();
       // receivedID - идентиф. на най-приоритетното съобщ.
     command_count = pn.inputCount;//бр.команди=бр.входове
     for(i=0;i<command_count;i++){//Обхождане на записите
     //във input зоната на вх/изх дескриптор на модула
          if(receivedID==getNextID())//Ако текущото
                                     //ID =receivedID?
             break;//Да -полученото ID е очаквано
     }
     for(j=0;j<command_count;j++){
          if(j==i)
               flag[j] = 1;
               // сигнализиране за получена команда
          else
               flag[j] = 0;
               //нулиране на останалите флагове(аргументи в
               //изх. зона)
     }
}
```

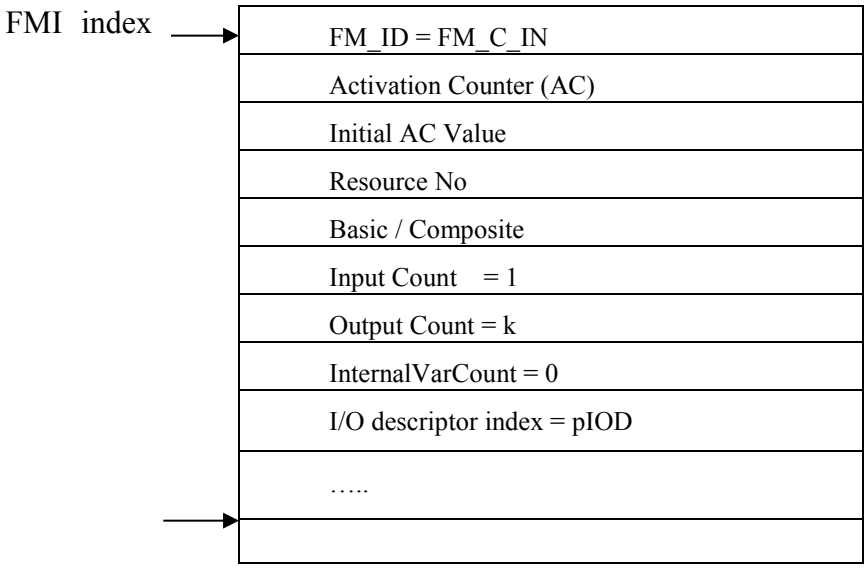
3.3 Входен драйвер

Входния драйвер е предназначен за прочитане и декомпозиране на получената команда на съставни променливи. Входен аргумент е идентификатора на командното съобщение.



● *Организация на структурите с данни на сигналния драйвер C_IN*

C_IN Function Module Instance (FMI) Invocation Block



C_IN Function Module Instance (FMI) – I/O Descriptor Block

	Variable Type	External/Local	Address	Rows	Columns	
pIOD →	…_TYPE		&ID	1	1	I_1
pIOD +1 →	…_TYPE		&cv_1			O_1
	…	…	…	…	…	…
pIOD +k-1 →	…_TYPE		&cv_(k-1)			
pIOD +k →	INT_TYPE		&cv_k			O_k

C_IN FM Instance Argument Block

Input
Outputs

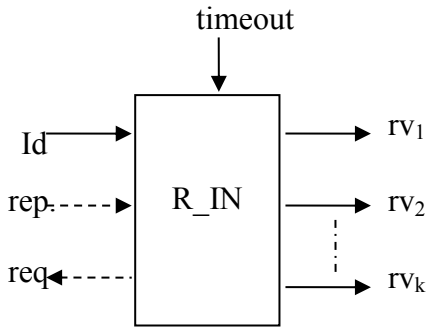
● *Псевдокод на C_IN*

```
void C_IN( pn ){

      &ID = pn.pIOD.addr;//Четене на ID от първия запис в
                         //FMI_IOD_Block
      buff = get_input_buffer(ID);//взимане на вх буфер
      decomposeMessage(buff);//декомпозиране на съобщ.от
                             //буфера
      fillOutputArgumentValues();//Попълване на
                                 //изх.аргументна зона

}
```

4. Драйвери за комуникация тип клиент-сървър

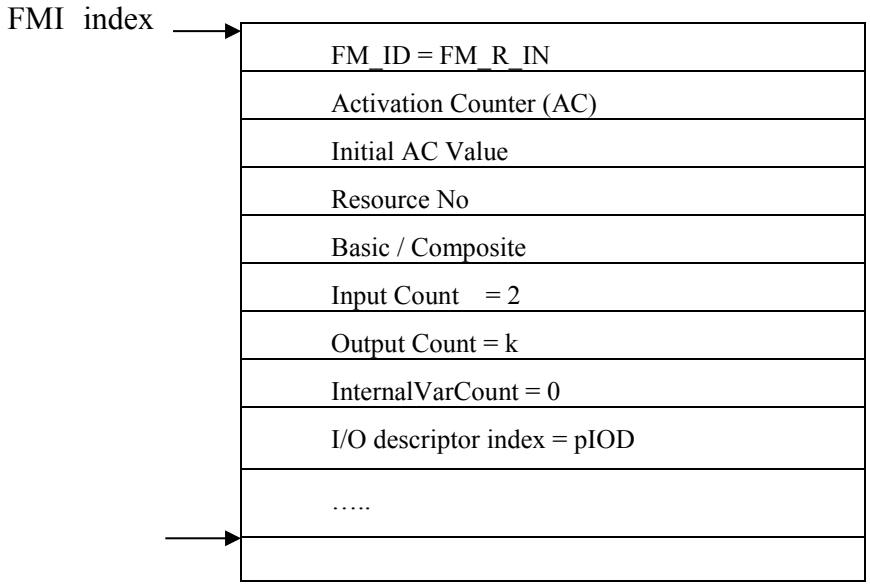
4.1 Драйвер за клиентска заявка

Драйверът съдържа блокиращата командата на ОС *request*(*var_name, timeout*), с която се изисква дадена променлива. При получаване на желаната променлива изпълнението продължава с декомпозирането ѝ на съставящи и записа им в изходната аргументна зона. Ако заявеното съобщение не пристигне в рамките на *timeout* интервал, ОС деблокира задачата.



● *Организация на структурите с данни на сигналния драйвер R_IN*

R_IN Function Module Instance (FMI) Invocation Block



R_IN Function Module Instance (FMI) – I/O Descriptor Block

	Variable Type	External/Local	Address	Rows	Columns	
pIOD →	…_TYPE		&TIMEOUT	1	1	
pIOD +1 →	INT_TYPE		&ID	1	1	I_1
	…_TYPE		&var_1			O_1
	…	…	…	…	…	…
pIOD +k →	…_TYPE		&var_(k-1)			
pIOD +k+1 →	…_TYPE		&var_k			O_k

R_IN FM Instance Argument Block

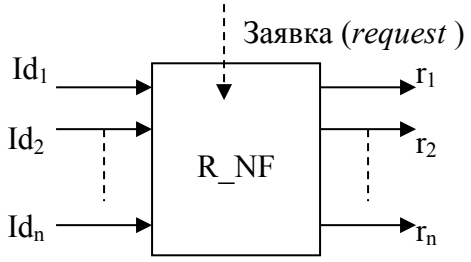
Inputs
Outputs

● *Псевдокод на R_IN*

```
void R_IN( pn ){
     pIOD = pn.IODescIndex;//Взимане на указателя към
                           //IODescriptorBlock
     &time_out = pn.pIOD.addr;//четене на time_out от
                              //входната аргументна зона
     &ID = (pn.pIOD+1).addr; //четене на ID от
                             //входната аргументна зона
     if(request(ID,time_out)== EXIT_OK)//Ако е успешно
                                       //получаването
          buff = get_input_buffer(ID);
          decomposeMessage(buff);//декомпозиране на
                                 //полученото съобщение
          fillOutputArgumentZone();//попълване на
                                   //изх.аргументна зона
     }

}
```

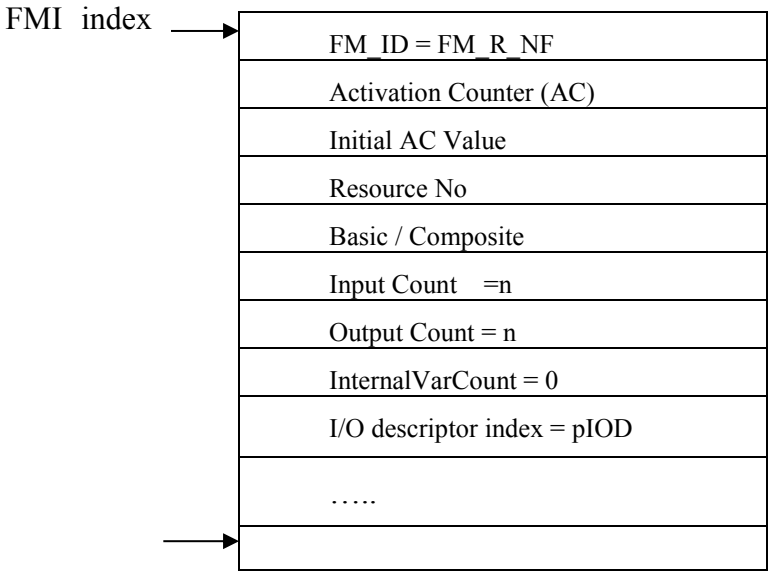
4.2 Драйвер за нотификация за постъпила заявка

Драйверът R_NF (Request_NotiFication) служи за оповестяване на получена заявка в рамките на двупосочното двуточково клиент-сървър взаимодействие. Той е напълно аналогичен на C_NF с тази разлика, че командата *receive*( ) се замества с *accept*( ).



● *Организация на структурите с данни на сигналния драйвер R_NF*

R_NF Function Module Instance (FMI) Invocation Block



R_NF Function Module Instance (FMI) – I/O Descriptor Block

	Variable Type	External/Local	Address	Rows	Columns	
pIOD →	LONG_TYPE		$\&ID_1$	1	1	I_1
	...	...	…	...	...	...
pIOD +n-2 →	LONG_TYPE		$\&ID_{n-1}$	1	1	I_{n-1}
pIOD +n-1 →	LONG_TYPE		$\&ID_n$	1	1	I_n
pIOD +n →	…_TYPE		&r_1			O_1
	…	…	…	…	…	…
pIOD +2n-2 →	…_TYPE		&r_(n-1)			O_{n-1}
pIOD +2n-1 →	…_TYPE		&r_n			O_n

R_NF FM Instance Argument Block

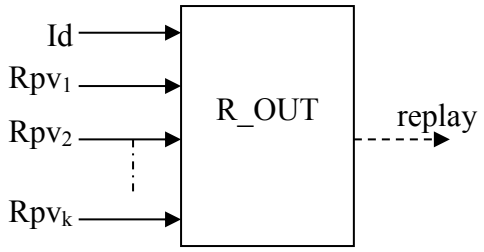
Inputs
Outputs

● *Псевдокод на сигналния драйвер R_NF*

```
void R_NF( pn ) {

     receivedID = accept();
      // receivedID - идентиф. на най-приоритетното заявка
     _count = pn.inputCount;//бр. заявки за нотификация
     for(i=0;i< _count;i++){
          if(receivedID==getNextID())
          //Търсене на receivedID във входната аргументна
          //зона на модула
             break;// Да полученото ID е очаквано
     }
     for(j=0;j<_count;j++){
          if(j==i)
               flag[j] = 1;
               //Вдигане на съответния флаг в изх. зона
          else
               flag[j] = 0;
               //нулиране на останалите
     }
}
```

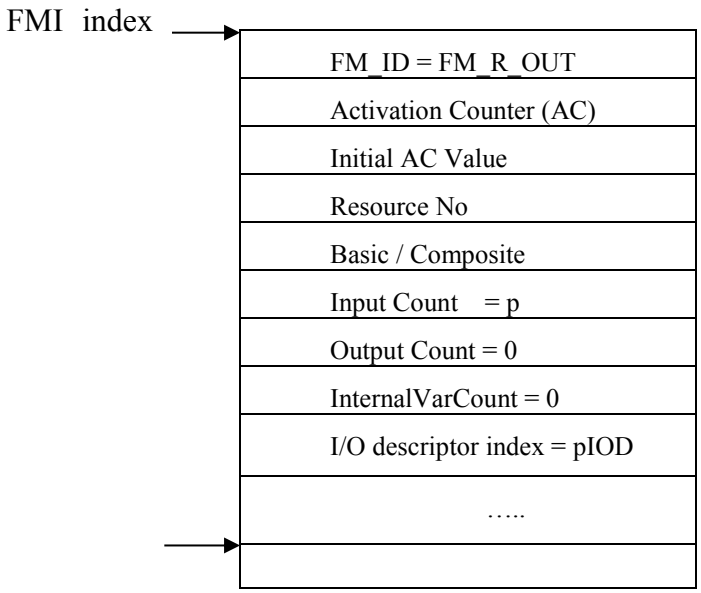
4.3 Драйвер за отговор на клиентска заявка

Драйверът R_OUT (Replay_variable_OUT)осигурява композирането и изпращането на отговор на клиентска заявка. Съдържа командата на ОС *replay*( ), която изпраща съобщението-отговор към клиента.



● *Организация на структурите с данни на сигналния драйвер C_OUT*

R_OUT Function Module Instance (FMI) Invocation Block



R_OUT Function Module Instance (FMI) – I/O Descriptor Block

	Variable Type	External/Local	Address	Rows	Columns	
pIOD →	…_TYPE		&Rpv_1			I_1
pIOD +1 →	…_TYPE		&Rpv_2			I_2
	…	…	…	…	…	…
pIOD +p-2 →	…_TYPE		&Rpv_n			
pIOD +p-1 →	INT_TYPE		&ID	1	1	I_p

R_OUT FM Instance Argument Block

Inputs

● *Псевдокод на сигналния драйвер R_OUT*

```
R_OUT( pn ) {

  p =  pn.inputCount;//бр.входни аргументи
  pIOD = pn.IODescIndex;//Указ.към IODescriptorBlock
  &ID = (pIOD + p - 1).Address ;//четене ID на съобщ.
  composeMessage();//композиране на отговора от
     //променливите във входната зона(IODescriptorBlock -
     //позиции от 0÷p-2)
  replay( ID );//отговор.
}
```

5. Интерпретатор на графа на преходите (State Transition Manager)

```
State_transition_manager is: {
        If (DEE_flag = 1) {
                DEE_flag=0;
                goto E;  // интерпретиране на текущото състояние
        }

        /* Определяна на текущото състояние */
        N:      if( current P_index != 0 ){

                // Изчисляване на предикатите
                execution_sequence_manager(current_P_index);
        }
                current = determine_current_state(current_Fa_index)


        E:  /* интерпретиране на текущото състояние*/

        //Инициализиране на deadline/period таймерите
                if( current_Deadline != 0 )
                        set_deadline(current_Deadline);
        if( current_Period !=0 )
                set_period(current_Period);

        if( current_Reaction != 0){

                /* изпълняване на реакцията */
                if(current_R_type =4 )
                        enqueue(cur_Reaction)
                else
                        execution_sequence_manager(cur_Reaction)

                /*повтаряне на последователности, ако прехода е спонтанен, в противен случаи
                изход */
                if (current_R_type =  1 ) goto N;
                else task_exit();

        }
}
```

6. Интерпретатор на сигналния граф (Execution Sequence Manager).

```
execution_sequence_manager is :
    {
        for all Mi ∈ NeM0 : Mi → ER_FIFO;

        do{
                fetch execution request from ER_FIFO;
                lock(recourse); // specified in FM inv Blk – if any
                execute Function module;
                unlock(resource); // if locked any
                for all Mj ∈ Next(executed module){
                      decrement activation counter ;
                       if (activation_counter =  0)
                            insert request into ER_FIFO;

                }
        }until (ER_FIFO_is_empty);
    }
```

7. Библиотека на функционалните модули

7.1 FILT – Функционален модул реализиращ експоненциален цифров филтър.

Алгоритъм за работа

$y(k) = a * (y(k-1) - x(k)) + x(k)$

- входни параметри
  - x(k) - технологичната величина на входа на модула.
  - a - коефициент на филтъра
- изходни параметри
  - y(k) - технологичната величина след филтрацията
- работни параметри
  - y(k-1) - стойност на технологичната величина в предния такт.
- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
Double	External	x(k)	1	1	I_1
Double	External	a	1	1	I_2
Double	Local	y(k)	1	1	O_1
double	Local	y(k-1)	1	1	V_1

7.2 LC – Функционален модул за проверка за долна и горна граници

Алгоритъм за работа:

Проверява се дали технологичната величина се намира в технологичните граници , и ако не е така се вдига съответния флаг за нарушена долно или горна технологична граница. Проверка се прави и за нарушаване и на аварийни граници. В този случаи се вдига и флаг за нарушена аварина граница.

flagDA =1	y(k) < y_min_a
flagD =1	y(k) < y_min
flagU =1	y(k) > y_max
flagUA =1	y(k) > y_max_a
Allflags=0	y_min <= y(k) <= y_max

- входни параметри
  - y(k) - технологичната величина на входа на модула.
  - y_min_a - долна аварийна граница
  - y_min - долна технологична граница
  - y_max - горна технологична граница
  - y_max_a -- горна аварийна граница
- изходни параметри
  - flagDA - флаг за долна аварийна граница
  - flagD - флаг за долна технологична граница
  - flagU - флаг за горна технологична граница
  - flagUA - флаг за горна аварийна граница
- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
double	External	y(k)	1	1	I_1
double	External	y_min_a	1	1	I_2
double	External	y_min	1	1	I_3
double	External	y_max	1	1	I_4
double	External	y_max_a	1	1	I_5
short	Local	FlagDA	1	1	O_1
short	Local	FlagD	1	1	O_2
short	Local	FlagU	1	1	O_3
short	Local	FlagUA	1	1	O_4

7.3 GC – градиентна проверка

Алгоритъм за работа

Проверява се условието 0.003*y_min < y < 0.997*y_max, и ако това условие не е изпълнено се вдига flagF и работа на модула приключва. Ако е изпълнено, то образува разликата $\Delta y(k) = abs|y(k) - y(k-1)|$ и се проверява на нарушаване на долен и горен градиент.

flagF =0 0.003*y_min < y < 0.997*y_max
flagU=0 ,flagD=0 d_y_min <= d_y(k) <= d_y_max
flagU=1 ,flagD=0 d_y(k) > d_y_max
flagU=0 ,flagD=1 d_y_min > d_y(k)

- входни параметри
  y(k) - технологичната величина на входа на модула.
  d_y_min - долна граница на градиента
  d_y_max - горна граница на градиента
  y_min - минимална стойност на технологичната величина
  y_max - максимална стойност на технологичната величина
- изходни параметри
  flagU - флаг за нарушена горна граница на градиента
  flagD - флаг за нарушена долна граница на градиента
  flagF -
- работни параметри
  y(k-1) - стойност на технологичната величина в предния такт.
- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
Double	External	y(k)	1	1	I_1
Double	External	d_y_min	1	1	I_2
Double	External	d_y_max	1	1	I_3
Double	External	y_min	1		I_4
Double	External	y_max	1	1	I_5
Short	Local	flagU	1	1	O_1

Short	Local	flagD	1	1	O_2
Short	Local	flagF	1	1	O_3
Double	Local	y(k-1)	1	1	V_1

7.4 SPG – Функционален модул за комутиране на отделните източници на задаващо въздействие.

Алгоритъм за работа

```
                | K=1  => y(k)
y_out(k)=       | K=2  => a*( y(k-1) - y_param(k)) + y_param(k)
```

входни параметри

y(k) - технологичната величина на входа на модула.

К -коефициент определящ източника на задаващото въздействие.

y_param - параметричен вход на задаващото въздействие

a - коефициент на демфера

- изходни параметри

y_out - технологичната величина след линеаризацията

- работни параметри

y(k-1) - стойност на технологичната величина в предния такт.

- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
Double	External	y(k)	1	1	I_1
Short	External	К	1	1	I_2
Double	External	y_param	1	1	I_3
Double	External	a	1	1	I_4
Double	Local	y_out	1	1	O_1
Double	Local	y(k-1)	1	1	V_1

7.5 LIM - Функционален модул реализиращ ограничаване на технологичната променлива

Алгоритъм за работа

Проверява се дали технологичната величина се намира в минималната и максималната граници . Ако е нарушена някоя от тях се вдига съответния флаг, като изхода приема граничната стойност на технологичната величина.

```
flagUp=0,flagDown=0 | y_min <= y(k) <= y_max
                    | y_out = y(k)

flagUp=0,flagDown=1 | y(k) < y_min
                    | y_out = y_min

flagUp=1,flagDown=0 | y(k) > y_max
                    | y_out = y_max
```

входни параметри

y(k) - технологичната величина на входа на модула.

y_min - минимална стойност на технологичната величина.

y_max - максимална стойност на технологичната величина.

- изходни параметри

y_out - технологичната величина след проверката

flagUp - флаг за нарушена горна граница

flagDown - флаг за нарушена долна граница

- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
double	External	y(k)	1	1	I_1
double	External	y_min	1	1	I_2
double	External	y_max	1	1	I_3
Double	Local	y_out	1	1	O_1
Short	Local	FlagUp	1	1	O_2
Short	Local	FlagDown	1	1	O_3

7.6 ENC - Функционален модул реализиращ сравнение между две флагови стойности.

Алгоритъм за работа

```
          | aut=1,man=0  => y(k) = 1
  out=    |
          | aut=0,man=1  => y(k) = 2
```

- входни параметри

aut - флаг

man –флаг

- изходни параметри

out - изходя на модула , описващ състоянието на входните флагове.

- Таблица на параметрите на модула. (Function module Instance Descriptor Table)

Variable Type	External/Local	Address	Rows	Cols	
short	External	Aut	1	1	I_1
short	External	Man	1	1	I_2
short	Local	out	1	1	O_1

П2. Приложение 2

Измерване на времето за изпълнение на елементи от комуникационната подсистема за целите на анализа на диспечируемостта

(извадка от ръкопис, предаден за рецензия в IEE Software Engineering)

1. Входящ протокол

За целите на работа в реално време с гарантирана диспечируемост на комуникационната среда е реализирано изменение на оригиналния Ethernet протокол. В нормален режим протоколната програма обработва постъпващите заявки синхронно. Това е неприемливо за целите на приложенията, работещи в реално време, защото предполага непредвидимо закъснение при обработката на постъпващите данни. Поради това е разработен видоизменен MAC протокол на базата на цикличен маркер (token-ring MAC protocol). Разпознаването на типа на входния пакет става на нивото на входната програма за обработка на прекъсване от комуникационния адаптер (фиг. П2.2). При установяване на приемане на пакет от разработения нов протокол, той се обработва веднага. В противен случай пакетите се обработват по стандартния начин.

2. Изходящ протокол

При регистриране на изходящо съобщение, генерирано от средата за работа в реално време, то се записва в приоритетно-подредена изходяща опашка. Изходният драйвер е модифициран така, че да може да поддържа новата модификация на маркерен протокол (фиг. П2.1). При приоритетната подредба на изходящата опашка, най-високоприоритетното съобщение се изпраща първо. Всеки пакет приема от 0 до *n* съобщения с фиксирана големина.

3. Анализ на диспечируемостта

За целите на анализа на диспечируемостта е разработена методика и апаратура за провеждане на измервания на времето за транспорт и обработка на съобщенията в мрежата за реално време.

Граф на състоянията на драйвера при разпознаване на типа на входния пакет е показан на фиг. П2.3.

Два варианта на времедиаграми на процеса на предаване на съобщение (според използваните синхронизационни механизми в драйвера) са показани на фиг. П2.4 и фиг. П2.5.

Измерването са проведени на базата на методиката, разгледана в гл. 5. За целта се използва измерваща платка с 5-канален таймер с разрешаваща способност 0.125 μs. Всяка програма (или елемент от програма), която стартира даден етап от предаването/приемането на съобщение, активизира дискретен сигнал, свързан към входната логика на таймерната платка. Това води до пускане и съответно спиране на даден брояч или едновременно спиране на един брояч и пускането на друг такъв. Чрез този механизъм се осигурява елиминиране на програмните закъснения при измерванията.

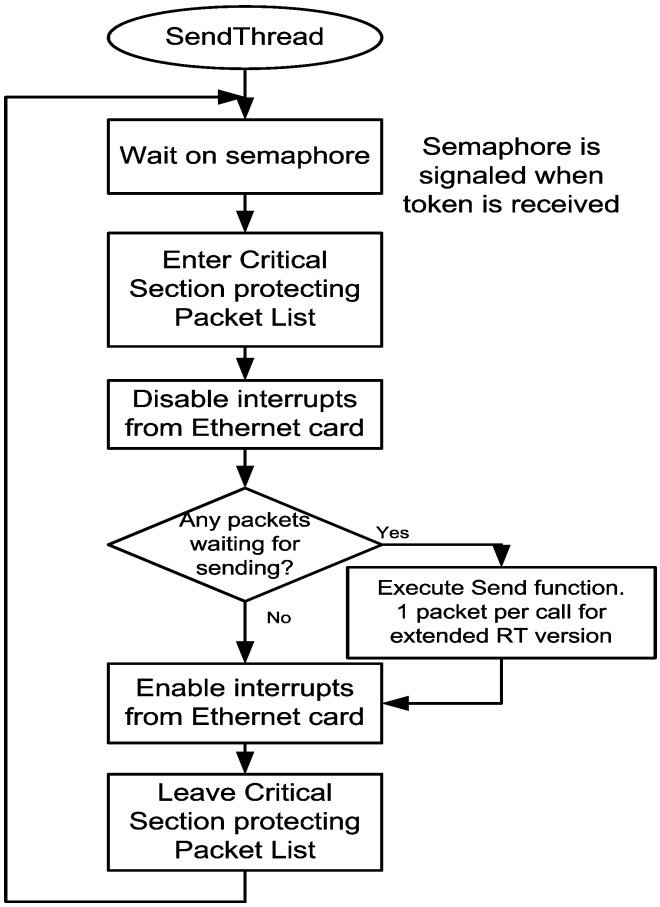
Използвани съкращения в таблиците и времедиаграмите:

ISR (interrupt service routine)	- програма за обработка на прекъсване
OSLAT (operating system's latency)	- закъснение на операционната система
IST (interrupt service thread)	- нишка за обработка на заявка за прекъсване
SendThread (sending thread execution)	- време за изпълнение на изходния драйвер (нишка на драйвера)
Transmit time (physical transmission time - send function)	- време за физически транспорт (изходящо)
ISTTKN – interrupt service thread operations to the point where enabling token processing semaphore is released	- работа на нишката за обработка на заявка за прекъсване до освобождаване на семафора, защитаващ обработката

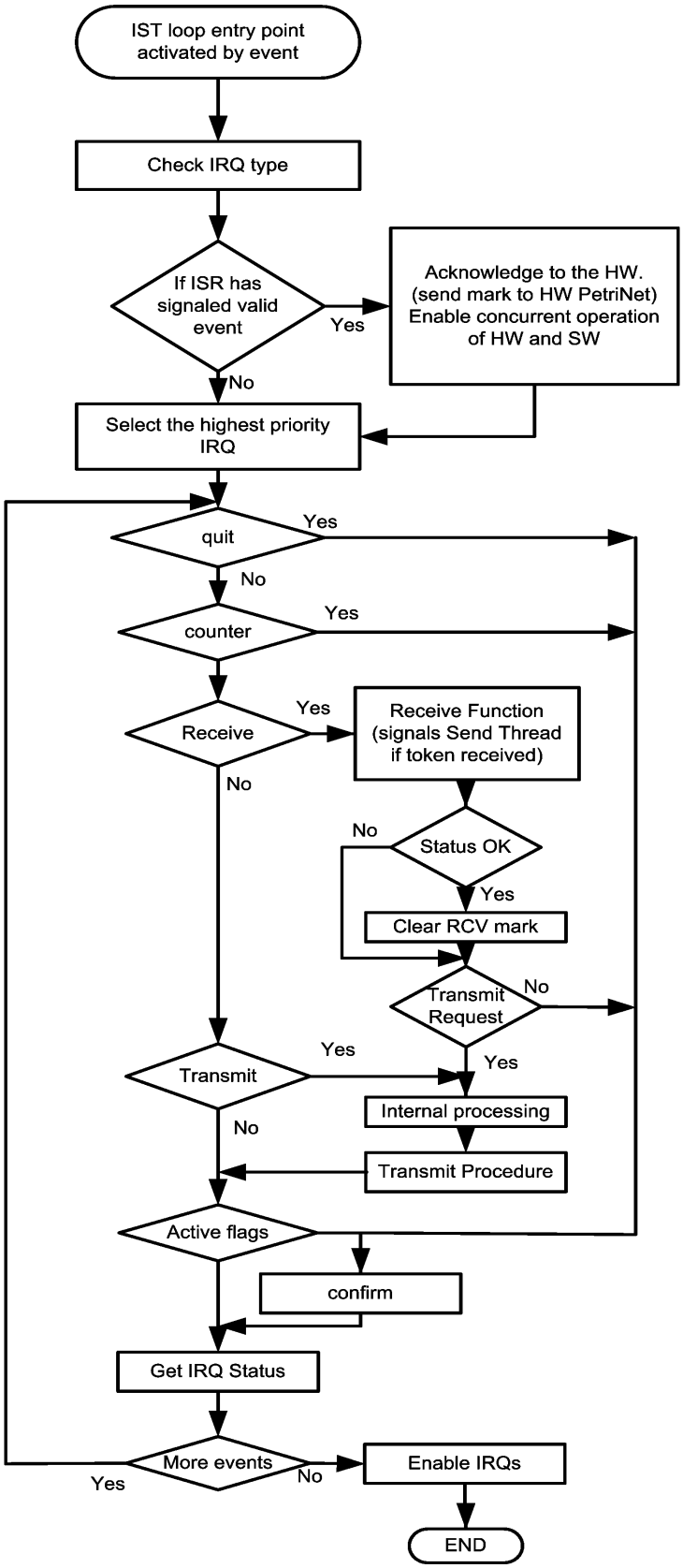
Резултатите от измерванията на максималните, минимални и съответните претеглени средни времена показани в табл. 1. На фиг. П2.6 и на фиг. П2.7. е показано разпределението на измерените времена в графична форма като брой повторения / време за изпълнение.

	Време за изпълнение max [μs]	Време за изпълнение min [μs]	Време за изпълнение – претеглено средно [μs]	ст. отклонение
OSLAT	53	41	44.578029	1.446756
IST	83	59	Фиг. П2.6.	
ISTTKN	456	44	Фиг. П2.7.	
OSLAT2	64	28	31.247062	0.972091
OSLAT1	58	45	47.928696	1.372155
SendThread	41	30	32.467073	0.878602
Transmition time	108	97	99.839842	1.212466

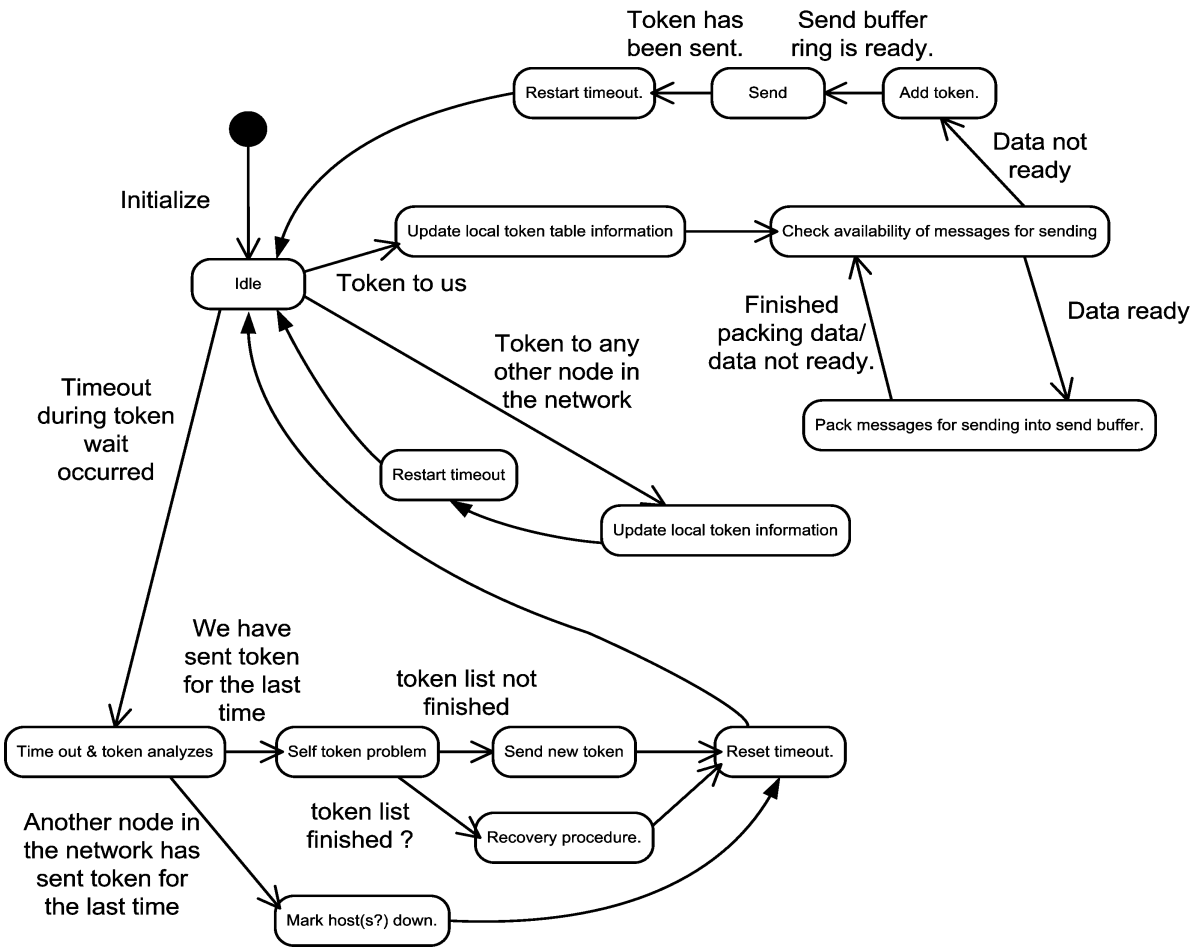
Табл. 1. – Измерени времена за работа на драйвера в отделните етапи съгласно времедиаграмите



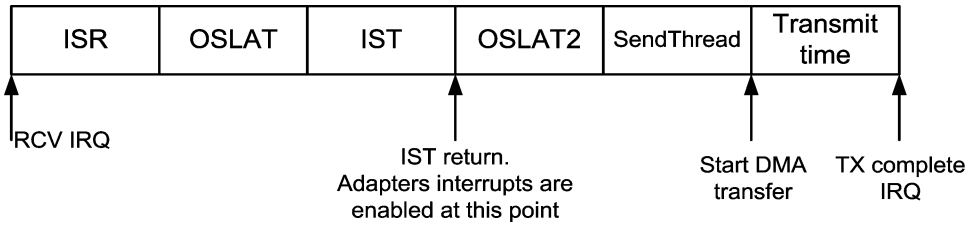
Фиг. П2.1. Модифициран Ethernet драйвер – опростена блок-схема на изходната нишка



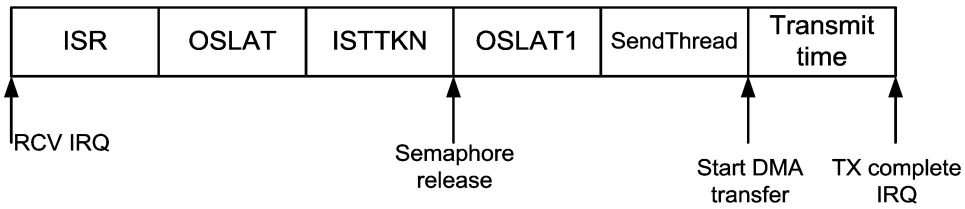
Фиг. П2.2. Модифициран Ethernet драйвер – опростена блок-схема на входната нишка



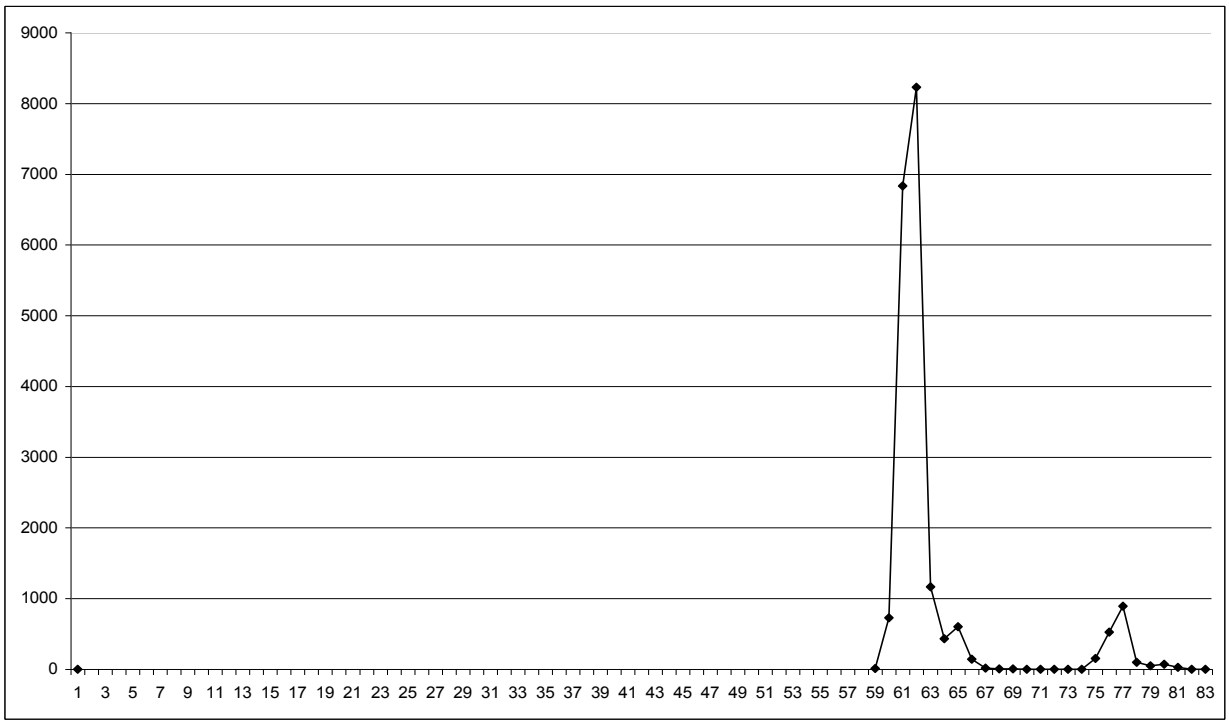
Фиг. П2.3. Модифициран Ethernet драйвер –диаграма на състоянията при обработка на маркер



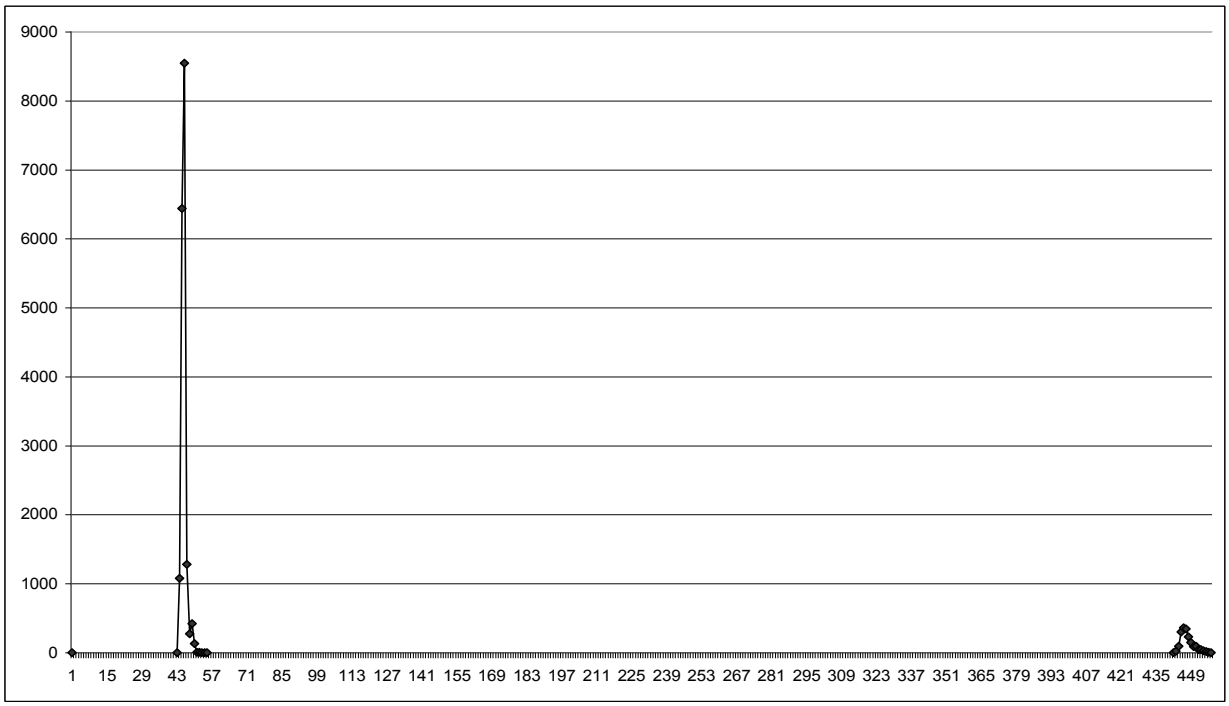
Фиг. П2.4. Модифициран Ethernet драйвер – времедиаграма на работа – вариант 1



Фиг. П2.5. Модифициран Ethernet драйвер – времедиаграма на работа – вариант 2



Фиг. П2.6. Модифициран Ethernet драйвер – разпределение на измерените времената – вариант 1



Фиг. П2.7. Модифициран Ethernet драйвер – разпределение на измерените времената – вариант 2