Проект по компютърни системи за управление на роботи

Език за улесняване на програмиране на вградени устройства

Галин Симеонов Ф.Н. 81635
gtsimeonov@uni-sofia.bg

1. Увод

При програмирането на вградени устройства, някои от проблемите се моделират чрез автомати. Също е практика да се програмира на езици от ниско ниво, като **C**, при които ръчното имплементиране на автомати с каквито и да е размери често е неинтуитивно, предразполагащо към грешки и трудоемко. С цел да помогна разработването на програми, чийто модел е ориентиран около състоянията на системата, предлагам и имплементирам експериментален миниатюрен език транспилиращ до **C**.

За цели имам той да е прост, интуитивен и евентуално лесен за генериране от инстурменти с графичен интернфейс[1] .

2. Описание на езика

2.1. Общ поглед

Една 'програма' съдържа един или повече 'машини', които действат като контейнери за състояния,събития и преходи. Всяка машина си има име и има отделно пространство за имена на състояния и събития. Чрез преходите може да се извикват външни функции. Ето най-простият пример за това как изглежда кодът:

```
machine light_bulb
[
	states [ ON , OFF ];
	events [ TURN_ON , TURN_OFF , SWITCH_STATE ];
	starting on OFF;
	transitions
	[
		from ON to OFF on event TURN_OFF;
		from ON to OFF on event SWITCH_STATE;

		from OFF to ON on event TURN_ON;
		from OFF to ON on event TURN_ON;
	];

];
```

1. Макар, че директното генериране на код без този език да действа като посредник би било по-смислено.

След транспилация се генерират 3 файла - xxx.h xxx.c и xxx_external.h, който съответно съдържат декларациите на служебните функции, тяхната имплементация и декларациите на външните функции, които са използвани от някой преход. Подаването на събития към тези 'автомати' става посредством служебната функция - **push_event_to_machine**

Горният пример не е много функционален, защото комуникацията е само в една посока. За да може да връща информация и да има функционалност, на всеки преход може да се сложат функции, които да бъдат изпълнени преди преходът да е завършил[2] . Така горният пример може да бъде преработен да вика функция, която действително да включва и изключва някаква лампа:

```
machine light_bulb
[
	states [ ON , OFF ];
	events [ TURN_ON , TURN_OFF , SWITCH_STATE ];
	starting on OFF;
	transitions
	[
		from ON to OFF on event TURN_OFF
				execute light "off";
		from ON to OFF on event SWITCH_STATE
				execute light "off";

		from OFF to ON on event TURN_ON
				execute light "on";
		from OFF to ON on event TURN_ON
				execute light "on";
	];

];
```

Тук също е демонстрирано как се подават аргументи към тези функции. Символните низове са избрани като единствен начин да се подават аргументи, защото е предвидено да се транспилира до езици, различни от **C**, а въвеждането на типова система или нещо, което да описва различните аргументи би натоварило езикът твърде много и би донесло само минимална печалба.

Възможно е да има повече от една 'машина' в кодът. Например, нека към горния пример добавим:

```
machine light_controler
[
	states [ STATIC , BLINKING ];
	events [ SIGNAL , GO_STATIC, START_BLINKING ];
	starting on STATIC;
	transitions
	[
		from STATIC to BLINKING on event START_BLINKING;
```

2. Това свойство е важно и авторът се е стремил да го запази. Това позволява, например, да се подават събития от функции изпълнени по време на преход

```
        from BLINKING to STATIC on event GO_STATIC;

        from BLINKING to BLINKING on event SIGNAL
                execute prod_light_bulb;

    ];

];

machine timer
[
    states [ ON , OFF ];
    events [ TICK , START , STOP ];
    starting on OFF;
    transitions
    [
        from ON to OFF on event STOP;
        from OFF to ON on event START
            execute prod_timer;
        from ON to ON on event TICK
            execute prod_timer | prod_light_controler;
    ];

];
```

Тук може да се види и 'навързване' на различни функции. timer 'машината' праща сигнал до себе си и до light_controler, а light_controler праща сигнал до light_bulb. Тук можем да прескочим предаването на събития между 'машините' чрез използването на **if**. Променяме третия преход от timer:

```
    from ON to ON on event TICK
        if(light_controler.BLINKING)
            execute prod_timer | prod_light_bulb;
```

Това е условно изпълнение на командите, преходът се случва и състоянието е променено независимо от истинността на условието. За реализацията на условен преход, в езикът има **granted** ключовата дума. Горното може да се опитаме да имплементираме по следния начин:

```
    from ON to ON on event TICK granted (light_controler.BLINKING)
            execute prod_timer | prod_light_bulb;
```

Тук има проблема, че timer спира да работи ако light_controler не е в състояние BLINKING, защото преходът няма да се случи и **execute prod_timer** няма да се изпълни. [3]

prod_timer,**prod_light_bulb**,**prod_timer** и **light** са функции, чиято имплементация трябва да бъде предоставена от програмиста. Всички външни функции се събират и се записват в генерирания xxxx_exter.h файл. Този пример би генерирал:

```
#ifndef XXXX_EXTERN_H
```

3. Това, че този пример може да бъде имплементиран на **C** под 10 реда, е забелязано от автора.

```
#define XXXX_EXTERN_H XXXX_EXTERN_H

extern machine_buffer_t* light(machine_buffer_t *arguments,machine_buffer_t *input);
extern machine_buffer_t* prod_light_bulb(machine_buffer_t *arguments,machine_buffer_t *input);
extern machine_buffer_t* prod_light_controler(machine_buffer_t *arguments,machine_buffer_t *input);
extern machine_buffer_t* prod_timer(machine_buffer_t *arguments,machine_buffer_t *input);

#endif
```

Ето и една примерна тяхна импелементация заедно с **main** функцията:

```
machine_buffer_t* light(machine_buffer_t *arguments,machine_buffer_t *input)
{
	printf("light %s0,arguments->buffer);
	return NULL;
}
machine_buffer_t* prod_light_bulb(machine_buffer_t *arguments,machine_buffer_t *input)
{
	push_event_to_machine(light_bulb,light_bulb_EVENT_SWITCH_STATE,NULL);
	return NULL;
}
machine_buffer_t* prod_light_controler(machine_buffer_t *arguments,machine_buffer_t *input)
{
	push_event_to_machine(light_controler,light_controler_EVENT_SIGNAL,NULL);
	return NULL;
}
machine_buffer_t* prod_timer(machine_buffer_t *arguments,machine_buffer_t *input)
{
	push_event_to_machine(timer,timer_EVENT_TICK,NULL);
	sleep(1);
	return NULL;
}
int main()
{
	push_event_to_machine(light_controler,light_controler_EVENT_START_BLINKING,NULL);
	push_event_to_machine(timer,timer_EVENT_START,NULL);
	return 0;
}
```

Това ни дава изхода:

```
light on
light off
light on
light off
light on
light off
light on
light off
light on
...
...
```

2.2. Формално описание на езика

Със затъмнените думи и символи обозначавам думи и символи, които трябва да се интерпретират директно.

2.2.1. Програма

програма : машина [програма]

Програмата е поредица от машини. Всяка машина има уникално име.

2.2.2. Машина

машина : ***machine*** име ***[*** вътрешна част на машината ***] ;***

вътрешна част на машината : ***states [*** поредица от състояния ***] ;*** [вътрешна част на машината]
events [поредица от събития ***] ;*** [вътрешна част на машината]
transitions [поредица от преходи ***] ;*** [вътрешна част на машината]
starting on име на състояние ***;*** [вътрешна част на машината]

В една машина може да се срещне само веднъж декларация на състоянията, събитията, преходите и посочване на стартиращо състояние. Декларацията на стартиращо състояние трябва да е след декларацията на състоянията. То трябва да е сред декларираните състояния. Декларацията на преходите трябва да е след декларациите на състоянията и на събитията. Сред декларираните състояния и събития не трябва да има повтарящи се. Сред преходите не трябва да има две различни, които излизат от едно състояние и имат за етикет едно събитие.

2.2.3. Преход

преход : ***from*** име-на-състояние
to име-на-състояние ***on*** име-на-събитие [опашка-на-прехода] ***;***

опашка-на-прехода : [***granted*** израз] [условно-изпълнение]

условно-изпълнение : ***if*** израз условно-изпълнение [***else*** условно-изпълнение]

условно-изпълнение : ***execute*** опашка на изпълнението

опашка-на-изпълнението : име-на-външна-функция **"**символен-низ**"** [**|** опашка-на-изпълнението]

Ако изразът след ***granted*** е истина то преходът се реализира и командите в условното изпълнение се изпълняват спрямо семантиката, иначе преходът не се изпълнява и опашката на преходът не се изпълнява. Ако изразът след ***if*** е истина то условното изпълнение след изразът се изпълнява, иначе, ако има ***else*** съответстващ на ***if***-а то условното узпълнение след ***else*** се изпълнява. Ако условното изпълнение е от типа започващ с ***execute*** то външните функции се изпълняват в ред на срещане като изходът на всяка се подава на следващата. Изходът на последната изпълнена функция се изхвърля.

2.2.4. Израз

израз : израз-или

израз-или : израз-и [**||** израз-или]

израз-и : израз-не [***&&*** израз-и]

израз-не : [***!***]базов-израз

базов-израз : име-на-машина**.**име-на-състояние | ***(***израз***)***

Израз може да се оцени до истина или лъжа. Логическите оператори имат обичайната семантика. В базовия израз е позволено да се посочват състояния на други машини, но не е позволено да се посочват състояния на машината, в която се намира изразът. Тези посочвания се оценяват до истина ако посочената машина е заела посоченото състояние. Могат да се посочват имена на машини, които са декларирани след сегашната, но те трябва да съществуват. Това важи и за състоянията, те трябва също и да принадлежат на посочената машина.

3. Детайли на имплементацията

За да се реализира обмена на информация между генерирания код и написания, е дефинирана структура **machine_buffer_t**, в която се записват данните и техният размер. Генерират се и няколко помощни функции, които улесняват работата с такива структури. За да се запази свойството - командите на преходът да се изпълнят преди състоянието да се смени, се използва опашка, в която се записват

4. Описание на командните аргументи

Имплементацията на този език предадена от автора приема следните аргументи.

--print-tokens
Извежда разпознатите лексеми
--print-ast
Извежда разпознатите структури в текста. (Абстрактното синтактично дърво)
-o име-на-файл | --output име-на-файл
Посочва префиксът на генерираните файлове. Например **xxxx.h xxxx.c xxxx_external.h**.
--extern-mutex
Добавя мутекс преди и след подаването на събитие. Този мутекс трябва да се имплементира външно.
--extern-queue
Дава възможност на програмиста да даде собствена имплементация на опашката използвана при задържането на събития.
--extern-buffer
Дава възможност на програмиста да даде собствена имплементация на структурата използвана за пренос на данни.

5. Забележки

За имплементацията е използвана само стандартната **C** библиотека, което би помогнало този транспилатор да бъде компилиран до много операционни системи.

6. Бъдещи насоки

1) Да се добави ключова дума **signal** която да праща сигнал до определена машина, за да не трябва да го имплементира програмистът.
2) Да се добави семанитка за състояния и събития, като например - ’при пристигане до това състояние изпълни ... ’.
3) Да се направи така, че отделните машини да могат да бъдат на различни физически устройства, т.е. да бъде направен разпределен.
4) Да се направи на пълен език за програмиране.