Obelix Configuration Tool Front-End*

Anders Davidsson
adn99006@student.mdh.se

July 18, 2000

*Part of the Asterix frame-work

Contents

1 Introduction

This is a description of the configuration tool front-end in Obelix Development Environment. The Obelix grammar specification is also included.

The front-end consists of a scanner and parser configured for the Obelix configuration language. The front-end scans the configuration file for tokens that is passed on, with attributes, to the parser. The parser controls the syntax and apply semantic actions and controls to the scanned file. The semantic actions consists of a name-analysis [], using symbol-tables, and building of a intermediate representation. The name-analysis makes sure that all used identity's are declared and only used once in a mode.

The intermediate representation is a syntax tree consisting of nodes for the different building blocks of the grammar. A more solid description and a programmer manual is found in []

There are different semantic controls and some calculations applied to the different parts of the syntax tree, these are described later in this document. The Obelix configuration language is specified by a context free grammar.

This document describes the scanner and the parser implementation and the semantic controls and calculations that are applied to the intermediate representation. It also specifies the grammar and the last section is an users manual for writing configuration files.

2 Implementation of the scanner and the parser

To build the scanner and the parser we utilized the reliable GNU tools Flex and Bison []. Bison is a general-purpose parser generator that converts a grammar description for an LALR(1)[1] context-free grammar into a C program to parse the grammar. Flex is a tool for generating programs that performs pattern matching on text (scanning). Bison gives the opportunity to include semantic actions during the parsing and a combination of Flex and Bison is our configuration tool front-end. The implementation is done by writing input files to Flex and Bison. These files sets up the configuration of our scanner and parser. The Flex input file specifies which tokens to deliver, and with which attributes, to the parser. The Bison input file is a grammar specification along with semantic actions. The semantic actions, performed in parallel with the parsing, are name analysis and building of an internal representation.

3 Semantic controls and computations

The semantic controls and calculations are applied to the different parts of the syntax tree.

3.1 Tasks

For each task triggered by a signal, the signal is examined, and if the current task is a user, nothing is done. If the signal does not exist or if the task is not among the users of the signal an error is generated.

The priority's are checked out. All tasks are assigned unique priority's by Obelix. All soft tasks are assigned lower priority's than the hard tasks. The semantic for the user is to assign each task a unique priority, but if he gives two or more tasks from the same class (hard/soft) equal priority Obelix will differentiate them and generate a warning. Obelix assigns priority's from 0 to $no\_tasks - 1$ on the basis of the user assignment. 0 is the lowest priority and the task assigned priority 0 will be treated as the idle task by the kernel.

A warning is generated if the stack size of a task is declared to be less than 10.

3.2 Semaphores

Obelix makes sure that all users of semaphores are declared as tasks in the system. A semaphore can only be accessed by tasks from the same class (hard/soft), if it is accessed by tasks from booth classes an error is generated. The semaphore ceiling of each semaphore is calculated.

3.3 Signals

Obelix makes sure that all users of all signals in the system are declared as tasks in the system.

[1] LookAhead Left-to-right Rightmost-derivation

3.4 Wait-free communication

Obelix makes sure that all users of wait-free communication channels are declared as tasks in the system. The number of buffers at each wait-free communication channel is checked. If the number is less than $R+2$ an error is generated. If no number is given, the number of buffers are calculated to $R+2$. R stands for the number of readers of the wait-free channel.

4 Grammar specification

This is the context free grammar for the Obelix configuration language. The grammar is a 4-tuple:
`(V, T, P, S)`

- `V` is syntactic variables, nonterminals, that denote sets of strings.

```
V = { file, systemmode, ram, modes, mode, resolution, tasks, hardtasks,
      softtasks, task, activator, args, error_routine, resources,
      communication, waitfrees, waitfree, num_buf, readers, reader,
      synchronization, signals, signal, users, user, semaphores, semaphore }.
```

- `T` is terminals, basic symbols from which strings are formed, The word "token" is a synonym for "terminal".

```
T = { SYSTEMMODE, SYSMODE, RAM, INT_CONST, MODE, ID, RESOLUTION,
      HARD_TASK, SOFT_TASK, ACTIVATOR, OFFSET, DEADLINE, PRIORITY,
      ROUTINE, ARGUMENTS, ERR_ROUTINE, STRING_CONST, WAITFREE, WRITER,
      TYPE, NUM_BUF, READER, SIGNAL, USER, SEMAPHORE, STACK }.
```

- `P` is productions which specifies the manner in which the terminals and nonterminals can be combined to form strings. `P = { See figure 1 }`[2].
- `S` is the start symbol of the grammar.

```
S = { file }.
```

5 How to write configuration files

This section is a brief description of how the configuration file should be written. Each part of the configuration file is described and examples are given. At the end, a complete example of a configuration file is presented.

5.1 System

The first thing to define in the file is which system mode to run. There are two possibilities NORMAL or TEST. NORMAL is used when the system should run sharp and TEST is used when a target system is measured. The amount of ram available to the system should also be described. The syntax is:

```
SYSTEMMODE = NORMAL;
RAM = 512000;
```

5.2 Mode

It is now time to enumerate the modes. There must be, at least, one mode in the system. A mode consists of resolution, tasks and resources. The resolution is how often the timer interrupt should be triggered. Tasks is a enumeration of all tasks that will operate in the mode, first all hard tasks and then all soft tasks. Resources is communication channels and synchronization resources. The order in which the different parts of the mode are specified must be the same as in this description.

```
MODE mode_1{
     resolution
     tasks
     resources
};
```

Resolution is defined by:

```
RESOLUTION = 1000;
```

[2]epsilon means empty rule

```
file            ->  systemmode ram modes
systemmode      ->  SYSTEMMODE = SYSMODE;
ram             ->  RAM = INT_CONST;
modes           ->  modes mode
                 |  mode
mode            ->  MODE ID { resolution tasks resources };
resolution      ->  RESOLUTION = INT_CONST;
tasks           ->  hardtasks softtasks
hardtasks       ->  hardtasks HARD_TASK task
                 |  epsilon
softtasks       ->  softtasks SOFT_TASK task
                 |  epsilon
task            ->  ID { ACTIVATOR = activator;
                         OFFSET = INT_CONST;
                         DEADLINE = INT_CONST;
                         PRIORITY = INT_CONST;
                         STACK = INT_CONST;
                         ROUTINE = ID;
                         args
                         error_routine };
activator       ->  INT_CONST
                 |  ID
args            ->  ARGUMENTS = STRING_CONST;
                 |  epsilon
error_routine   ->  ERR_ROUTINE = ID;
                 |  epsilon
resources       ->  communication synchronization
communication   ->  waitfrees
waitfrees       ->  waitfrees waitfree
                 |  epsilon
waitfree        ->  WAITFREE ID { WRITER = ID;
                                  readers
                                  num_buf
                                  TYPE = STRING_CONST; };
num_buf         ->  NUM_BUF = INT_CONST;
                 |  epsilon
readers         ->  readers reader
                 |  reader
reader          ->  READER = ID;
synchronization ->  signals semaphores
signals         ->  signals signal
                 |  epsilon
signal          ->  SIGNAL ID { users };
users           ->  users user
                 |  user
user            ->  USER = ID;
semaphores      ->  semaphores sempahore
                 |  epsilon
semaphore       ->  SEMAPHORE ID { users };
```

Figure 1: Obelix configuration language.

5.3 Task

A task consists of:

ID - an identifier

ACTIVATOR - which defines how often or by which signal the task should be triggered

OFFSET - relative the activator

DEADLINE - relative activator

PRIORITY - should be unique for each task

STACK - size of the stack

ROUTINE - code to execute (start routine)

ARGUMENTS - optional arguments to the start routine

ERR_ROUTINE - optional error routine which is called by the system if something goes wrong.

The task assigned the lowest priority in the system will be the idle task. The syntax:

```
HARD_TASK ht_1{
     ACTIVATOR = 100;
     OFFSET = 0;
     DEADLINE = 50;
     PRIORITY = 10;
     STACK = 50;
     ROUTINE = ht_1_routine;
     ARGUMENTS = "9";
     ERR_ROUTINE = ht_1_error_routine;
};

SOFT_TASK st_1{
     ACTIVATOR = sig_1;
     OFFSET = 0;
     DEADLINE = 10;
     PRIORITY = 10;
     STACK = 100;
     ROUTINE = st_1_routine;
};

SOFT_TASK st_2{ /* idle task */
     ACTIVATOR = 0;
     OFFSET = 0;
     DEADLINE = 0;
     PRIORITY = 0;
     STACK = 100;
     ROUTINE = idle_routine;
};
```

5.4 Resources

The resources are communication channels and synchronization resources. First the communication in the system should be defined. For the moment there is only wait- and lock-free inter-process communication available. A wait-free channel is defined by:

```
WAITFREE w_1{
     WRITER = ht_1;
     READER = ht_2;
     NUM_BUF = 3;
     TYPE = "my_type";
};
```

WRITER and READER are task identity's, there could be more than one reader. NUM_BUF is the the number of buffers the channel should use (optional), default is the $numberofreaders + 2$. Type is the data type of the buffers.

The available synchronization resources are signals and semaphores. First all signals in the system are defined. A signal is defined by:

```
SIGNAL sig_1{
     USER = ht_1;
     USER = st_1;
};
```

USER is a task identity, there can be one or more users. There are some limitations in the kernel concerning the signals. An aperiodic task is not supposed to wait for any other signal accept the one it triggers on.

Semaphores are defined in the same way:

```
SEMAPHORE sem_1{
     USER = ht_1;
     USER = ht_2;
};
```

Users of the same semaphore must be of the same task class, that is hard or soft. A semaphore priority ceiling is calculated with respect to the users.

6 Complete example of a configuration file

A complete configuration file could look like this:

```
SYSTEMMODE = NORMAL;
RAM = 512000;
MODE mode_1{
     RESOLUTION = 1000;
     HARD_TASK ht_1{
          ACTIVATOR = 100;
          OFFSET = 0;
          DEADLINE = 50;
          PRIORITY = 10;
          STACK = 50;
          ROUTINE = ht_1_routine;
          ARGUMENTS = "1, 2, 3";
          ERR_ROUTINE = ht_1_error_routine;
     };
     HARD_TASK ht_2{
          ACTIVATOR = 50;
          OFFSET = 0;
          DEADLINE = 20;
          PRIORITY = 20;
          STACK = 50;
          ROUTINE = ht_2_routine;
     };
     SOFT_TASK st_1{
          ACTIVATOR = sig_1;
          OFFSET = 0;
          DEADLINE = 10;
          PRIORITY = 10;
          STACK = 50;
          ROUTINE = st_1_routine;
     };
     WAITFREE w_1{
          WRITER = ht_1;
          READER = ht_2;
          NUM_BUF = 3;
          TYPE = "my_type";
     };
     SIGNAL sig_1{
          USER = ht_1;
          USER = st_1;
     };
     SEMAPHORE sem_1{
          USER = ht_1;
          USER = ht_2;
     };
};
```