The can machine exercise (Introduction to SDL)

This document describes the introductory lab in the programming language SDL. You will have four hours to complete the lab. The lab is done in pairs and you don’t necessary need to do it with your group members.

Before the lab, you should have read the documents “Introduction to SDL” (sections 1-4.2.4, 7.1-7.4, 10) and “Introduction to the Telelogic Tau SDL Suite” (pages 8-12, 14-15, 17-18). You should also have read and understood the next section in this document, “An SDL exercise”.

The laboratory session is described in the last section “The can machine”.

An SDL exercise

This exercise is aimed at giving a first practical experience with SDL and MSC. The system we use is small enough to be described on paper, and thus do not require the use of tools. Before the exercise you should have read the chapters in the reading list on the previous page. After completing this exercise you will have practice in the basic SDL and MSC constructs.

Introduction to SDL

SDL (Specification and Description Language) is a graphical high-level programming language to write C code in a fashion that is easily documented and organized. The graphical interface makes it easier to go from planning to code, especially when working in the field of telecommunications. SDL is using the state machine model. The programming is focused on states and transitions. Programs have states, which they will not leave until a signal has been received and this in turn will trigger the sending of other signals and so on. Systems described in SDL consist of many processes running simultaneously that communicate with each other via signals. Each process is described by an extended finite state machine. The state machines are labelled extended since variables and timers can also be defined in processes.

Systems Diagrams

You will use the symbols in Figure 1 in Systems diagrams.

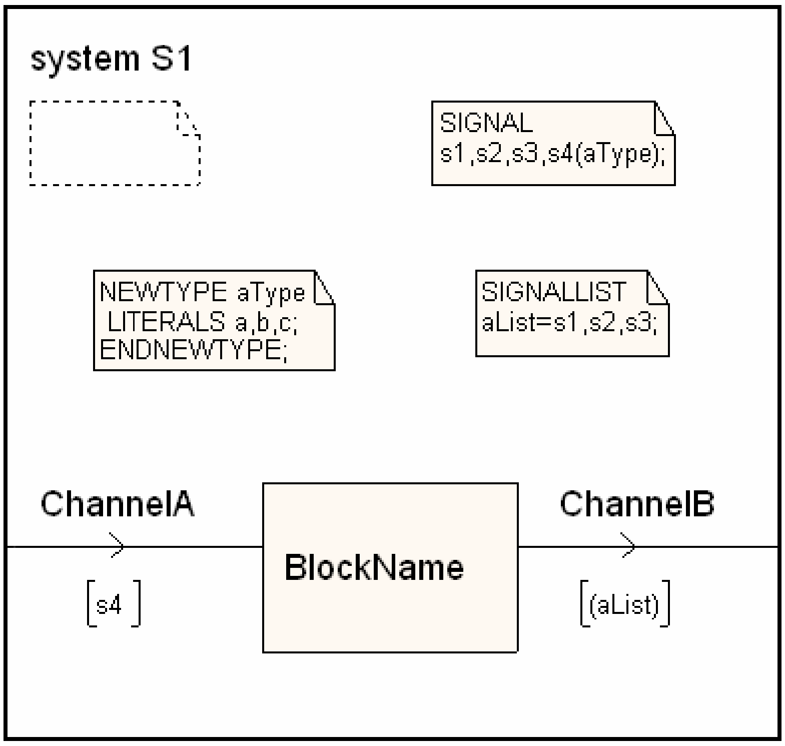

Figure 1

The system name is written after the keyword *System*. Text symbols are used for declarations. Blocks are connected to the environment or to other blocks with channels. All signals in the system are declared after the keyword *Signal*. If signals have parameters, their types are written within parenthesis.

Signal lists are used when you have many signals and do not want to clutter the channels with long lists of signals. Instead you can declare a signal list and use this list in the signal specification by the channel like this: [(aList)]. The parenthesis indicates a signal list. No signal parameters are included in the signal list declaration.

Newtype declarations are used to introduce new types in addition to the built in types, such as integer and boolean.

Block Diagrams

You will use block diagrams including the symbols in Figure 2.

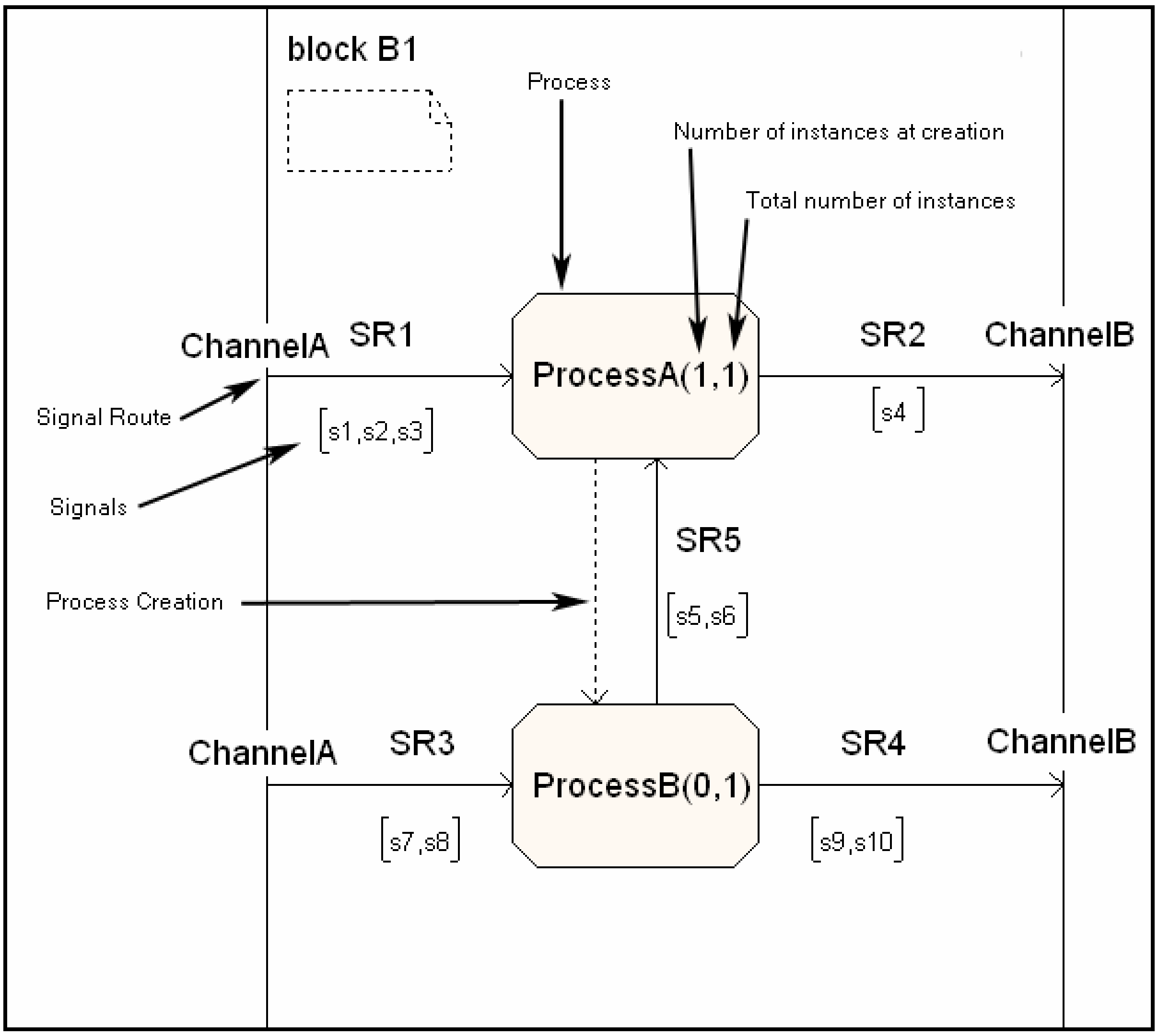

Figure 2

Blocks contain processes. Processes are connected with signal routes. Signals that are sent on a signal route are written within brackets. If one process dynamically creates another process, this is indicated by a dotted line. The numbers within parenthesis inside the process symbol, indicate how many instances exist at system startup and the maximum number of instances. The Channel-declaration connects signal routes on the block level to channels on the system level.

Process Diagrams

You will use process diagrams including the symbols in Figure 3.

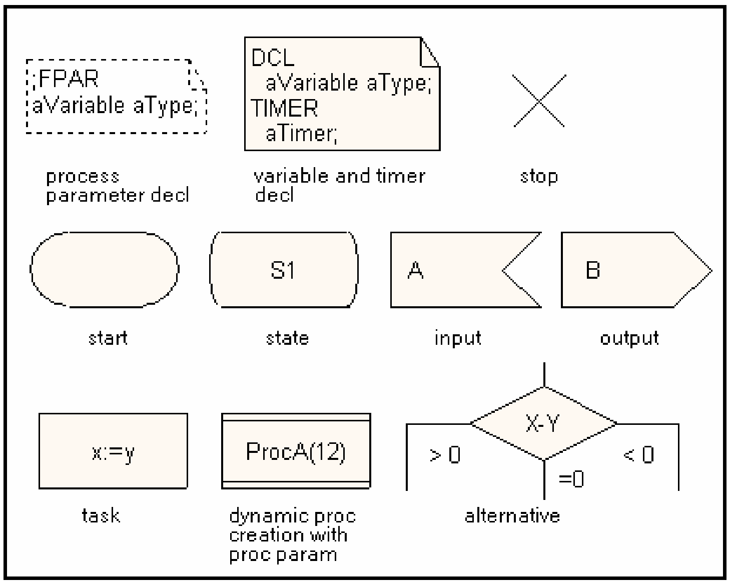

Figure 3

For more information, see “Introduction to SDL”.

Message Sequence Charts (MSC)

MSC diagrams are used to describe the dynamic behavior of a communicating process. We will use MSC to animate our SDL description. You will exercise the basic constructs of MSC; process instance, message and timer.

As MSC describes execution instances of SDL systems, only one specific scenario of message sequencing is described by one MSC. Many MSC diagrams are needed to reflect the different executions possible for a system. Often MSC diagrams are constructed for both a normal case of execution and some exceptional cases. Figure 4 shows the elements of a simple MSC.

Each process and the environment have their own time-axis, with time progressing downwards. Signals between processes (with parameter values if present) are represented by labeled arrows from the sending process to the receiving process. Dynamic creation of processes is represented by dashed arrows and termination is represented by crosses.

In our example we see that the signal Signal1 is sent from the environment to the process instance ProcessA, which results in a process creation with the parameter value of 12. There can be several instances of the same process type.

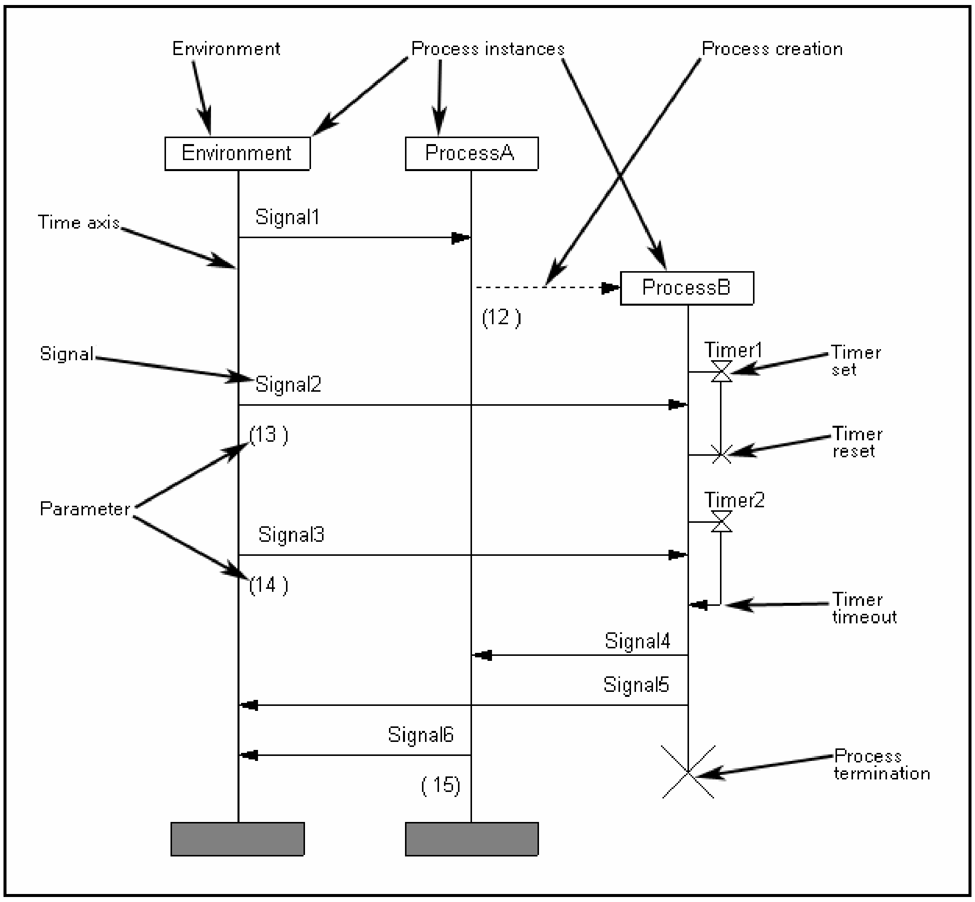

Figure 4

After the instance of ProcessB has been created, a timer Timer1 is set. Timers are used to keep track of time in processes. When a timer reaches timeout, a signal is sent to the process itself. This is shown in the picture with the timer Timer2. A timer can also be reset to prevent it from reaching timeout. This is shown in Figure 4 by the timer Timer1.

Subsequently, the two signals Signal2 and Signal3 are sent with the parameter values 13 and 14, which result in the Signal4 and the Signal5 signals, sent to ProcessA and Environment respectively. After this the process instance ProcessB is terminated and a signal Signal6 with parameter value 15 is sent to the environment.

The order of signals in an MSC is only partial. No global time axis is assumed. The only implied order is:

- A signal is sent before it is received,
- On each time axis (viewed separately) the time is running from top to bottom,
- The distances between messages are not proportional to elapsed time.

In the Figure it is not determined whether, for example, the receiving of Signal4 happens before or after the receiving of Signal5. MSC diagrams can be used to verify the behavior of SDL systems. SDT 2.3 has automatic support for this. Also, MSC diagrams can be generated by SDT 2.3 when systems are simulated.

The Can Machine

The system described is a simple soda-machine, where a customer can buy cans with three different types of expensive luxury drinks: cola, water and beer. Water costs \$5, cola costs \$10 and beer costs \$15. Figure 5 shows the system.

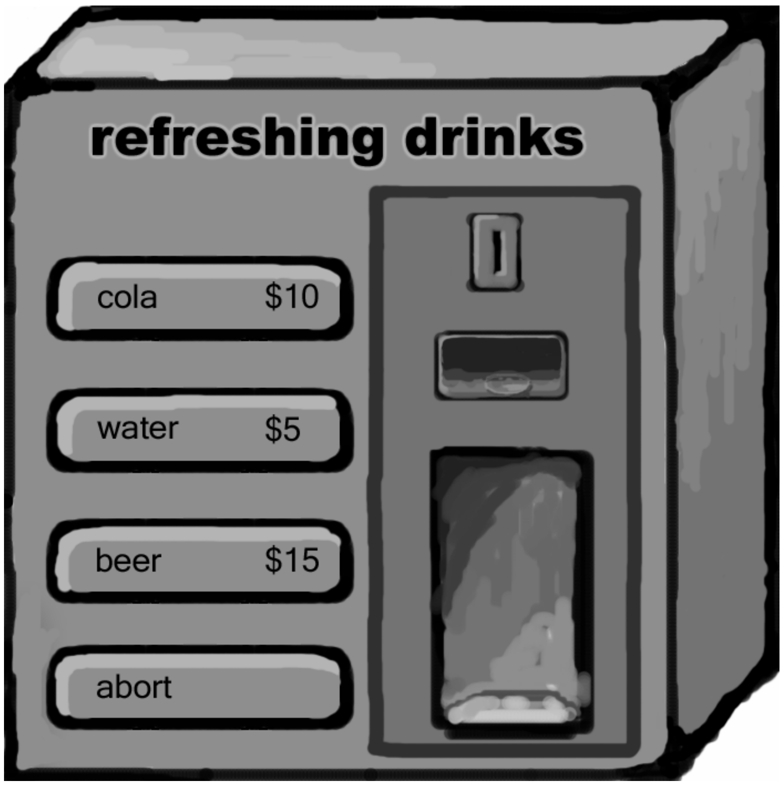

Figure 5

A normal operation of the machine is described by the following sample scenario (note that the drink is selected *before* payment):

1. The user presses the cola button.
2. The user inserts a \$5 coin.
3. The user inserts a \$5 coin.
4. The machine releases a cola can.

We only concentrate on the software of the machine, and to make the system simple the following assumptions are made:

- The machine will never run out of cans of any kind.
- The machine will always have enough coins to give back as change.
- The hardware takes care of coin recognition.
- We assume that the user always starts by pressing one of the drinks buttons.

The (software) system receives the following signals from its environment:

- **Cola –** received when the cola-button is pressed.
- **Water –** received when the water-button is pressed.
- **Beer –** received when the beer-button is pressed.
- **Abort –** received when the abort-button is pressed.
- **Coin(x) –** received when a coin with the value x (of type Natural) has been inserted.

The system sends the following signals to its environment:

- **ReleaseCan (aCan) –** a can of type **CanType** is given to the user. **CanType** can be one of the following literals: **colaCan, waterCan** or **beerCan**.
- **GiveCoins(x) –** coins amounting to x (of type **Natural**) are returned.
- **KeepCoins –** the coins inserted in the input slot are dropped in the money reserve.
- **ReleaseCoins –** the coins inserted in the input slot are returned back to the user.

We have already decided on the design of the system by dividing it into two processes; **CanHandler** and **MoneyHandler**. We here give a description of how these processes should work.

CanHandler is responsible for monitoring the drinks buttons and releasing the cans to the customer. **MoneyHandler** is responsible for keeping track of the coins inserted and detecting if the amount is correct for the desired drink. **MoneyHandler** also checks that the user does not wait too long before or between coin insertions. This is accomplished by a timer that is set to 10 time units. **MoneyHandler** also takes care of the situation where payment is interrupted (the user presses the abort button).

When the user presses a button for a specific type of drink, the **CanHandler** determines the price and dynamically creates a process instance of type **MoneyHandler** that keeps track of the coins inserted by the user, and reports back to **CanHandler** when the drink is paid and then finally terminates.

From **MoneyHandler** the following signals can be sent back to **CanHandler**:

- **Paid –** if enough money is paid with not too much delay.
- **Aborted –** if the user pressed the abort-button or waited too long.

If **CanHandler** gets the signal **Paid** from **MoneyHandler,** an appropriate can is released by sending the signal **ReleaseCan(item)** to the environment. If the paid amount is greater than the price, **MoneyHandler** sends the signal **GiveCoins(x)** to the environment, where **x** is the amount of exchange to be given back. If the payment procedure is interrupted for some reason, indicated by the signal **Aborted**, the **CanHandler** should not release any cans, but **MoneyHandler** should send **ReleaseCoins** to the environment to give back the money that might have been inserted into the input slot. Note that we do not bother about the special cases where the user starts a transaction with coin insertion or pressing the abort button.

Exercises for CanMachine

1. Make a system diagram for the system called **CanSystem** with the following components: Hint: Choose Add New in the Edit menu and chose SDL: system in the next window.

   1.1. One block called **CanMachine.**
   1.2. Two channels called **Ch1** (from environment to **CanMachine**) and **Ch2** (from **CanMachine** to environment). Use a signal-list named **FromEnv** for **Ch1** and a signal-list named **ToEnv** for **Ch2.**
   1.3. A **SIGNAL** declaration in a separate text-symbol with all the signals in the system. Give the types of parameters for the three signals that have parameters.
   1.4. Two **SIGNALLIST** declarations in one separate text-symbol.
   1.5. A **NEWTYPE** declaration in a separate text-symbol for the type **CanType.**

2. Make a block diagram for **CanMachine** with the following ingredients: Hint: Enter the system by double-clicking on it. Chose process interaction page in the next window.

   2.1. Two processes called **CanHandler** and **MoneyHandler** respectively.
   2.2. Four signal routes called **SR1** to **SR4**, two per process connecting it to and from the environment.
   2.3. One signal route called **SR5** from **MoneyHandler** to **CanHandler.**
   2.4. Figure out what signals should travel what signal routes, and put the correct signals in brackets by the routes.
   2.5. A process creation arrow between **CanHandler** and **MoneyHandler.**
   2.6. Figure out how many process instances there should be of each process type at start-up and maximum, and declare this inside the process symbols.

3. Make a Message Sequence Chart with three time axis: **environment**, **CanHandler** and (eventually) **MoneyHandler**. Describe the following events:
   3.1. The signal **cola** is sent to **CanHandler.**
   3.2. **MoneyHandler** is created by **CanHandler.**
   3.3. **MoneyHandler** starts a timer **Time_Out.**
   3.4. The signal **Coin** with parameter **5** is sent from environment to **MoneyHandler.**
   3.5. The timer **Time_Out** is reset.
   3.6. **MoneyHandler** starts a timer **Time_Out.**
   3.7. The signal **coin** with parameter **5** is sent from environment to **MoneyHandler.**
   3.8. The timer **Time_Out** is reset.
   3.9. The signal **Paid** is sent to **CanHandler** from **MoneyHandler.**
   3.10. The signal **KeepCoins** is sent to env. from **MoneyHandler.**
   3.11. **MoneyHandler** terminates.
   3.12. The signal **ReleaseCan** with parameter **ColaCan** is sent from **CanHandler** to environment.

4. Make two more MSC's that describe the following situations:
   4.1. Beer is ordered and two $10 coins are inserted.
   4.2. Water is ordered but no coins are inserted within the time limits.

5. Make a process diagram for **CanHandler** with the following:
   5.1. A text symbol with declaration of the two variables: **price** and **item**. **Price** is of type **Natural** and **item** is of type **CanType**. We will use **price** to give the price of each drink, and **item** to remember what type of drink is ordered.
   5.2. The process will have two states: **idle** and **wait.**
   5.3. The process starts in **idle** and then waits for the drink selection signals **Cola**, **Water** and **Beer**. After each of these signals, a task symbol should be used to give the variables **price** and **item** the correct values.
   5.4. Before we make a transition to the state **wait**, we create the process **MoneyHandler** with a process parameter that gives the amount needed for each drink.
   5.5. In the **wait** state we wait for **MoneyHandler** to send back two signals. Which?
   5.6. One of these signals will result in sending a signal to the environment. Which?
   5.7. The transitions from **wait** go back to **idle**.

6. Make a process diagram for **MoneyHandler**. Try to figure out for yourself how it should look like. We give the following hints:
   6.1. Use two variables: **amount** to keep track of how much is paid, and **value** to keep track of the value of each paid coin.
   6.2. You need a formal parameter declared in text symbol with a; FPAR declaration to get the price of the ordered drink at process creation.
   6.3. You need a **TIMER** declaration for the timer **Time_Out**.
   6.4. After the start symbol you need to initiate amount and set the timer.
   6.5. You only need one state (e.g. called **collect**) in which we can receive three signals. Which?
   6.6. For each coin signal, you need to keep track of its value and check if **amount** minus **price** is greater, less than or equal to zero. What should happen in each case?
   6.7. Don’t forget to **Reset Time_Out** where needed.

7. Answer the following questions:
   7.1. What will happen in your implementation if the user starts by pressing Abort?
   7.2. What will happen in your implementation if the user starts by pressing several drink-buttons in a row?
   7.3. What will happen in your implementation if the user starts by entering coins?