How to use `JavaLLFSMsTransformation` and its associated tools

Vladimir Estivill-Castro, Miguel Carrillo Barajas, David A. Rosenblueth

August 21, 2024

Contents

Chapter 1

Introduction

This document illustrates the use of the software

```
JavaLLFSMsTransformation
```

and its associated tools. The aim of this software is to enable the construction of executable models of behaviour (as logic-labelled finite-state machines) that can be transformed either into executable models (in C, MIPS, or LISP) or into formal models (in NuSMV or TLA+) for formal verification.

An short video (https://www.youtube.com/watch?v=RFpwlsb50ds) of the usage of the tools with the Microwave example of Section 3.3.

1.1 Running the public docker image `llfsms-examples`

You should have `docker` installed[1]. We recommend you run the image with a `results` folder mapped so you can move PDF files to your host.

```
rm -fr results
mkdir results
```

If you are running under **MacOs** (your host is a Mac) you need to run (and thus download with the tag **darwin**)

```
docker run -v $PWD/results:/opt/results -ti -rm vladestivillcastro/llfsms-examples:darwin /bin/bash
```

If you are running under **Ubuntu**[2] you need to run (and thus download with the tag **linux-gnu**)

```
docker run -v $PWD/results:/opt/results -ti -rm vladestivillcastro/llfsms-examples:linux-gnu /bin/bash
```

You will be root in the image and the software is copied to `opt`

```
opt
|__ Examples
|   |__ OutputOnlyPingPongTikiTaka.d
|   |__ RealTimeBounds.d
|   |   |__ Fischer.d
|   |   |__ FischerCopies.d
|   |   |__ Microwave.d
|   |   |__ TimedExpresion.d
|   |__ SchedulersForLLFSMs.d
|   |__ set_environment.sh
|__ README.md
|__ Tools
```

[1]www.docker.com

[2]It is common in Ubuntu to use `sudo` before each `docker` command.

```
    Cexecutable
    JavaLLFSMsTransformation
    MIPS
    lisp-scheduler
    mipsel-linux-musl-cross
```

As you run the container, you should find yourself in the folder with the examples.

```
cd /opt/Examples
```

Moreover, the `.bashrc` should have set the environment running

```
set_environment.sh
```

You are now ready to work with the examples. Move on to Chapter 2. **NOTE:** At this time, We are regularly updating this release, so if you have the image locally, it may be the case that you are not running the latest version. Check the last update in hub.docker.com/search?q=llfsms

1.2 Downloading the repository with the examples

The examples are available, currently in a private directory, at git@github.com:vladestivillcastro/LLFSMTransformationExamples.git.

If you are starting from a fresh installation of Ubuntu, you may need to install `git`.

```
sudo apt install git
```

The cloning from `github` may vary according to your credentials. However, a common way to download the repository is as follows.

```
git clone git@github.com:vladestivillcastro/LLFSMTransformationExamples.git
```

This will create a directory `LLFSMTransformationExamples`. The repository has a structure similar to this.

```
LLFSMTransformationExample
    Dockerfile
    setImage.sh
    runImage.sh
    Examples
        OutputOnlyPingPongTikiTaka.d
        RealTimeBounds.d
            Fischer.d
            FischerCopies.d
            Microwave.d
            TimedExpresion.d
        SchedulersForLLFSMs.d
        set_environment.sh
    README.md
    Tools
        Cexecutablelinux-gnu.tar
        MIPSlinux-gnu.tar
        JavaLLFSMsTransformation.tar
        lisp-scheduler.tar
        runLLFSMS.asm
```

The file `README.md` that comes with the examples complements these instructions to reproduce the examples.

The file `Dockerfile` is the file to build a docker image similar to the public release mentioned in Section 1.1. However, here we show that we can create a clean image. This is an alternative way of working with the examples. **Remember that the simplest way of working with the examples in a way agnostic of whether you are running MaCOS or Linux is to run inside a container**.

We recommend to build the image using the script `setImage.sh`. If you are running under **MacOs** (your host is a Mac), run the script as follows.

```
source ./setImage.h
```

But if you are running under **Ubuntu**, you probably need `sudo` to run `docker` and its commands. So you run the script to build the image as follows.

```
sudo ./setImage.h
```

Now you can run a container for this newly lcoally created image. If you are running under **MacOs**, run the scripta that starts the container as follows.

```
source ./runImage.h
```

But with **Ubuntu**, reacall you probably need `sudo` to run `docker`.

```
sudo ./runImage.h
```

You are in the container now. Navigate to the folder with the examples.

```
cd /opt/Examples
```

Set up the environment.

```
source set_environment.sh
```

You are now ready to work with the examples. Move on to Chapter 2.

1.2.1 Directory `Tools`

The directory `Tools` is a subdirectory of `LLFSMTransformationExample`. In the directory `Tools` you will find the tools in executable format. You should move to this directory and open the packages.

```
cd Tools
tar -xvf JavaLLFSMsTransformation.tar
tar -xvf Cexecutable.tar
tar -xvf lisp-scheduler.tar
```

This expands the file structure as follows.

```
LLFSMTransformationExample
 ├─Examples
 │  ├─OutputOnlyPingPongTikiTaka.d
 │  ├─RealTimeBounds.d
 │  │  ├─Fischer.d
 │  │  ├─FischerCopies.d
 │  │  ├─Microwave.d
 │  │  └─TimedExpresion.d
 │  ├─SchedulersForLLFSMs.d
 │  └─set_environment.sh
 ├─README.md
 └─Tools
    ├─Cexecutable.tar
    ├─Cexecutable
    │  ├─cobjects
    │  ├─cobjects-linux
    │  ├─includes
    │  └─MakefileForArrangement
    ├─JavaLLFSMsTransformation
    │  └─dist
    ├─JavaLLFSMsTransformation.tar
    ├─lisp-scheduler
    │  ├─MacOs
    │  └─linux
    └─MIPS
       └─runLLFSMS.asm
```

1.2.2 Setting your environment

Our software is written in Java, and we have tried to use the most agnostic version of `java`. For MacOS we are using version 1.8. For Ubuntu 20 and Ubuntu 22 we are using openjdk 11.0.18. The EMF (Eclipse Modelling Framework) classes we have in the repository are starting to give warnings, meaning that a higher `java` version may not work unless corresponding EMF classes are used instead.

We use several other tools, and you may need to install these. If you are using Ubuntu from a fresh installation, some of them are installed as follows.

```
sudo apt install gcc
```

This is the C-compiler, it is needed after the translation of a model to C, to execute the model. If you want to run the LISP examples you will need the following in Ubuntu.

```
sudo apt install clisp
```

Also, if you want to visualise the models with `dot`, you need to install `graphviz` in Ubuntu as well.

```
sudo apt install graphviz
```

Other requirements are tools (like the `TLC` model checker or the NuSMV model checker) that you will need to install by downloading that software and following the corresponding instructions. Some of them have prerequisites, like `TLC` requires

```
sudo apt install curl
```

in Ubuntu.

The directory `Examples` is a subdirectory of `LLFSMTransformationExample`. To help you setting the environment, in the directory `Examples` there is a script that supports setting the environment variables of the `bash` shell. This file has the name `set_environment.sh`. If you are still in the `Tools` directory, the commands are as follows.

```
cd ../Examples
source set_environment.sh
```

Inspect the output from this carefully. It would indicate if it finds the `Tools` directory, whether you are using MacOs or Linux (Ubuntu) and if it has found the necessary compilers, or model checkers and where to get them from so it can set environment variables to where these tools are.

Chapter 2

LLFSMs: Executable and verifiable models

2.1 Simple example: `OutputOnlyPingPongTikiTaka`

Each example should come with a file `test.sh` that is a script that illustrates the demonstrations we have prepared with a particular example.

```
 1  #
 2  #       (c) 2022 Vladimir Estivill-Castro
 3  #         This commands illustrate different targets of the Makefile
 4  #
 5  #
 6  make clean
 7  echo Producing DOT file to derive PDF
 8  make dot
 9  make brief-dot
10  make timer-dot
11  echo Producing LISP file to test running
12  make lisp
13  # How to run the LISP version need the path to the LISP LLFSM interpreter
14  #source `make name`.sh 2> junk
15  #
16  echo Producing NUSMV file to test properties
17  make smv
18  echo testing properties with NuSMV
19  make test-properties
20  echo Producing C program to compile with C-language
21  # How to compile in C needs path to plainCscheduler includes and compiled objects
22  make C
23  make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForModelToTextTransformation cfile=`make name`
24  #Running the compiled C file
25  #./executables/`make name`.exe
26  echo Producing TLA file to produce specification with  TLA notation
27  make tla
28  # How to produce the TLA in PDF depends on where TLS's command tools are installed
29  #$TLA_COMMANDS_INSTALLATION/tlatex    -latexCommand  pdflatex -latexOutputExt pdf `make name`.tla
30  # To test the properties with tlc, see the Makefile, now the translator produces config files for TLF
31  # If you want to use the Toolbox, create a new specification (spec)
32  # use the tla file to copy-paste the spec
33  make test-tla-properties
34  echo Producing asm file to simulate with MARS http://www.cs.missouristate.edu/MARS/
35  make mips
36  # To run the assembler version
37  # RUN MARS from the command line with a recommended limit of MIPS instructions
38  #java -jar $MARS_HOME/Mars4_5.jar 5000 `make name`.asm
```

Figure 2.1: A fragment of the `test.sh` shell script.

These demonstrations are usually coded into the `Makefile` as specific targets. If we start with the simplest example `OutputOnlyPingPongTikiTaka` we need to enter the directory of the example.

```
cd OutputOnlyPingPongTikiTaka.d
```

Note that this directory contains a file `OutputOnlyPingPongTikiTaka.xmi`. The `xmi` format is used by the Enterprise Modelling Framework. We created it with our graphical editor for our models.

Here, we can inspect the file `test.sh`

```
more test.sh
```

and see that one of the earliest things to do (Line 8 in Figure 2.1) is to produce a PDF file of the model.

```
make dot
```

Issuing this target with `make` will invoke the translator which uses as input the model (the model is the input file `OutputOnlyPingPongTikiTaka.xmi`) and produces a `dot` file `OutputOnlyPingPongTikiTaka-regular.dot`. Also the target calls `dot` to generate the corresponding PDF. So now you can use your favourite tools to open a PDF file and open `OutputOnlyPingPongTikiTaka-regular.pdf` and see an arrangement of two logic-level finite-state machines, one called `PingPong` and the other called `TikiTaka`.

If you are using the docker image, there is no GUI environment, so you shall move the PDF files from the container to your host.

```
mv OutputOnlyPingPongTikiTaka-regular.pdf /opt/results/
```

Now you can use any PDF viewer in your host to inspect the PDF file.

2.1.1 Translating to `LISP`

The `test.sh` file also illustrates how to generate the LISP translation of the same model.

```
make lisp
```

This produces a script file `OutputOnlyPingPongTikiTaka.sh`. Now, if you want to run the model using `clisp` we see how this illustrated in the `test.sh` file.

```
source `make name`.sh 2> junk
```

Note that the `Makefile` has a target `name`, so that `make name` saves writing out the name of the particular example. That is `make name` is uniform across all examples (and almost all `Makefile`s and `test.sh`s files are equal for all examples.

Be prepared to send the stop signal to kill this process as the machines run continuously, alternating between their two states and the arrangement providing a turn to each machine in round robin fashion.

2.1.2 Model checking with `NuSMV`

The above section shows the model is executable. Now we are going to verify properties in LTL and CTL using `NuSMV`. You would need to install `NuSMV` and make sure that `NuSMV` can be found. Test this with

```
which NuSMV
```

If this does not work, maybe you need to edit your `.bashrc` and add the path to where the binary of `NuSMV` is. If `NuSMV` can be found, we translate the model and also run `NuSMV` with option `-ctt` all within the same `make` target.

```
make smv
```

You should get output that indicates no deadlock exists in this model. To check for properties, we have a target as well

```
make test-properties
```

You would see that the model satisfies all properties. Some are properties in the temporal logic LTL and appear in the file `ltl-properties.nusmv` and some are CTL properties placed in the corresponding file `properties.nusmv`. Inspect these files to see comments to these properties so you can probably review their explanations.

```
more ltl-properties.nusmv
more properties.nusmv
```

2.1.3 Running with C

The previous execution with `LISP` is probably interpreted. But logic-labelled finite-state machines can be compiled. Or at least the scheduler and running system. We also use this illustration to show that the executable model is agnostic to the target implementation. Thus, to translate the same model to C, the target is as follows.

```
make C
```

Again, this command is illustrated in the `test.sh` file. Now to compile against the object files of the meta-model, we invoke the corresponding `Makefile` for invoking `gcc`.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

This uses the environment variables when we ran `set_environment.sh`. Inside a container we link against binaries that use proper system calls for the architecture of the host. That is why the images come in two versions. This should be now transparent to you (whether we link against Ubuntu-linux binaries or MacOS binaries). You will see the invocation of the `gcc` compiler and the executable is placed in a subdirectory `executables` So to execute the compiled C translation we now do

```
./executables/`make name`.exe
```

(this line is also in `test.sh` so you can copy and paste from it).

2.1.4 Using a second model checker: `TLC`

`TLC` is another model checker part of the series of tools of `TLA+`, which uses a subset of LTL called TLA (temporal logic of actions) to define the model and its properties. So you need `TLC` (which uses `java` and `curl`). Again, `test.sh` shows what the target is to use this model checker.

```
make tla
```

To run the `TLC` against the translated model now in `TLA+` we issue the following.

```
make test-tla-properties
```

You can inspect the properties (many are the translation to `TLA+` from how they were LTL properties in `SMV` for `NuSMV`).

```
more properties.tla
```

Here, properties must be written so they occupy a single line for our script to properly place them in the configuration files for `TLC`. So the comments comply with the requirement of one line per property.

2.1.5 Generation and verification of sanity properties

The command

```
make smv
```

not only generates the NuSMV model but also uses the model checker with the flag `-ctt` which performs some verifications and validations over the corresponding Kripke structure (the NuSMV manuals says *enables checking for the totality of the transition relation*).

However, we argue that one advantage of the automatic generation of models for model-checkers is that we also can generate the properties to verify automatically, thus, minimising the chances we forget something to check. For example, we may want to make sure that every state is each of our LLFSMs in an arrangement is eventually reachable. Otherwise, why would we have included a state that is never used. Thus, for this example we can run the command

```
make test-sanity
```

and an LTL formula for all states in the arrangement is generated:

```
F PingPong.At_START_PING_PONG
F PingPong.At_PING
F PingPong.At_PONG
F TikiTaka.At_START_TIKI_TAKA
F TikiTaka.At_TIKI
F TikiTaka.At_TAKA
```

and also run against the NuSMV model checker. This sanity check can also be performed with `TLC` since this formulas are within the LTL subset of `TLA+`.

```
test-tla-sanity
```

Similarly, for every state S that is source of some transition, we test that if we reach S, we also eventually leave S. Therefore,

```
make test-sanity
```

also generates during translation and then verifies the following properties.

```
G (PingPong.At_START_PING_PONG -> F !PingPong.At_START_PING_PONG)
G (PingPong.At_PING -> F !PingPong.At_PING)
G (PingPong.At_PONG -> F !PingPong.At_PONG)
G (TikiTaka.At_START_TIKI_TAKA -> F !TikiTaka.At_START_TIKI_TAKA)
G (TikiTaka.At_TIKI -> F !TikiTaka.At_TIKI)
G (TikiTaka.At_TAKA -> F !TikiTaka.At_TAKA)
```

All these properties are true of this arrangement. However, although they may be intuitively always true of all arrangements, this is not the case for open models. Recall that open models will have external variables that represent input from the environment (say a user, or a sensor). In those cases, the path to a state S may be enables, but then the environment doe snot provide the input to exit such a state,

Another series of automatically generated properties are those regrading the scheduling of turns to each LLFSM in the arrangement. But, as we will see later, also these properties should be tailored to the scheduler. For example, only in the sequential scheduler the turns follow the round robin patterns that

```
turn' = (turn+1) % length of arrangement
```

2.1.6 Translating to assembly and running in MIPS simulator

The last demonstration produces a translation into the assembly language of MIPS. Inside a container, the code is cross-compiled to binary code for MIPS little endian of 32 bits, and executed by the `qemu` emulator. On the source version, the code can be executed with the `MARS` simulator (MARS is also written in `java`). For this option, the path to where you installed `MARS` is necessary and you need to adjust the `set_environment.sh` accordingly.

In the docker container, launching the translation of the model to assembly is the following target.

```
make mips_docker_qemu
```

The file `test.sh` shows how to cross-compile the model with `gcc` from `musl`.

```
make -f $MUSL_MIPS_SCHEDULER_INSTALLATION/MakefileMUSL cfile=`make name`
```

Finally, we execute the MIPS binary with the corresponding emulator.

```
qemu-mipsel ./executables/OutputOnlyPingPongTikiTaka.exe
```

The option with source is built with

```
make mips
```

Once more `test.sh` shows how to execute the model with the `MARS` simulator.

```
java -jar $MARS_HOME/Mars4_5.jar 5000 `make name`.asm
```

In this case, we have put a limit to the execution so there is no need to kill the process.

2.2 The Fischer's mutual exclusion example — two threads

This example illustrates an algorithm for mutual exclusion. The first example has only two threads and is in the directory

```
RealTimeBounds.d/Fischer.d
```

This example provides a `README.TXT` file for two references that describe Fischer's algorithm. The production of visualisations using `dot` has been tested here for MacOS and Ubuntu. The different versions are as follows.

```
make dot
make brief-dot
make timer-dot
```

The brief version does not draw `OnExit` and `Internal` sections, which is convenient since all the examples we have do not have these sections. The *timer* version draws timers in case the model has temporal transitions. This option is best illustrated with the *Microwave* example (see Section 3.3).

2.2.1 The LISP version

The target for `lisp` is issued as follows

```
make lisp
```

This command produces the LISP version. You can use the illustration in the file `test.sh` again to run the LISP LLFSM scheduler.

```
source `make name`.sh
```

In this example, we have four external variables to hold each of the two threads at two parts of the behaviour. That is, the user (playing the role of the environment) can control how long does each thread stays in its critical section. Also, the user can control how long does each thread stays in its non critical section. The Boolean variables `T1_CRITICAL_SECTION_DELAY`, `T1_NON_CRITICAL_SECTION_DELAY`, `T2_CRITICAL_SECTION_DELAY`, and `T2_NON_CRITICAL_SECTION_DELAY` regulate such delays. They are external Boolean variables. When the program is run with the LISP scheduler, for every ringlet, the user is asked to provide Boolean values for these four external variables. A quick view of the behaviour consists in setting each variable repeatedly to 1 in every ringlet. In that case both threads exit critical and non-critical sections as soon as they arrive. When formal verification is performed on this example, obviously all possible sequences of providing values to these four variables are considered. Verifying mutual exclusion means, the user can not find ways of providing values (delaying or shortening the stay on the critical or non-critical sections) so that both threads are at some point in their critical section. That is, such verification means that no matter how short or long is the performance of the single CPU while at each of the sections, it is never possible that both threads enter their critical sections.

The LISP scheduler does not obey the LLFSMs semantics that after the evaluation of a transition to *true*, the `OnEntry` of the next section is atomically executed. That is, LLFSMs have a semantics where the evaluation of the transition and the execution of the code in the target state are similar to the *Compare and Swap* primitive for mutual exclusion (See: Herlihy, Maurice [January 1991]. "Wait-free synchronization", ACM Trans. Program. Lang. Syst. 13 (1): 124–149.) That is, when the expression in the transition evaluate to true, the body of the target state is executed atomically. If it evaluates to false, nothing happens.

Since the current LISP scheduler does not obey this atomicity semantics, it is possible by answering always 1 repeatedly in every ringlet that both threads reach their respective critical sections simultaneously. This illustrates critically the required implementation of the atomicity semantics of LLFSMs. A discussion of the possible semantic variants that emerge because of using sections inside states or because of when and how often sensor variables are read from the environment see Section 3.4. We leave for further work re-writing the LISP scheduler so instead of a very old semantics where LLFSMs had three sections in a state (OnEntry, Internal, OnExit), it only has the atomicity to OnEntry. Note that the old semantics of running a ringlet had atomicity when a transition fired and the OnExit of the source state.

2.2.2 Using `NuSMV` and different schedulers

The translation of the LLFSM model for Fischer's algorithm with the standard semantics (of a round-robin scheduler and atomicity from evaluating the transition and in the case it is true following it up with the evaluation of the `OnEntry` of the target state) is achieved with the following target.

```
make smv-sequential
```

The mutual exclusion properties are then evaluated with the following target.

```
make test-properties-sequential
```

This verifies mutual exclusion properties in LTL in the file `ltl-properties.nusmv`. The file for CTL properties is `properties.nusmv` and at the moment we have not placed a property in there.

Multi-threaded Fischer's algorithm

To test the same properties above that not only verify mutual exclusion but also can ensure some form of progress we can use what we name a non-deterministic scheduler. That is, the CPU can be awarded to either thread at any point in time. The NuSMV model is constructed as follows.

```
make smv-non-deterministic
```

You now would see the resulting file as `FischerDelaysByBooleanSensors-non-deterministic.smv`. To check all the properties in the file `ltl-properties.nusmv` we use the following target.

```
make test-properties-non-deterministic
```

The non-deterministic scheduler places a bound on the number of consecutive turns one of the threads (one of the LLFSMs) can receive the CPU. With only two threads this guarantees that no thread starves. However, with 3 threads, one thread can let to starve while the other two share all the turns, and even if one of those reaches the maximum of consecutive turns, it can pass it on to its partner thread, leaving one always starving.

There is a third scheduler we have named *fair*. This scheduler can be used in this example with the target

```
make smv-fair
```

All the properties can now be verified with `NuSMV` with the following target

```
make test-properties-fair
```

However, we show that with an unrestricted scheduler that allows one thread to monopolize the CPU, there is no progress. Although the property, as expressed by Lamport in his paper, is true, a property of effectively at least one thread entering and departing their critical section and then their non-critical section fails. The unrestricted scheduler can provide CPU turns to one thread that stays indefinitely in its non-critical section. To illustrate this we can build the model with an unrestricted scheduler.

```
make smv-unrestricted
```

and then verify the same properties with NuSMV.

```
make test-properties-unrestricted
```

and observe the last property is now false.

2.2.3 Using `TLA+` specifications

We now show the translation to `TLA+` specifications of this example, and the verification of properties with `tlc`. The translation is achieved with the following target.

```
make tla-sequential
```

This is a translation to a sequential scheduler. Threads receive a turn in round-robin fashion (see Equation 2.1) below. Now, to test properties using the `tlc` model check we use the following target

```
make test-tla-properties-sequential
```

In the output we should see the line

```
Model checking completed.  No error has been found.
```

Besides all the invariants generated by the translation, the verification includes the properties in the file `properties.tla`. These properties are a subset of the properties listed in `ltl-properties.nusmv` since the logic of `TLA+` does not include some operations of LTL (such as `X` i.e. *next*, and `U` i.e. *until*).

The file `properties.tla` does not have comments because it needs to be integrated with configuration files. Also, properties are restricted to be written in only one line.

We can also use `tlc` to verify properties with a different scheduler.

```
make test-tla-properties-unrestricted
```

Here again, the unrestricted scheduler cannot guarantee the last property which is a strong sense of progress, since it can allocate all the turns to a thread that is waiting for another one.

2.2.4 Running the model

The model can be executed by translating to C (at the moment only under the sequential schedule semantics) using the following target.

```
make C
```

Then, the programs ask for one of the four variables that are external variables (and can hold a thread in its critical and in its non-critical section). The C executable only ask for an external variable when it needs it and not in every ringlet. Refer to the `test.sh` on how to compile the resulting translation and how to run it. In particular, to compile the resulting `C` code, the `make` target is as follows.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

The, you can run it as follows

```
./executables/FischerDelaysByBooleanSensors.exe
```

Naturally, this execution is with the round-robin (sequential scheduler) of the semantics of an arrangements of LLFSMs.

Running with MIPS in this example is a before. To translate the model to MIPS, we use

```
make mips_docker_qemu
```

Next, we need to compile the MIPS assembly into MIPS executable

```
make -f $MUSL_MIPS_SCHEDULER_INSTALLATION/MakefileMUSL cfile=`make name`
```

To run the compiled MIPS code we use the emulator `qemu`.

```
qemu-mipsel ./executables/`make name`.exe
```

Here we can delay each of the two threads as much as we want by answering 0 or 1 (as *false* or *true*) to whether they leave their critical section or whether they leave their non-critical section (whatever is the case). A 0 (zero) keeps the thread in the corresponding section, while a 1 indicates the thread has completed the section and moves on. Abort the program to exit the emulation. Naturally, because of the formal verification, we are confident you will not be ale to place both threads as executing their critical section.

2.3 Fischer's Algorithm with more than two threads

The directory `FischerCopies.d` has a version of the Fischer's algorithm that enables to select the number of threads. For instance, the following target

```
make dot THREADS=6
```

generates a visualisation the PDF (via a `dot` version) of the model where there is only one instance drawn of a thread but where six copies of these thread are in the arrangement. The number of copies of a LLFSM in an arrangement is indicated by the appearance of `instacnes (6)` in the information about one drawing of the LLFSM.

2.3.1 Verifying mutual exclusion with `NuSMV`

To verify using NuSMV mutual exclusion we can construct the `NuSMV` model.

```
make smv-sequential
```

or

```
make smv-sequential THREADS=6
```

if we want a different number of threads from the default in the `Makefile`. We use `sequential` to indicate that the scheduler is the round-robin scheduler that provides a turn to each thread sequentially with

$$\texttt{turn} \leftarrow (\texttt{turn} + 1) \bmod (\sharp \text{ of threads}) \tag{2.1}$$

This NuSMV model can be simulated with NuSMV and you can elect the sensor variable that determines which thread gets the CPU as follows.

```
make smv-unrestricted THREADS=3
NuSMV -int FischerCopies-unrestricted.smv
go
pick_state -r
simulate -i -k 100 -c "(Sensor.Non_Critical_Section_Delay = TRUE)
  & (Sensor.Critical_Section_Delay = TRUE)
```

From there on, you interactively chose to allocate the turn to one of the three threads.

To test the mutual exclusion property there is a version where the file `ltl-properties.nusmv` is constructed as part of the build and the translation.

```
make test-properties-sequential THREADS=6
```

However, there is also a brief version. This version uses index variables in the LTL of `NuSMV`.

```
make test-properties-succinct-sequential THREADS=6
```

However, the interesting case is when the CPU is awarded arbitrarily to the any thread without any restriction.

```
make smv-unrestricted THREADS=6
```

Now, the mutual exclusion property is showing that property at its full power.

```
make test-properties-unrestricted THREADS=6
```

2.3.2 Verifying mutual exclusion with `tlc`

The target to verify the same mutual exclusion property with `tlc` and the sequential scheduler (Equation 2.1) is as follows.

```
make test-tla-properties-sequential THREADS=7
```

It is also possible to verify with `tlc` mutual exclusion for the unrestricted scheduler. **Warning**, depending on your computer this may take several minutes.

```
make test-tla-properties-unrestricted THREADS=7
```

ToDo: Show mutual exclusion for the other two schedulers (fair and bounded) and then show lack of progress with unrestricted scheduler with NuSMV and TLC.

ToDo: Show execution for C and for MIPS.

2.4 The schedulers as LLFSMs and their properties

The behaviour of the different schedulers (and therefore their properties) can be also analysed using the modelling with LLFSMs, and the corresponding translation tools. The examples of the schedulers are in the directory `SchedulersForLLFSMs.d`.

2.4.1 Unrestricted scheduler

Unrestricted scheduler

The first scheduler we will illustrate is the most flexible one where at any point, any thread can be awarded the CPU. The model for this scheduler is in `SchedulersForLLFSMs.d/UnrestrictedScheduler`. To use `dot` to visualise the scheduler as a state machine:

```
make dot
```

If you open the PDF file you should get the following image.

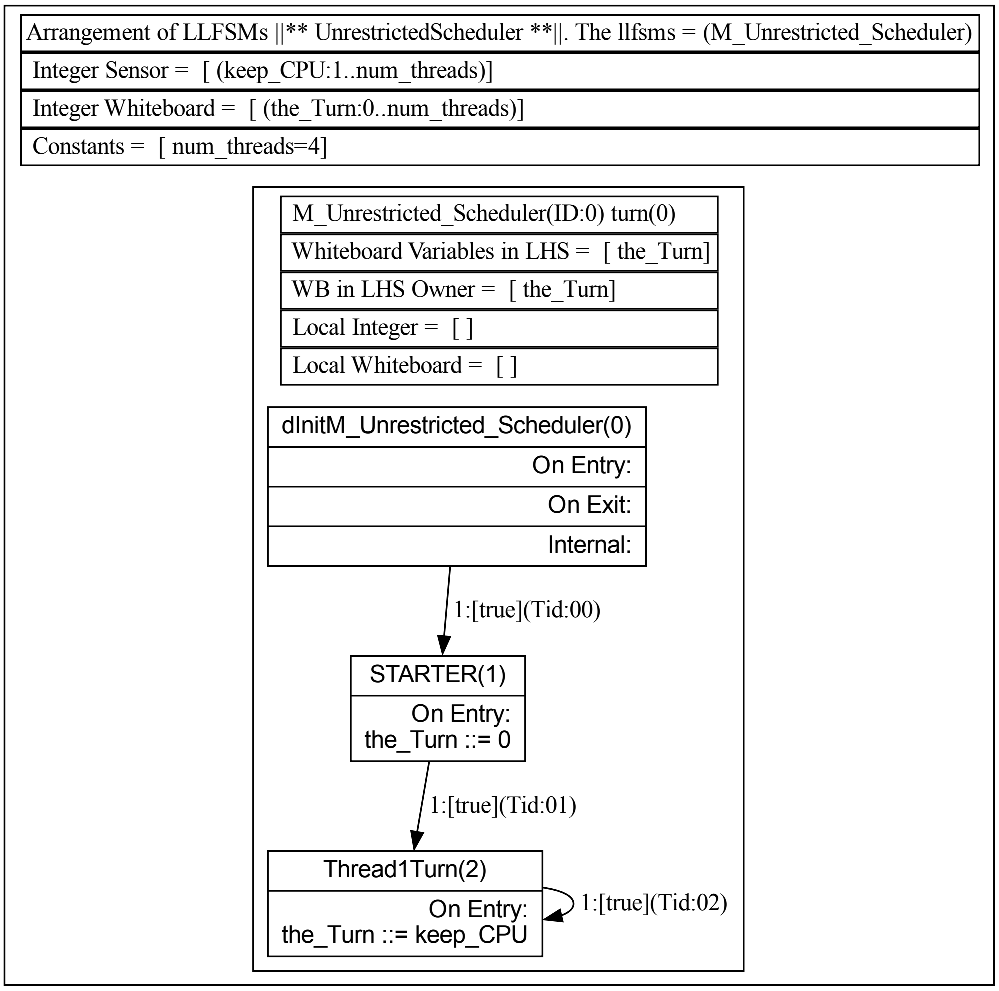

Arrangement of LLFSMs ||** UnrestrictedScheduler **||. The llfsms = (M_Unrestricted_Scheduler)
Integer Sensor = [(keep_CPU:1..num_threads)]
Integer Whiteboard = [(the_Turn:0..num_threads)]
Constants = [num_threads=4]
M_Unrestricted_Scheduler(ID:0) turn(0)
Whiteboard Variables in LHS = [the_Turn]
WB in LHS Owner = [the_Turn]
Local Integer = [ ]
Local Whiteboard = [ ]
dInitM_Unrestricted_Scheduler(0)
On Entry:
On Exit:
Internal:
1:[true](Tid:00)
STARTER(1)
On Entry:
the_Turn ::= 0
1:[true](Tid:01)
Thread1Turn(2)
On Entry:
the_Turn ::= keep_CPU
1:[true](Tid:02)

Figure 2.2: The unrestricted scheduler.

Running the unrestricted scheduler

Like all arrangements of LLFSMs, the model of the unrestricted scheduler is executable, and we can produce the LISP model.

```
make lisp
```

And then run it

```
source `make name`.sh 2> junk
```

Running it with LISP shows another aspect of the old semantics where states included sections (see Section 3.4). In that old semantics, a transition that fires only causes the execution of the `OnEntry` section of the target state if the source is a **different** state. So when running this with LISP, the variable `the_Turn` does not get updated although the sensor variable `keep_CPU` is receiving a different value. Under the LISP execution, only when the LLFSM moves from the state `STARTER` to `Thread1Turn` that the variable `the_Turn` is updated. We prefer the more uniform semantics that a transition that fires **always** causes the `OnEntry` section code to execute, since it applies across LLFSMs where states have sections as to LLFSMs where states have no sections. Moreover, it is visually more intuitive. Figure 2.2 illustrates this point: the transition `Tid:02` in the LISP semantics, although targets a state with an assignment, when it fires, it never causes the execution of the assignment!

We can also translate this model to `C` and run it with the proper semantics of LLFSMs. The target

```
make C
```

will perform the translation. In the file `test.sh` we see the command to compile.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

Now, to execute the C compiled file we issue the following command.

```
./executables/UnrestrictedScheduler.exe -v
```

The `-v` option run the executable model in verbose mode, and thus shows the value of whiteboard variables after each schedule completion. The value supplied (the reading of the environment in a sensor variable) can be any integer in $\{1, \ldots, 4\}$ by the default set in the `Makefile` If a larger number of threads is desired, we need to translate with a parameter. For example, if we want to share the CPU among 7 threads:

```
make C THREADS=7
```

And then we compile as before.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

Thus, in this example and with the `-v` option, we see the effect of supplying a value in the sensor[1]. and how it changes the variable `the_Turn`. The execution shows that any thread can be the next thread at the next turn.

Properties of the unrestricted scheduler

The unrestricted scheduler has the property that

at any time, any thread in the future can have a turn.

To show this is true for thread 1 we use the following property in CTL.

```
AG (EF w_b.the_Turn = 1)
```

It can be verified with the following target

```
make test-properties
```

as is a property in the file `ctl-properties.nusmv`. You will notice that we can use variables in NuSMV to prove the property in general for all threads.

```
VAR
        -- i and arbitary thread
        i : 1..g_c.num_threads;
CTLSPEC
AG (EF w_b.the_Turn = i)
```

Also, in this scheduler we can have *starvation* as

with two or more thread, there is a path that does not never awards a turn to thread i, for all i.

We show this by a CTL property that NuSMV shows to be true.

[1]Sensor variables allow to model non-deterministic variables, or variables the system does not have control, such as environment variables. In this case we are modelling that the CPU is awarded arbitrarily.

```
CTLSPEC
EF (EG w_b.the_Turn = 1)
```

This says the in all paths to some state S_0, eventually after S_0 there is a later state S_L from where we should be in a state where `the_Turn = 1` and also globally afterwards.

In LTL we must show starvation by false properties where NuSMV responds that the property is false and shows a path where the computation never awards a turn to a different thread. The file `ltl-false-properties.nusmv` has the following LTL properties.

```
LTLSPEC
G ( F w_b.the_Turn=1)

LTLSPEC
G( w_b.the_Turn=2 -> F (!(w_b.the_Turn=2 ) ))
```

The first one says that, no matter what state, eventually thread 1 will have a turn (false since this scheduler can award the turn for ever to other threads). The second one says that, no matter what state where the turn is given to thread 2, eventually it will be taken away from thread 2 (false because the turn can be awarded to thread 2 for ever). To run the verification of this properties and see in log files the counter-example generate by NuSMV use the following target.

```
make test-false-ltl-properties
```

2.4.2 Restricting repetition of turns

We now present a scheduler that limits the number of consecutive turns awarded to one particular thread. The diagram in Figure 2.3 presents a generic form of the scheduler. The variable `keep_CPU` is a sensor variable used to represent which thread (is requesting) the CPU. The variable `keep_CPU` is not in the sphere of control of the scheduler. The scheduler does not handle queues of requests. These schedulers are to manage an arrangement of LLFSMs as a multi-threaded system or a distributed system simulation (assigning which is the next machine to advance). They do not reserve turns to a thread because it has been requested in the sensor variable more. Simply, they model the non-determinism of the next turn when the LLFSMs in the arrangement are allocated to different CPUs.

The variable `the_Turn` is the thread to which the scheduler assigns the CPU. Thus, every time the scheduler assigns a value of a thread to `the_Turn`, that thread is being awarded the CPU. The strategy of this scheduler is concentrated in the transition from the state `starter's turn` to the state `thread` *i*`'s turn`. Namely, if thread i demands the CPU and the scheduler has not awarded this thread a maximum number of consecutive turns, then thread i is awarded the CPU. However, if thread i demands the CPU but this thread has had a maximum of consecutive turns, then we find some other thread j that has not reached the maximum number of consecutive turns, and j is awarded the CPU. Along the presentation of this example, and its formal verification, we will establish that, for example, such a j always exists if the number of threads is 2 or more.

We take this opportunity to highlight something about the semantics of LLFSMs. When Boolean expressions labelling a transitions are a disjunction, they are evaluated in short circuit mode (also called minimal evaluation or McCarthy evaluation). That is, and expression such as $A \vee B$ is determined true once A is found to be true and B is not evaluated. Also, since transitions out of the same source state are in a sequence and thus evaluated in the prescribed order of the sequence, disjunctions are equivalent to a set of transitions with common source state and target state but each labelled by one of the disjunction terms. Figure 2.4 shows the equivalent LLFSM to Figure 2.3 where the disjunction is replaced by two transitions. Figure 2.3 and Figure 2.4 are using a Booleans expression with an exists operator. This use of $\exists$ is indeed shorthand notation for a loop that iterates over the range of the variable local i, and sets

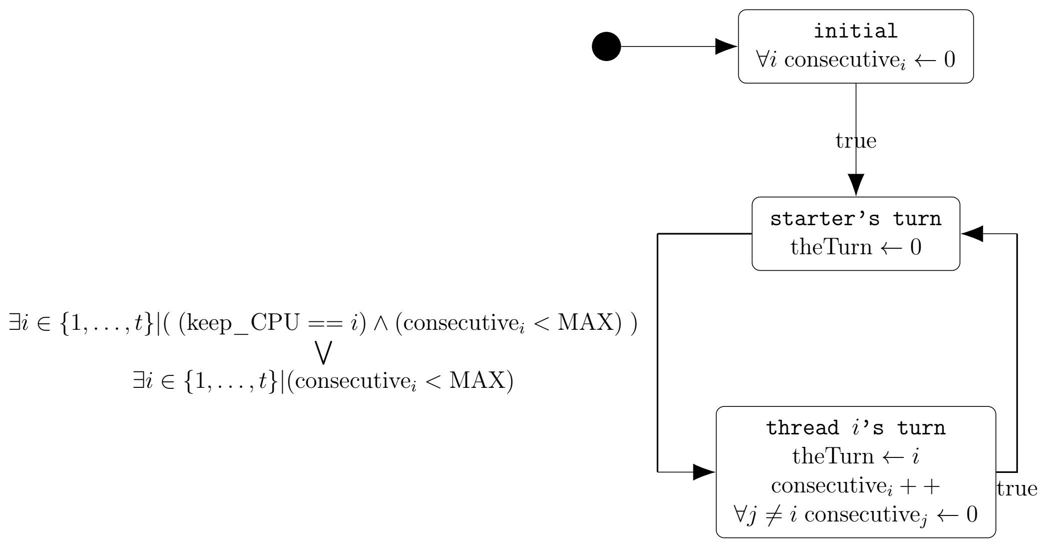

initial
$\forall i$ consecutive$_i \leftarrow 0$
true
starter's turn
theTurn $\leftarrow 0$
$\exists i \in \{1,\ldots,t\} | (\ (\text{keep\_CPU} == i) \wedge (\text{consecutive}_i < \text{MAX})\ )$
$\bigvee$
$\exists i \in \{1,\ldots,t\} | (\text{consecutive}_i < \text{MAX})$
thread i's turn
theTurn $\leftarrow i$
consecutive$_i$ ++
$\forall j \neq i$ consecutive$_j \leftarrow 0$
true

Figure 2.3: A scheduler encoded as an LLFSM (the integer MAX is the maximum number of consecutive turns for a thread, and keep_CPU $\in \{1,\ldots,t\}$ non-deterministically decides which of the t threads grabs the CPU).

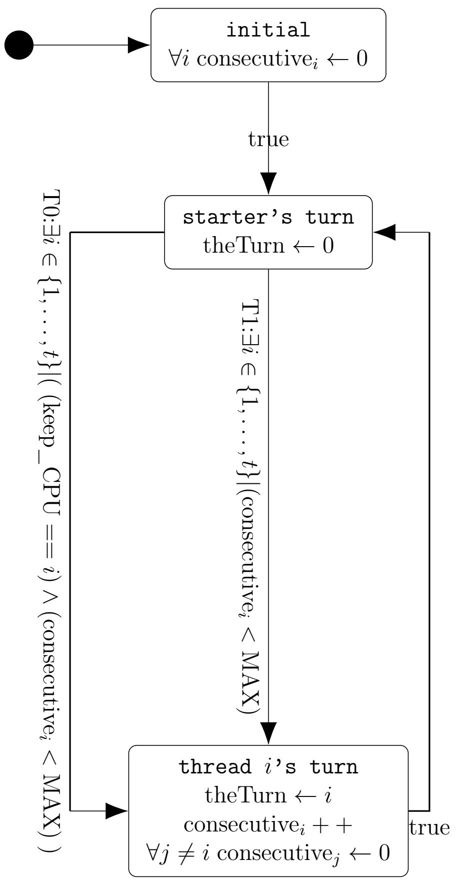

initial
$\forall i$ consecutive$_i \leftarrow 0$
true
starter's turn
theTurn $\leftarrow 0$
T0:$\exists i \in \{1,\ldots,t\} | (\ (\text{keep\_CPU} == i) \wedge (\text{consecutive}_i < \text{MAX})\ )$
T1:$\exists i \in \{1,\ldots,t\} | (\text{consecutive}_i < \text{MAX})$
thread i's turn
theTurn $\leftarrow i$
consecutive$_i$ ++
$\forall j \neq i$ consecutive$_j \leftarrow 0$
true

Figure 2.4: Equivalent LLFSM to Figure 2.3 where ordered transitions are used to express the short circuit evaluation of disjunctions.

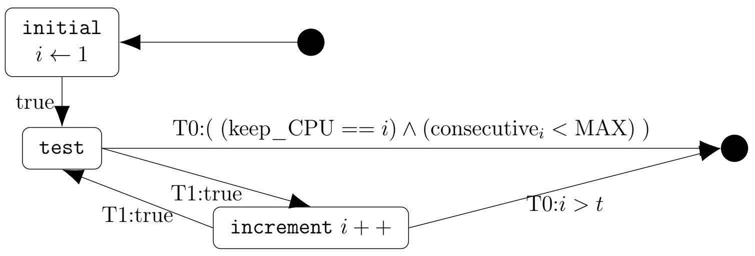


Figure 2.5: The expression $\exists i \in \{1,\ldots,t\}|($ (keep_CPU == i) $\wedge$ (consecutive$_i$ < MAX)). is really a fragment of an LLFSM.

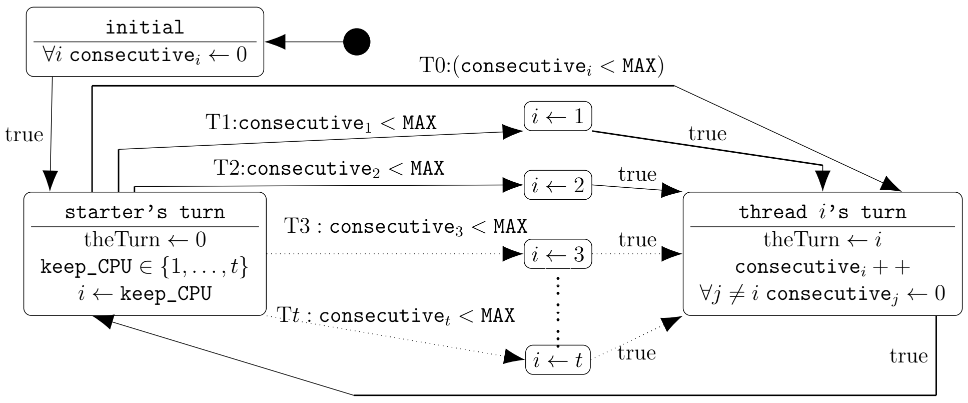


Figure 2.6: A scheduler encoded as an LLFSM (the integer MAX is the maximum number of consecutive turns for a thread, and keep_CPU $\in \{1,\ldots,t\}$ non-deterministically decides which of the t threads grabs the CPU).

i to the first value that satisfies the qualified predicate. Figure 2.5 illustrates the corresponding fragment of the LLFSM that corresponds to

$$\exists i \in \{1,\ldots,t\}|(\ (\texttt{keep\_CPU} == i) \wedge (\texttt{consecutive}_i < \text{MAX})\ ).$$

Once again this enables us to underline an important issue of the semantics of LLFSMs. This issue is regarding the appearance of the variable `keep_CPU` in transitions out of the state `starter's turn` in both Figure 2.3 and Figure 2.4. All transitions out of a state are evaluated in a snapshot of the environment where sensor variables are read only once and then included with a fixed value in the context where all the transitions are evaluated (see Section 3.8. Thus, the expression (`keep_CPU` == i) changes the value of i along the range of the sensor variable, but `keep_CPU` has a constant value over all transitions out of the state `starter's turn` until a transitions is deemed to fire or no transition fires.

Figure 2.6 presents another illustration of this scheduler, that removes the need for the operator $\exists$ but then uses an enumeration with several dots.

Note that the variables consecutive$_i$ for $i \in \{1,\ldots,t\}$ could be structured as an array. However, at the moment, the prototype does not offer facilities for LLFSMs with arrays. Thus, we will start by presenting the version for two thread. The examples that follow are in subfodler of the folder `CapConsecutiveTurnsSchedul`

Two threads with turns capped to a maximum

This example is in the folder

```
LLFSMTransformationExample
  └─Examples
      └─OutputOnlyPingPongTikiTaka.d
      └─RealTimeBounds.d
```

```
|   |__Fischer.d
|   |__FischerCopies.d
|   |__Microwave.d
|   |__TimedExpresion.d
|__SchedulersForLLFSMs.d
    |__CapConsecutiveTurnsScheduler
        |__TwoThreadsCapConsecutiveTurnsScheduler
```

So, move to the folder `TwoThreadsCapConsecutiveTurnsScheduler`. Like always, we can obtain a visualisation of the model with the target

```
make dot
```

Figure 2.7 presents the diagram for the case of two threads when we translate to `dot`.

There are a few aspects in which the generic form presented in Figure 2.3 and the instance of two threads of Figure 2.7 differ. The first is that the generic form Figure 2.3 always alternates a turn to thread 0 with any other thread. Here we have preferred to give only one turn to thread 0 as with the unrestricted scheduler of the previous section. This makes no difference for examples such as the Fischer algorithm that needs some initialisation, performed by a starter machine in the first position (that is position 0). Because such initialisation is in the LLFSM in position 0 and then reaches a terminal state, a turn alternating with other machines is a void turn as there is nothing to do.

Another aspect is that Figure 2.7 is the state `RESET`. This state is somewhat redundant and the transitions out of the states `Thread_1_Turn` and `Thread_2_Turn` could be arranged between each other. We believe the current display of the scheduler is clearer, although the behaviour in Figure 2.7 will update the variable `the_Turn` every two turns of the simulation when translated. Once for the corresponding thread assigning state (`Thread_1_Turn` or `Thread_2_Turn`) and once for the state `RESET`.

We can run this scheduler in LISP, first translating the model of Figure 2.7 with the target

```
make lisp
```

Then, as per the indication of the file `test.sh`, we run it with

```
source `make name`.sh 2> junk
```

We now supply to which thread we award the CPU (warning that as explained before this takes effect when the simulation is the state `RESET`). However, a quick way of observing the behaviour of the scheduler is to repeatedly in pairs provide the value 1 to the variable `KEEP_CPU`. If you constantly provide the value 1, you will see the variable `the_Turn` takes the value 1 and the variable `consecutive_1` increments until it reaches 4. At that time, a turn is awarded to thread 2 and we see `the_Turn` take the value 2. Playing with the simulator you shall become convinced you cannot starve a thread. Similarly if you run the `C` version by using the target

```
make C
```

This produces a file with extension `.c` that must be compiled as indicated by the `test.sh` file.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

Recall that to see the effect on the whiteboard variable `the_Turn` you should run the C program in verbose node with the option -v.

```
./executables/TwoThreadsCapConsecutiveTurnsScheduler.exe -v
```

In this case that the variable `the_Turn` is updated every two loops but since we are queried for the sensor variable when needed, the change seems immediate.

In the current version of the prototype translation to MIPS not all aspects used by this scheduler (such as a Boolean expression to test the equality of two arithmetic expressions). Therefore we do not execute this scheduler in MIPS.

We move on to the formal verification using NuSMV. The target

```
make smv
```

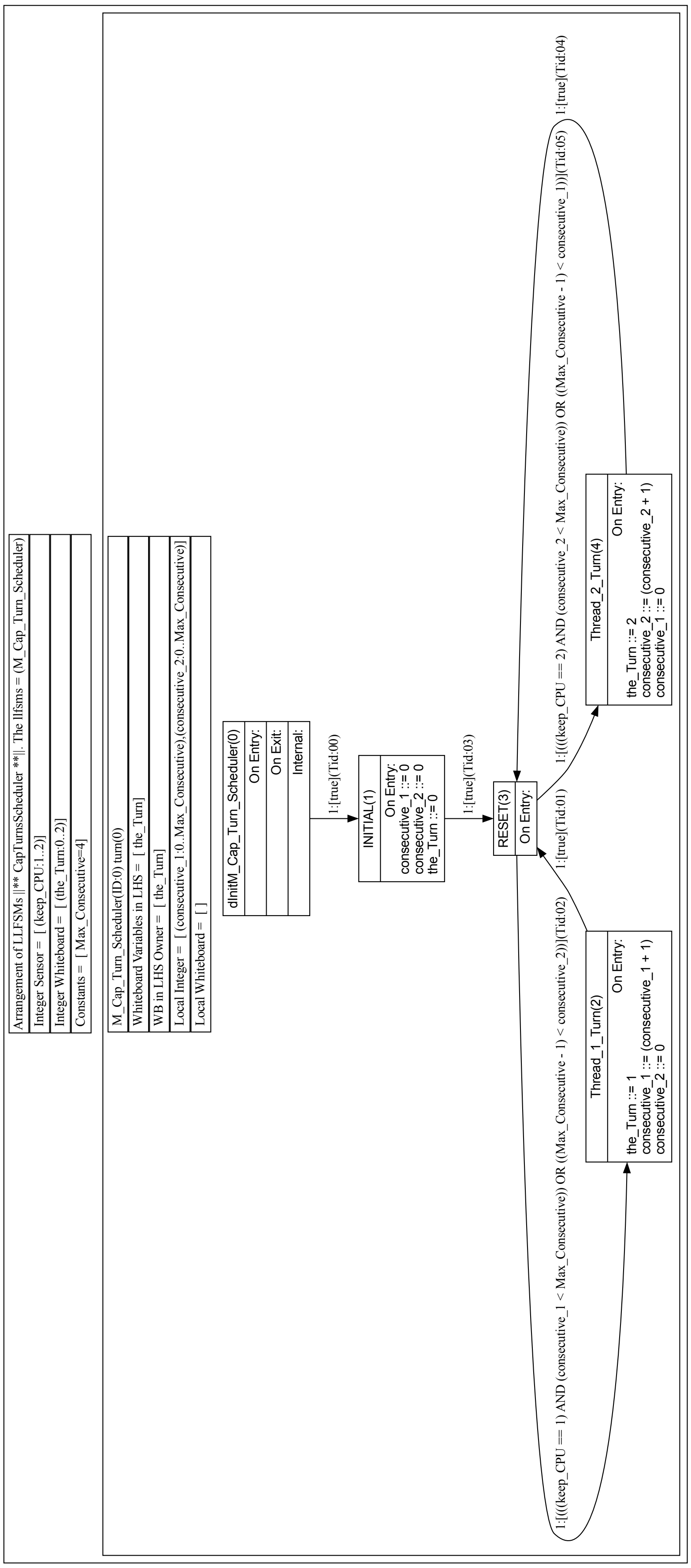

Arrangement of LLFSMs ||** CapTurnsScheduler **||. The llfsms = (M_Cap_Turn_Scheduler)
Integer Sensor = [(keep_CPU:1..2)]
Integer Whiteboard = [(the_Turn:0..2)]
Constants = [Max_Consecutive=4]
M_Cap_Turn_Scheduler(ID:0) turn(0)
Whiteboard Variables in LHS = [the_Turn]
WB in LHS Owner = [the_Turn]
Local Integer = [(consecutive_1:0..Max_Consecutive),(consecutive_2:0..Max_Consecutive)]
Local Whiteboard = [ ]
dInitM_Cap_Turn_Scheduler(0)
On Entry:
On Exit:
Internal:
1:[true](Tid:00)
INITIAL(1)
On Entry:
consecutive_1 ::= 0
consecutive_2 ::= 0
the_Turn ::= 0
1:[true](Tid:03)
RESET(3)
On Entry:
1:[true](Tid:01)
1:[(((keep_CPU == 2) AND (consecutive_2 < Max_Consecutive)) OR ((Max_Consecutive - 1) < consecutive_1))](Tid:05)
1:[true](Tid:04)
Thread_2_Turn(4)
On Entry:
the_Turn ::= 2
consecutive_2 ::= (consecutive_2 + 1)
consecutive_1 ::= 0
Thread_1_Turn(2)
On Entry:
the_Turn ::= 1
consecutive_1 ::= (consecutive_1 + 1)
consecutive_2 ::= 0
1:[(((keep_CPU == 1) AND (consecutive_1 < Max_Consecutive)) OR ((Max_Consecutive - 1) < consecutive_2))](Tid:02)

Figure 2.7: The scheduler of Figure 2.3 in the case that the number of threads is 2.

produces the NuSMV model and check fundamental properties, such as in being a deadlock-free model. We now verify similar properties as before. The properties that it is possible for each thread to have a turn remain true (see the corresponding file `ctl-properties.nesmv`). First for thread one.

```
AG (EF w_b.the_Turn = 1)
```

and in general:

```
VAR
        -- i and arbitrary thread
        i : 1..2;
CTLSPEC
AG (EF w_b.the_Turn = i)
```

However, now the property that showed starvation is false. For thread 1 is as follows.

```
EF (EG w_b.the_Turn = 1)
```

For thread 2 is as follows.

```
EF (EG w_b.the_Turn = 2)
```

Now these are in the file `ctl-false-properties.nusmv` and we use the target

```
make test-false-ctl-properties
```

Analogously with LTL. The properties that showed starvation for the Unrestricted Scheduler are now true. For example, we can present a property for showing that at any time, eventually thread 1 gets a turn.

```
G ( F w_b.the_Turn = 1)
```

This one and other LTL are also verified with the target

```
make test-properties
```

The NuSMV verifier shows the following two properties are true.

```
G (w_b.the_Turn = 1 ->  F w_b.the_Turn = 2)
G (w_b.the_Turn = 2 ->  F !(w_b.the_Turn = 2))
```

We can also verify some of the LTL properties with `tlc`. The file `properties.tla` has the following content.

```
[] <> ( the_Turn=1 )
[] <> ( the_Turn=2 )
[] ( the_Turn=2 ~> <> (~ (the_Turn=2))  )
```

To build the TLA model, we issue the following command.

```
make tla
```

To build the TLA model, the configuration file with the properties is the file `properties.tla` and to run the verification, the command is as follows.

```
make test-tla-properties
```

Thus, with `tlc` we know that

1. no matter where is the computation, eventually thread 1 will get a turn,
2. no matter where is the computation, eventually thread 2 will get a turn, and
3. every time thread 2 has a turn, then eventually it will lose it.

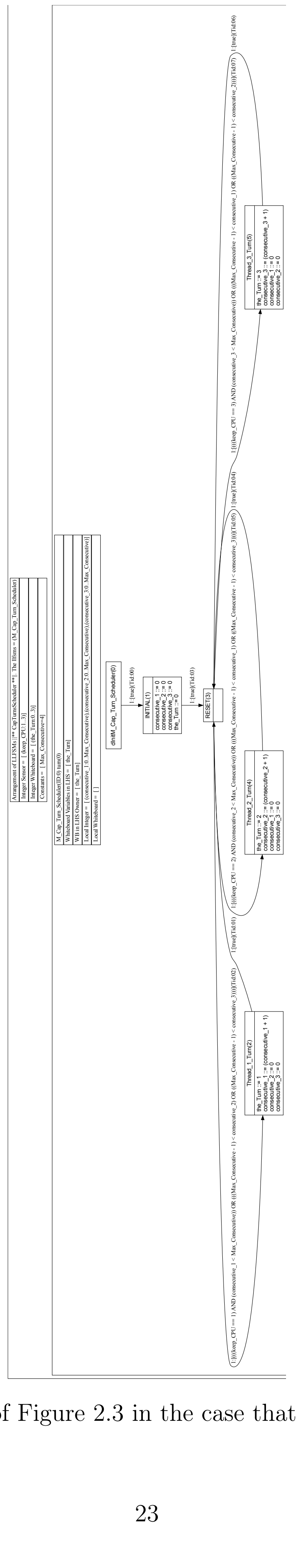

Arrangement of LLFSMs ||** CapTurnsScheduler **||. The llfsms = (M_Cap_Turn_Scheduler)
Integer Sensor = [(keep_CPU:1..3)]
Integer Whiteboard = [(the_Turn:0..3)]
Constants = [Max_Consecutive=4]
M_Cap_Turn_Scheduler(ID:0) turn(0)
Whiteboard Variables in LHS = [the_Turn]
WB in LHS Owner = [the_Turn]
Local Integer = [(consecutive_1:0..Max_Consecutive),(consecutive_2:0..Max_Consecutive),(consecutive_3:0..Max_Consecutive)]
Local Whiteboard = [ ]
dInitM_Cap_Turn_Scheduler(0)
1:[true](Tid:00)
INITIAL(1)
consecutive_1 ::= 0
consecutive_2 ::= 0
consecutive_3 ::= 0
the_Turn ::= 0
1:[true](Tid:03)
RESET(3)
Thread_1_Turn(2)
the_Turn ::= 1
consecutive_1 ::= (consecutive_1 + 1)
consecutive_2 ::= 0
consecutive_3 ::= 0
Thread_2_Turn(4)
the_Turn ::= 2
consecutive_2 ::= (consecutive_2 + 1)
consecutive_1 ::= 0
consecutive_3 ::= 0
Thread_3_Turn(5)
the_Turn ::= 3
consecutive_3 ::= (consecutive_3 + 1)
consecutive_1 ::= 0
consecutive_2 ::= 0
1:[(((keep_CPU == 1) AND (consecutive_1 < Max_Consecutive)) OR (((Max_Consecutive - 1) < consecutive_2) OR ((Max_Consecutive - 1) < consecutive_3)))](Tid:02)
1:[true](Tid:01)
1:[(((keep_CPU == 2) AND (consecutive_2 < Max_Consecutive)) OR (((Max_Consecutive - 1) < consecutive_1) OR ((Max_Consecutive - 1) < consecutive_3)))](Tid:05)
1:[true](Tid:04)
1:[(((keep_CPU == 3) AND (consecutive_3 < Max_Consecutive)) OR (((Max_Consecutive - 1) < consecutive_1) OR ((Max_Consecutive - 1) < consecutive_2)))](Tid:07)
1:[true](Tid:06)

Figure 2.8: The scheduler of Figure 2.3 in the case that the number of threads is 3.

Three threads with consecutive turns capped

Figure 2.8 shows the case where the scheduler handles three threads. Again, we can obtain visualisation by translating the model to dot. The following command produces the visualisation in Figure 2.8.

```
make brief-dot
```

We can execute this model with lisp by translating to lisp.

```
make lisp
```

And also we can translate to C.

```
make C
```

With this scheduler, it is possible to assign turn to any thread.

```
make test-properties
```

we can verify

```
AG (EF w_b.the_Turn = i)
```

The important aspects that this model shows is that we can verify two types of properties.

No monopolysation : As opposed to the unrestricted scheduler where a thread could monopolise the turns for ever, this scheduler, independent of the number of threads does not allow for a single thread to grab all the turns. For instance, with the target

```
make test-false-ctl-properties
```

shows that the following property is false.

```
EF (EG w_b.the_Turn = 1)
```

If it were true, then at any point, there would be a way to find the start of a path that always awards all the turns to the first thread.

Possible starvation : Now that there are more then two threads, it is possible to starve a thread (as the turns can be alternated by two thread without reaching their maximum number of consecutive turns). We can verify that at every point, there is always a computation that avoids awarding a turn to thread one. The target

```
make test-false-ltl-properties
```

shows that NuSMV finds false the following LTL formula.

```
G ( F w_b.the_Turn = 1)
```

The target will produce a log file where we can find a path of computation where thread 2 and thread 3 alternate the turn for ever, and thus thread 1 is never awarded a turn.

Model with three threads

In is not immediate that the general form of this scheduler, shown in Figure 2.3, is correct in its intent. Note that the transition says that as soon as we find an i such that there is $j \neq i$, where thread j has reached the maximum number of consecutive turns, we can assign the next turn to thread i.

The question is, how do we know that thread i has not also reached its maximum number of turns. The justification seems to be that, when a thread j is the first one to reach its maximum number of turns, it does so and all others get their counter set to zero. Then, some other thread gets the next turn (say i) and j's counter goes back to zero. Any thread that gets a turn sets the others' counters to zero. This seems to suggest there is never a state where two threads reach the maximum number of turns. This is something we should verify with the model checkers.

In the folder `ThreeThreadsCapConsecutiveTurnsScheduler` the file `ltl-properties` has the property:

```
G ( X (M_Cap_Turn_Scheduler.consecutive_1 = g_c.Max_Consecutive) ->
     ( ! (X (M_Cap_Turn_Scheduler.consecutive_2 = g_c.Max_Consecutive) )
      &  ! (X (M_Cap_Turn_Scheduler.consecutive_3 = g_c.Max_Consecutive) )
     )
)
```

This LTL property indicates that, always, at least for thread 1, except for the initial Kripke-state, if this thread reaches its maximum number of consecutive turns, the other two threads do not reach the maximum number of threads. Analogous LTL formulas for the other two threads appear in the file `ltl-properties`. Thus, always, if a thread has reached its maximum number of consecutive turns, there is always another thread that has not reached its maximum number of consecutive turns. Another way to express a virtue of this scheduler is as follows.

```
G (
   M_Cap_Turn_Scheduler.consecutive_1 = g_c.Max_Consecutive
 -> X(X(M_Cap_Turn_Scheduler.consecutive_1 = 0))
)
```

Always, in no more that two Kripke states after tread 1 has reached its maximum number of consecutive turns, another thread has a turn, and thread 1 does not have the next turn (as its consecutive turns goes down to zero).

We can use the translation to TLA.

```
make tla
```

And then, with the following command we prove analogous properties with `TLC`.

```
make test-tla-properties
```

The properties verified are in the file `properties.tla`. Recall that, because of the restrictions of the `TLC` tools they must be written in a single line

```
[] ( the_Turn=1 ~> (the_Turn=2 \/ the_Turn=3 ) )
[] ( the_Turn=2 ~> ~(the_Turn=2) )

[] ((~(M_Cap_Turn_SchedulerState = "dInitM_Cap_Turn_Scheduler") /\ M_Cap_Turn_Scheduler_consecutive_1 = Max_Consecutive)
   => (M_Cap_Turn_Scheduler_consecutive_2 = 0) )

[] [M_Cap_Turn_Scheduler_consecutive_1 = Max_Consecutive
=> M_Cap_Turn_Scheduler_consecutive_1' = 0
    ]_M_Cap_Turn_Scheduler_consecutive_1
```

The first one is plain LTL and says that always, if thread 1 has a turn, eventually one of the two other threads (thread 2 or thread 3) will have a turn. Similarly, the second one says that if thread 2 has a turn, then eventually it will not have a turn (thus, threads cannot monopolise the CPU). The third one also says that, always, except for the pseudo-initial kripke state, if thread 1 has reached its maximum umber of turns, then thread 2 (it could be thread 3) has had no turn. The last one is a bit different than standard LTL, but says that always, eliminating stuttering states in the projection of the variable `M_Cap_Turn_Scheduler_consecutive_1`, if thread 1 reaches its maximum number of state, the next value it will have is zero (and thus another thread received a turn).

More than three threads

The folder `kThreadsCapConsecutiveTurnsScheduler` has a model with up to five threads. The purpose of this model is to illustrate the features of the scheduler guided by capping the consecutive turns. Since our current prototype does not support arrays of integers, the number of states is unnecessarily large. The point is to illustrate that the design of this scheduler generalizes and that it can be incorporated as part of the translation of a multi-threaded algorithm (like Fischer's mutual exclusion) for both, the implementations (say to `C`) of for verification (smv or `TLA+`).

The model can be translated to LISP with the following command.

```
make lisp
```

Then it can be executed as with all our LISP translations.

```
source `make name`.sh 2> junk
```

Running the LISP version is a bit slow because every transition in the model request the values of the sensor variables even if they do not affect the coming transition. However, it has the advantage that the execution displays all variables (including local variables). With the `C` translation, the amount of output of the execution can be regulated and the interaction is also only when the sensor variable is needed. We translate to a `C` executable as follows.

```
make C
```

We need to compile as before.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

We can run with no command line arguments.

```
./executables/kThreadsCapConsecutiveTurnsScheduler.exe
```

We can supply a choice of thread repeatedly, for example, when the execution ask

```
The SENSOR variable *> keep_CPU <* with range [1,5] is read now:
```

we can supply the value 1 five times. We will notice that the first four, the turn is awarded to thread 1 but the fifth time, among the output we see

```
This is the moment *> the_assigned_turn <* is given the value 2
```

You can experiment and you will notice that you cannot award more than four consecutive turns to a thread, and if you arrive to the maximum number of consecutive turns of thread 2 or a larger numbered thread, then thread 1 gets a turn. Also, the scheduler awards a turn to thread 2 if thread 1 is requested for a fifth consecutive turn.

With the command

```
make test-properties
```

a series of properties in LTL and CTL are verified with NuSMV. For instance

```
VAR
-- i and arbitrary thread
             i : 1..5;
CTLSPEC
AG (EF Effector.the_assigned_turn = i)
```

is shown to be true. That is, always, no matter what state, it is possible to eventually award the turn to any thread. But it is also possible to show that a thread may starve.

```
make test-false-ltl-properties
```

shows that the property

```
G ( X ( F Effector.the_assigned_turn = 1))
```

is false. Thus, no matter what state, the computation from there can starve thread 1.

2.4.3 The fair scheduler

We achieve a scheduler that has freedom for the threads (any thread can be requested next) like the unrestricted scheduler, and also guarantees no starvation on any thread. Recall that the freedom of the unrestricted scheduler came with the price that threads could starve (even with only two threads). The previous scheduler that capped the number of consecutive turns allows any thread to be requested and avoid starvation but only for two threads (three or more threads resulted in the possibility of starving a thread). In this section, the scheduler will never starve a thread for an arbitrary number of threads (and yet any thread can be requested at any time).

This scheduler caps the maximum number of times a thread i has been waiting. That is, every time the turn is assigned to another thread, the value of $\texttt{waiting}_i$ increments by one. The scheduler iterates over two phases when assigning a turn.

First Phase : It checks if there is a thread i that has been waiting beyond `MAX_WAIT` (a constant that indicates the tolerance to have a turn).

- If there is i with $\texttt{waiting}_i$ greater or equal than`MAX_WAIT`, the scheduler assigns the turn to i no matter what thread was requested.

$$\texttt{the\_Turn} \rightarrow i$$

- otherwise, the requested thread gets the turn.

$$\texttt{the\_Turn} \rightarrow \text{requested thread}$$

Second Phase : For all threads j that did not get the turn in Phase 1, their waiting counters increases.

$$\texttt{waiting}_j \rightarrow \texttt{waiting}_j + 1$$

while the thread that got the turn has its counter reset.

$$\texttt{waiting}_{\texttt{the\_Turn}} \rightarrow 0$$

We have two implementations of this scheduler in our examples. These two implementations are in the folder `CapWaitingTurnsScheduler`.

```
LLFSMTransformationExample
└─ Examples
   ├─ OutputOnlyPingPongTikiTaka.d
   ├─ RealTimeBounds.d
   │  ├─ Fischer.d
   │  ├─ FischerCopies.d
   │  ├─ Microwave.d
   │  └─ TimedExpresion.d
   └─ SchedulersForLLFSMs.d
      ├─ CapConsecutiveTurnsScheduler
      └─ CapWaitingTurnsScheduler
         ├─ ThreeThreadsCapWaitingTurnsScheduler
         └─ kThreadsCapWaitingTurnsScheduler
```

The first implementation is the folder `ThreeThreadsCapWaitingTurnsScheduler`. As the name indicates, the scheduler works with three threads. We can again visualise the model with the command

```
make brief-dot
```

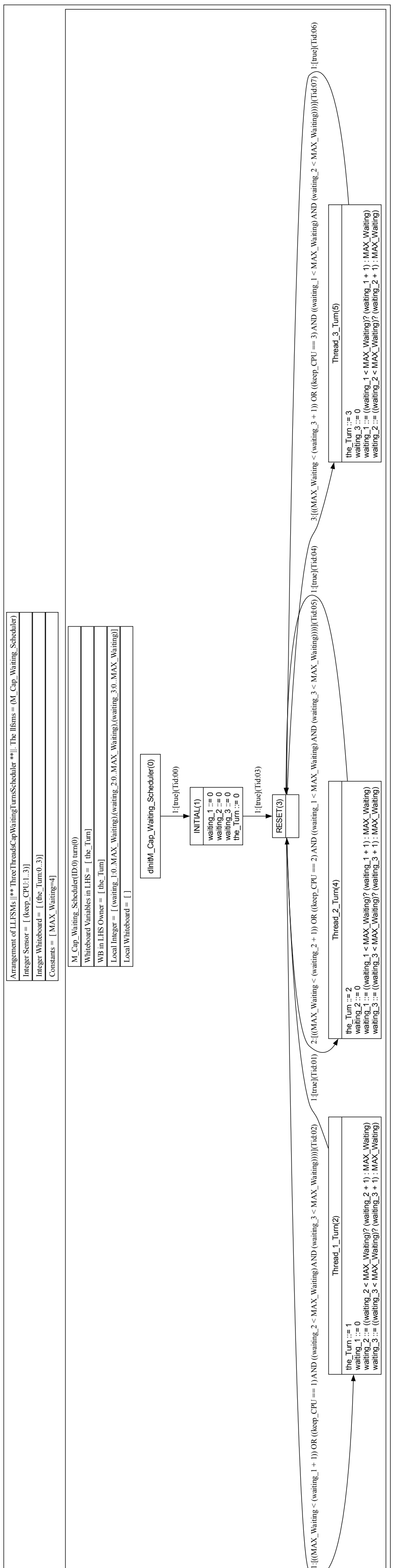

Arrangement of LLFSMs ||** ThreeThreadsCapWaitingTurnsScheduler **||. The llfsms = (M_Cap_Waiting_Scheduler)
Integer Sensor = [(keep_CPU:1..3)]
Integer Whiteboard = [(the_Turn:0..3)]
Constants = [MAX_Waiting=4]
M_Cap_Waiting_Scheduler(ID:0) turn(0)
Whiteboard Variables in LHS = [the_Turn]
WB in LHS Owner = [the_Turn]
Local Integer = [(waiting_1:0..MAX_Waiting),(waiting_2:0..MAX_Waiting),(waiting_3:0..MAX_Waiting)]
Local Whiteboard = [ ]
dInitM_Cap_Waiting_Scheduler(0)
1:[true](Tid:00)
INITIAL(1)
waiting_1 ::= 0
waiting_2 ::= 0
waiting_3 ::= 0
the_Turn ::= 0
1:[true](Tid:03)
RESET(3)
1:[((MAX_Waiting < (waiting_1 + 1)) OR ((keep_CPU == 1) AND ((waiting_2 < MAX_Waiting) AND (waiting_3 < MAX_Waiting))))](Tid:02)
1:[true](Tid:01)
2:[((MAX_Waiting < (waiting_2 + 1)) OR ((keep_CPU == 2) AND ((waiting_1 < MAX_Waiting) AND (waiting_3 < MAX_Waiting))))](Tid:05)
1:[true](Tid:04)
3:[((MAX_Waiting < (waiting_3 + 1)) OR ((keep_CPU == 3) AND ((waiting_1 < MAX_Waiting) AND (waiting_2 < MAX_Waiting))))](Tid:07)
1:[true](Tid:06)
Thread_1_Turn(2)
the_Turn ::= 1
waiting_1 ::= 0
waiting_2 ::= ((waiting_2 < MAX_Waiting)? (waiting_2 + 1) : MAX_Waiting)
waiting_3 ::= ((waiting_3 < MAX_Waiting)? (waiting_3 + 1) : MAX_Waiting)
Thread_2_Turn(4)
the_Turn ::= 2
waiting_2 ::= 0
waiting_1 ::= ((waiting_1 < MAX_Waiting)? (waiting_1 + 1) : MAX_Waiting)
waiting_3 ::= ((waiting_3 < MAX_Waiting)? (waiting_3 + 1) : MAX_Waiting)
Thread_3_Turn(5)
the_Turn ::= 3
waiting_3 ::= 0
waiting_1 ::= ((waiting_1 < MAX_Waiting)? (waiting_1 + 1) : MAX_Waiting)
waiting_2 ::= ((waiting_2 < MAX_Waiting)? (waiting_2 + 1) : MAX_Waiting)

Figure 2.9: Visualisation produced by `graphviz` when translating a model of the scheduler for three threads that limits how long the thread has been waiting to get the CPU.

The graphviz does not produce an image that uses the space well; thus the readability of the text in Figure 2.9 is limited.

We can run this scheduler in LISP by translating the model to LISP.

```
make lisp
```

We then run the resulting translation as follows.

```
source `make name`.sh 2> junk
```

When you run the LISP scheduler you are asked for values of the sensor variables after each ringlet is executed. Thus, you will be asked for the value of KEEP_CPU (by default LISP convert variables to all upper-case letters). Although only at one point the sensor variable is used. Repeat a value in $\{1, 2, 3\}$ and you will see the variable THE_TURN (it will show in the section WHITEBOARD-VALUES of the output) change to your input value. Also, in the section LOCAL-VALUES, you will see the values for WAITING_1 WAITING_2 and WAITING_3. You will notice that even if you persist with a fixed value for many times, once a waiting variable reaches 4, the value of the variable THE_TURN will change to this, showing the thread that has waited long enough gets a turn. You can translate the model changing the parameter of how long to wait. For example,

```
make lisp MAX_TURNS=6
```

will make translation where now thread has to miss 6 times being awarded the turn. However, you cannot make a thread starve, all eventually get a turn.

The same model can be run with C.

```
make C
```

We need to compile using the same command as before.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

We recommend you run it in verbose mode (option -v) so you can see the values of the assigned turn in the whiteboard variable the_Turn.

```
./executables/ThreeThreadsCapConsecutiveTurnsScheduler.exe -v
```

Now for verification, you can run a NuSMV check for similar properties as before.

```
make test-properties
```

Moreover, we now cannot starve a thread, so a property that a thread may never has a turn is false.

```
make test-false-ctl-properties
```

In Section 3.1 we will see that there are two types of real-time properties. First, those properties that guarantee that it is not to long before something happens. The second type are those properties that guarantee that something has at least a minimal duration. With the following LTL property in NUSMV we check that always, if thread 1 does not have a turn, then thread 1 will have a turn after a maximum of 11 Kripke states.

```
LTLSPEC
G   (                            -- At all future states:
    (! (w_b.the_Turn=1))                 --  if thread 1 does not have a turn,
    ->                                       -- then:
        (F[0,11]                             -- after a maximum of 11 transitions,
          w_b.the_Turn=1 )    -- thread 1 has a turn.
    )
```

This shows not only that a thread will eventually have a turn (and not starve), but exactly how long will it have to wait (in Kripke states, or ringlet executions, which can be bounded my finding the maximum execution time of all ringlets in the model of the scheduler).

For a real-time property of the second type we have the following LTL property for NuSMV.

```
G   (                            -- At all future states :
        (  (! (w_b.the_Turn=1)) & X ((w_b.the_Turn=1) ) )  -- if thread 1 just got a turn

   ->                            -- then
        ( H[0,1]                 -- it has been waiting
          (! (w_b.the_Turn=1))   -- without a turn
        )
  )
```

This property says that a property always waits at least one Kripke state to get a turn.

The second example of this fair-scheduler is a generalisation to k threads in the example under the folder `kThreadsCapWaitingTurnsScheduler`. However, the example is only of five threads because our prototype does not have an array of integers. However, NuSMV and TLA+ do have them and our translator can generate this scheduler as it translated the arrangement because at that stage the number of machine in the arrangement is known. We invite you to generate the PDF of the model with the command

```
make brief-dot
```

and inspect the resulting file

```
kThreadsCapWaitingTurnsScheduler-brief.pdf
```

You will see integer variables `waiting_i`, for $i = 1, \ldots, i$, which could be implemented as an array.

You can generate an executable for LISP.

```
make lisp
```

And execute the resulting translation as follows.

```
source `make name`.sh 2> junk
```

This time the output variable of the scheduler is `THE_ASSIGNED_TURN` and is shown with all other variables. Running the lisp version is a bit tedious because again an input is requested in every ringlet although it only matters in one state.

Thus, we suggest you run the C version.

```
make C
```

We need to compile using the same command as before.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

Run it without any verbose option.

```
./executables/kThreadsCapWaitingTurnsScheduler.exe
```

You will still have to input at least five times a value for `keep_CPU` that is not relevant because the model loops around the five threads to update the `waiting` variable of each. The `C` scheduler needs to build the context in case the execution goes to the state `RESET`.

However, the interesting aspects are the properties. The command

```
make test-properties
```

verifies, with NuSMV for each thread $i \in \{1, \ldots, 5\}$, properties of the form

```
G (!(Effector.the_assigned_turn = i) ->  F Effector.the_assigned_turn = i)
G !(M_Cap_Waiting_Scheduler.waiting_i > g_c.MAX_WAITING)
```

That is, always, if thread i does not have the turn, it will eventually get it. While the second one is unnecessary because the model has already defined the range of `waiting_i` we place it here because it may give the impression that a thread will never have to wait more that the constant `g_c.MAX_WAITING`. The model does not increase `waiting_i` any more once it has reached `g_c.MAX_WAITING`.

However, this enables us to have a parameterised model you can extend how long a thread waits.

```
make test-properties MAX_WAITING=6
```

With `MAX_WAITING=4` (the default in the Makefile), we can also produce the two types of real-time properties for this case.

```
LTLSPEC
-- Globally/Always, a thread that does not have a turn, will have a turn
-- after a maximum of transitions
G   (                            -- At all future states:
    (! (Effector.the_assigned_turn=1))          --  if thread 1 does not have a turn,
    ->                                      -- then:
        (F[0,104]                            -- after a maximum of 104 transitions,
          Effector.the_assigned_turn=1 )    -- thread 1 has a turn.
    )
--
LTLSPEC
-- Globally/Always, a thread waits at least
G   (                            -- At all future states :
        (  (! (Effector.the_assigned_turn=1)) & X ((Effector.the_assigned_turn=1) ) )  -- if

   ->                            -- then
        ( H[0,2]                 -- it has been waiting
          (! (Effector.the_assigned_turn=1))    -- without a turn
        )
  )
```

You can test these two properties with the command

```
make test-realtime-properties
```

Chapter 3

Real time properties

3.1 Real-time-properties

According to Leslie Lamport there are two types of real-time properties.

Upper-bound type of property: If we reach a state s_0, then in no longer than k steps, property P will hold in all further states.

Lower-bound type of property: From the instant we reach state s_0, property P will hold globally in all states in at least the next k steps.

The upper-bound type of property indicates that the system assures that after X steps, the property P will hold (once we reach s_0). It is stronger than the *eventually* operator of temporal logics because it explicitly says that P will hold in k or fewer steps.

Similarly for the lower-bound type of property, which ensures a minimum number of steps during which P will hold.

We aim here to illustrate the formal verification of these two types of properties from of executable models. We start with a simple example to also illustrate timed-transitions in LLFSMs, and then we proceed with a realistic example widely discussed in the literature.

3.2 Real time properties with a simple LLFSM

A simple LLFSM with a timed-transition is the example in the directory `TimedExpresion.d` of the path `LLFSMTransformationExamples/Examples/RealTimeBounds.d`. The arrangement is named `TimedExpresion` and has only one LLFSM named `Only`. Figure 3.1 shows the result of generating a visualisation from the executable model with the corresponding target.

```
make dot
```

3.2.1 Executing with LISP

This model is executable. For a version is LISP we use the familiar target.

```
make lisp
```

Moreover, if we run it,

```
source `make name`.sh 2> junk
```

we will notice that the transition from `INITIAL` to `OTHER` is, to our human eyes, instantaneous. However, it is noticeable that the transition from `OTHER` to `INITIAL` takes two or more seconds. This is because in the LISP version the units of the `after` predicate are seconds. The semantics of the `after` predicate is that it becomes true only after the next turn is awarded to the LLFSM (a timed transition is always false the first time even if `after(0)` is used), and at least as many units have passed since entering the source state of the timed transition.

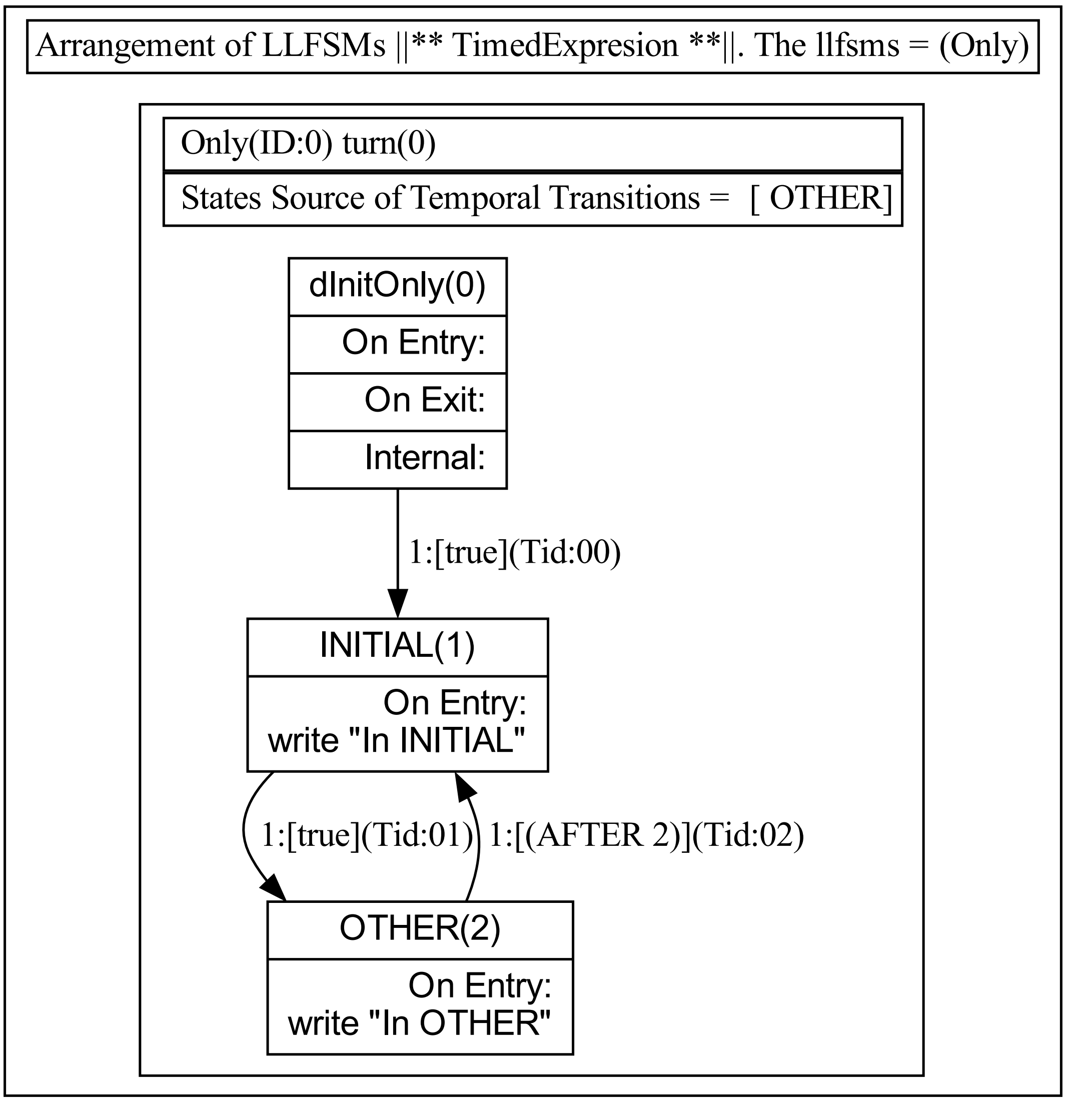


Figure 3.1: A LLFSM named `Only` with a timed transition.

3.2.2 Executing with C

The same model and with the same semantics can be executed in C. The target

```
make C
```

translates it to C, and the file `test.sh` has the following illustration on how to compile it

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

Then, we run it observing a very similar effect than the LISP versio.

```
./executables/TimedExpresion.exe
```

3.2.3 Executing with MIPS

Once more the same model can be translated into a model to text transformation with the corresponding target.

```
make mips
```

The, the file `test.sh` has the illustration on how to run it.

```
java -jar $MARS_HOME/Mars4_5.jar `make name`.asm
```

The execution also has the effect that the transition from `INITIAL` to `OTHER` seems instantaneous but the other way around takes some time. In the MIPS version, arrangements do not run forever; the simulation has always a limit.

3.2.4 Verification of lower-bound properties and upper-bound properties

The temporal logic TLA+ offers a subset of LTL where `X` (next) and `U` (until) do not exists. However, we can still use TLA+ to verify some temporal properties of this simple executable model. The target

```
make test-tla-properties
```

produces a translation and invokes the model checker `tlc` on a couple of properties. The properties are in the file `properties.tla` and say that

1. the state `INITIAL` for the machine `Only` occurs infinitely often in the future.
2. the state `OTHER` for the machine `Only` occurs infinitely often in the future.

Thus, LLFSM `Only` cannot remain in only one of the states.

We do not claim that NuSMV is more expressive than TLA+, but at least for the properties we want to illustrate here, our experience is that NuSMV is more accessible. In particular, we can formulate more properties in LTL and CTL. For example, the file `ltl-properties.nusmv` shows properties that are stronger than the two above properties because we show that `INITIAL` must be followed by `OTHER` and vice versa (so now we know there are only two states in the future for sure). Moreover, we can also explicitly verify that the transition from `INITIAL` to `OTHER` occurs infinitely often in the future and that so does the transition from `OTHER` to `INITIAL`. The formulation of these properties in NuSMV uses `X` and `U` and they seem particularly laborious if at all possible in TLA+.

So far these properties are not real time properties as we promised. However, the file `ltl-realtime-properti` has both types (upper-bound and lower-bound) real time properties. All properties are verified with the target

```
make test-properties
```

For example,

```
G (Only.At_INITIAL -> F[0,2] Only.At_OTHER)
```

is an upper-bound type of property saying that the LLFSM `Only` will never remain at the sate initial for longer than two steps.

Examples of lower-bound properties are also in this file.

```
G ((Only.At_OTHER & X Only.At_INITIAL) ->  H [0,11] Only.At_OTHER)
```

Using the history operator `H` in NuSMV we can say that the LLFSM stays at `OTHER` for at least 11 states. This latest lower-bound property has a complementary upper-bound property

```
G (Only.At_OTHER ->  F [0,12] Only.At_INITIAL)
```

that says also that `Only` does not stay further than 12 steps in state `OTHER`.

The file `properties.nusmv` uses CTL. We also can express here that always everywhere there is a path that takes the behaviour to `INITIAL` and analogously to `OTHER`. But the most interesting aspect is that the upper-bound properties can also be expressed in CTL (for NuSMV).

```
AG ((Only.At_INITIAL -> EBF 0..2 Only.At_OTHER) &
   (Only.At_OTHER -> EBF 0..12 Only.At_INITIAL))
```

3.3 Real time properties with the Microwave example

The microwave example is widely discussed in the literature, and we have many references in our papers to the work of others. It has an important reactive behaviour to the environment; namely, if the microwave is operating (cooking) and the door is opened, then the cooking must stop. This example is relevant for real time properties because when we express this in temporal logics it is insufficient that we verify that the cooking eventually stops but we must show that we can provide a precise bound of when will the microwave stop. That is, we will show here that the model can be used to determine precisely how many state changes are required since the sensor is indicating that the door is opened for the cooking to stop. Naturally, different implementations of the model in different micro-controllers result in different time measurements for how long is a change of state in the implementation, but this is just a matter of translating the worst execution time of the state transition to the abstract steps of the model.

The model also illustrates another subtle point. If the sensor for the door indicates the door is opened for an extremely short amount of time, then while the implementation is executing and examining something else, such short pulse of the door being opened would be ignored.

Thus, the microwave example is illustrative of the two types of real time properties Leslie Lamport argues that there are two fundamental types of possible properties one is interested in establishing. First, properties where all the successors no further than X transitions of one state S_0 satisfy a property P. This is the case that once the door is opened, there is a bound on the cooking being terminated. The other type of property is that something lasts at least a specified duration. That is, all the states in all computations after a particular state S_0 that are X steps or less, satisfy property P. In the case of the microwave example, after the cooking is completed, the bell must sound for a minimum amount of time.

The microwave example is in the directory `Microwave.d` that is the path

```
LFSMTransformationExamples/Examples/RealTimeBounds.d/.
```

This example is also parameterised. When the button is pushed and released time is added to the timer. The amount of time added is called the *increment.* Figure 3.2 shows the `Timer` llfsm of this arrangement. In the state `INCREASE_TIMER` we see that the current cooking time is increased by the increment when a button is pushed, and also released (the transition out of `INCREASE_TIMER` is labelled with `NOT button_Pushed`). If the Microwave is already cooking, keeping the `button_Pushed` prolongs the cooking indefinitely. That is, the button must be released for the timer to count down. The other parameter is the *maximum time* that can be set into the timer. This appears in the transition from `DECIDE` to `INCREASE_TIMER`. When the increment plus current cooking time is equal or larger than the maximum time, the state `INCREASE_TIMER` cannot be reached.

Therefore, when translating this example you will be asked to

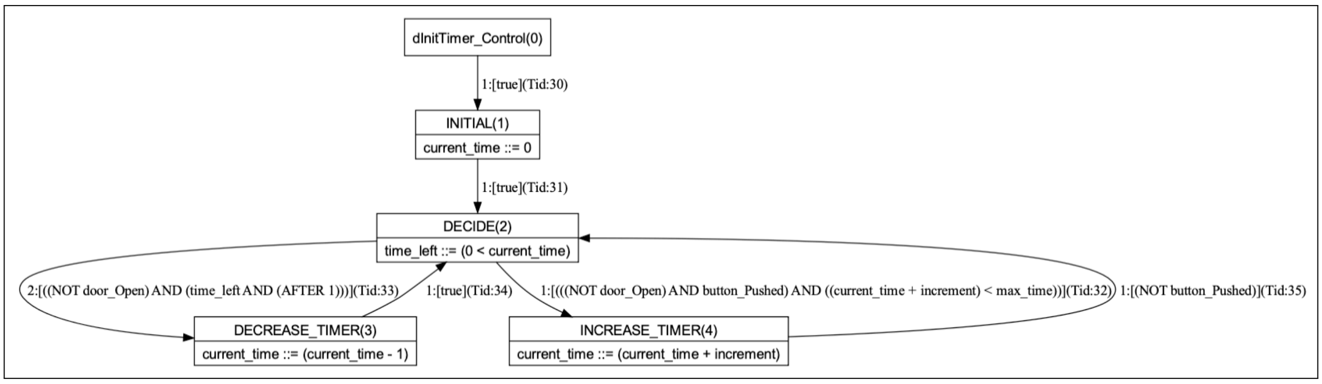

dInitTimer_Control(0)
1:[true](Tid:30)
INITIAL(1)
current_time ::= 0
1:[true](Tid:31)
DECIDE(2)
time_left ::= (0 < current_time)
2:[((NOT door_Open) AND (time_left AND (AFTER 1)))](Tid:33)
1:[true](Tid:34)
1:[(((NOT door_Open) AND button_Pushed) AND ((current_time + increment) < max_time))](Tid:32)
1:[(NOT button_Pushed)](Tid:35)
DECREASE_TIMER(3)
current_time ::= (current_time - 1)
INCREASE_TIMER(4)
current_time ::= (current_time + increment)

Figure 3.2: The llfsm that counts the cooking time in the Microwave example.

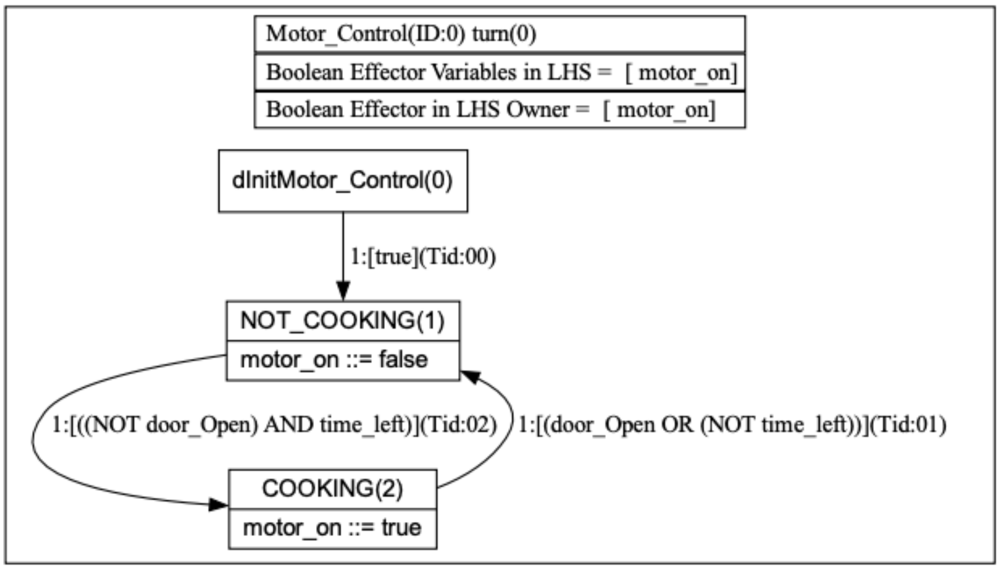

Motor_Control(ID:0) turn(0)
Boolean Effector Variables in LHS = [motor_on]
Boolean Effector in LHS Owner = [motor_on]
dInitMotor_Control(0)
1:[true](Tid:00)
NOT_COOKING(1)
motor_on ::= false
1:[((NOT door_Open) AND time_left)](Tid:02)
1:[(door_Open OR (NOT time_left))](Tid:01)
COOKING(2)
motor_on ::= true

Figure 3.3: LLFSM that sets cooking on or off.

```
Please enter integer value for parameter increment
```

Also, there is a maximum value the timer can reach. After that, no more time can be added to the timer. Thus, you will also be asked

```
Please enter integer value for parameter max_time
```

We recommend that you use small values for example set the increment to 1 or 2 and the maximum time to 5 or 6.

Like all other examples, we can produce a visual model with

```
make dot
```

The PDF `Microwave-regular.pdf` will show that the system has for LLFSMs. However,

```
make brief-dot
```

produces a slightly smaller drawing. The LLFSMs of the system are as follows. The first LLFSM in the arrangement controls whether cooking is on and off (Figure 3.3), the second one whether the light is on or off (Figure 3.4), the third whether the third actuator (the bell) is sounding or not (Figure 3.5). The last LLFSM control how long the cooking is going to be for (we saw it in Figure 3.2 but Figure 3.6 shows the list of variables), The cooking cannot happen while the door is open, but opening the door or cooking set the light on. To add time the door must be closed. Typically, the user should press and release the button

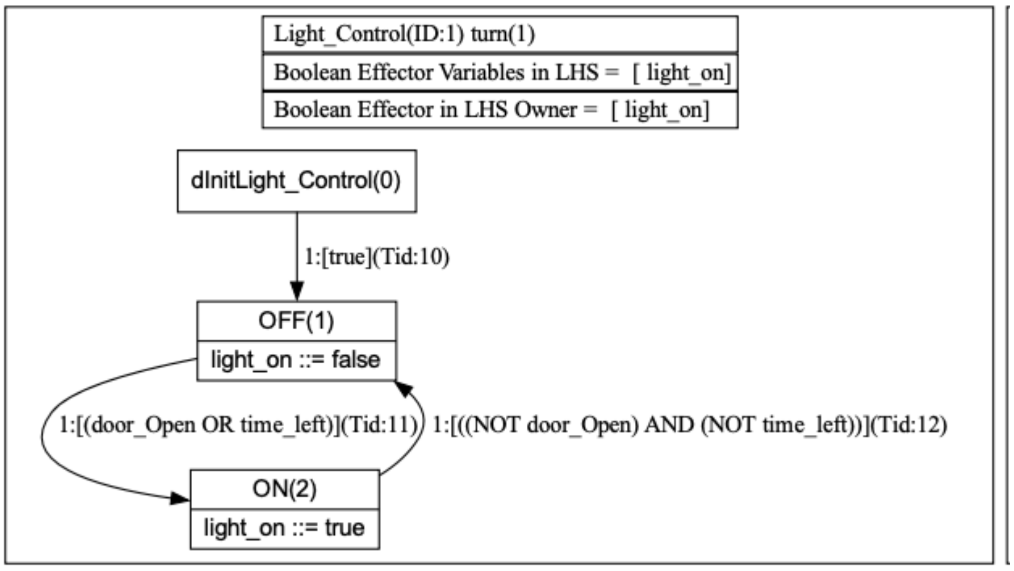

Light_Control(ID:1) turn(1)
Boolean Effector Variables in LHS = [light_on]
Boolean Effector in LHS Owner = [light_on]
dInitLight_Control(0)
1:[true](Tid:10)
OFF(1)
light_on ::= false
1:[(door_Open OR time_left)](Tid:11)
1:[((NOT door_Open) AND (NOT time_left))](Tid:12)
ON(2)
light_on ::= true

Figure 3.4: LLFSM that sets light on or off.

to set the desired cooking time with increments. The cooking will start and time will count down until it is zero and expiration of the time makes the bell ring for two time units. The exception is that opening the door pauses the timer, and halts the cooking. Once the door is closed again, the timer continues to count down and cooking resumes.

Thus, this model has two temporal transitions: one in the `Bell_Control` so the sound is set in state `RINGING` and is not silenced until at least two time units have elapsed and the behaviour moves to the state `SILENT`. The timed transition in the `Timer_Control` ensures that at least one time unit has elapsed before the value in the local variable `current_time` is decremented.

Timed transitions are implemented by timer-LLFSMs and this can be observed by

```
make timer-dot
```

Although only one timer-LLFSMs is shown here, there is one timer-LLFSMs for each timed transition in the arrangement. When execution arrives to the source state S_c of a timed-transition T, there is communication between the state machine and a timer-LLFSMs M_T for that transition T. The machine M_T counts rounds and the timed-transition T is now a signal from M_T that enough time has passed. This demonstrates that timed-transitions guarantee that at least a certain amount of time has elapsed (maybe more). Timed-transitions are not fired at exactly some time in the future.

3.3.1 Executable models (LISP, C and MIPS)

Producing executable for LISP

We can now generate the executable LISP with the same target that has been used for this purpose in earlier examples.

```
make lisp
```

As before, you can check `test.sh` for the ways to provide targets to the `Makefile` of this example. In `test.sh`, there is the command to tun the LISP version.

```
source `make name`.sh
```

Now, the LISP executable will ask you after the ringlet of each LLFSM what are the values of the two sensors of the system, whether the door is opened and whether the button is pushed. That the system interacts with the environment by these two sensor variables is shown in the summary of variables shown

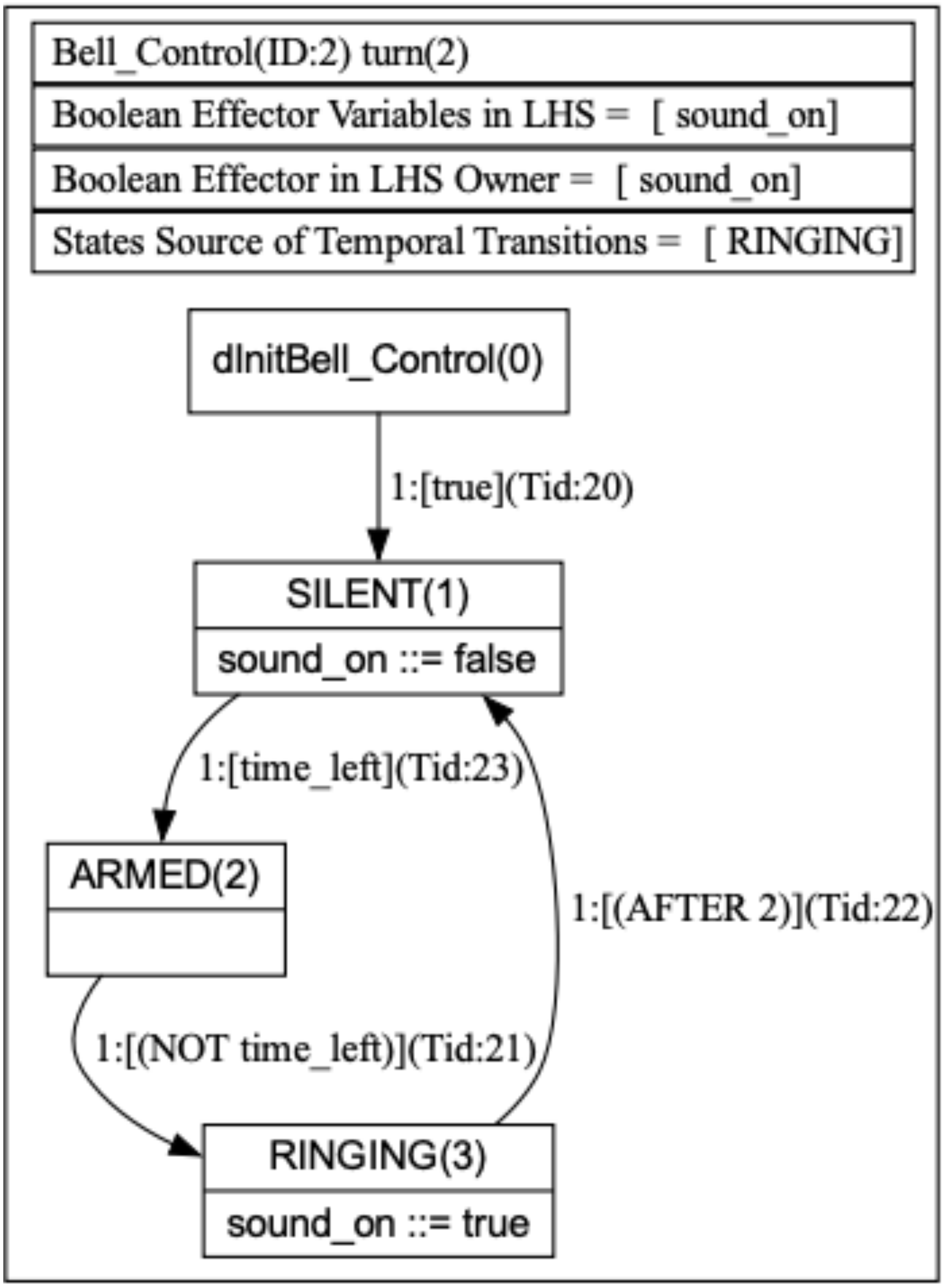

Bell_Control(ID:2) turn(2)
Boolean Effector Variables in LHS = [sound_on]
Boolean Effector in LHS Owner = [sound_on]
States Source of Temporal Transitions = [RINGING]
dInitBell_Control(0)
1:[true](Tid:20)
SILENT(1)
sound_on ::= false
1:[time_left](Tid:23)
ARMED(2)
1:[(AFTER 2)](Tid:22)
1:[(NOT time_left)](Tid:21)
RINGING(3)
sound_on ::= true

Figure 3.5: LLFSM that sets the bell to sounding or silent.

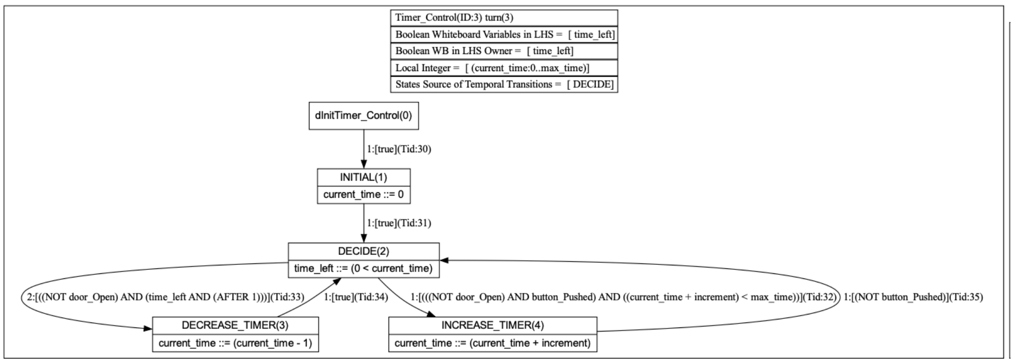
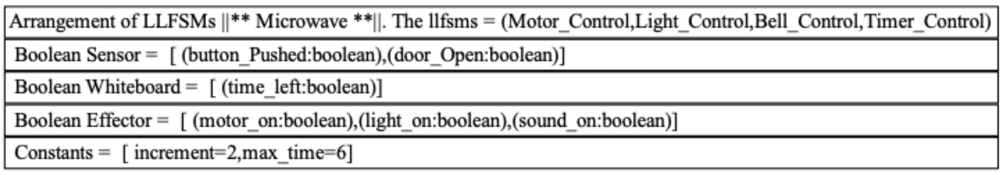

Timer_Control(ID:3) turn(3)
Boolean Whiteboard Variables in LHS = [time_left]
Boolean WB in LHS Owner = [time_left]
Local Integer = [(current_time:0..max_time)]
States Source of Temporal Transitions = [DECIDE]
dInitTimer_Control(0)
1:[true](Tid:30)
INITIAL(1)
current_time ::= 0
1:[true](Tid:31)
DECIDE(2)
time_left ::= (0 < current_time)
2:[((NOT door_Open) AND (time_left AND (AFTER 1)))](Tid:33)
1:[true](Tid:34)
1:[(((NOT door_Open) AND button_Pushed) AND ((current_time + increment) < max_time))](Tid:32)
1:[(NOT button_Pushed)](Tid:35)
DECREASE_TIMER(3)
current_time ::= (current_time - 1)
INCREASE_TIMER(4)
current_time ::= (current_time + increment)

Figure 3.6: LLFSM that controls the timer.



Arrangement of LLFSMs ||** Microwave **||. The llfsms = (Motor_Control,Light_Control,Bell_Control,Timer_Control)
Boolean Sensor = [(button_Pushed:boolean),(door_Open:boolean)]
Boolean Whiteboard = [(time_left:boolean)]
Boolean Effector = [(motor_on:boolean),(light_on:boolean),(sound_on:boolean)]
Constants = [increment=2,max_time=6]

Figure 3.7: The variables involved in the arragnement of four llfsms of the Microwave example.

in Figure 3.7. The sensor variables are read-only, meaning they cannot be in the left-hand side of an assignment statement.

Give the value 1 to `BUTTON_PUSHED` and 0 to `DOOR_OPEN` (that is keep the button pushed and the door closed). As you see the CURRENT-MACHINE-ID takes the values 0, 1, 2 and just then you can release the button (answer `BUTTON_PUSHED` 0 and `DOOR_OPEN` 0). Keep this for a while and you will see the variable `TIME_LEFT` becomes 1, and so after even further ringlets more `MOTOR_ON` becomes 1, and one ringlet after `LIGHT_ON 1` also becomes 1. If you keep `BUTTON_PUSHED` 0 and `DOOR_OPEN` 0) time will expire, the motor and the light will come off, the `SOUND_ON` will become 1 and then more further ringlets later then `SOUND_OFF` . Just recall you are being asked for the state of the sensors after the ringlet of each LLFSM in the arrangement.

Producing executable for C

We can run the same model in C.

```
make C
```

To compile for C the file `test.sh` has the following command.

```
make -f $PLAIN_C_SCHEDULER_INSTALLATION/MakefileForArrangement cfile=`make name`
```

This should produce the executable in a subdirectory `executables` and now you run it with

```
./executables/Microwave.exe
```

In this case, the simulation asks you for sensors when it needs to evaluate them (and not every ringlet). So it will ask you whether the door is open (and not about the button) because the first LLFSMs for cooking or the lights require whether the door is now open or not. Answer 1 the first time you are asked for the button to press it, and 0 the second time. Then just answer 0 for both the door being opened and the button being pushed. Because this adds time, the motor and the light will become on. If you keep answering 0 to all sensors, then the motor and light will become off, and the sound will be on for a bit and then off. After this simple trace of the behaviour you can explore any others you like.

Producing executable for MIPS

The translation of the executable model of LLFSMs to MIPS is the standard target.

```
make mips
```

After this, the `test.sh` shows how to run MARS and the MIPS executable.

```
java -jar $MARS_HOME/Mars4_5.jar `make name`.asm
```

In a similar form as in the C execution, you are asked for the two sensors when they are needed. Keeping the door closed by answering 0 to `doot_Open`, and then answering 1 to the first request of `button_Pushed` followed by answering 0 will add time. Add a few units and then just repeatedly answer 0. You will see that the microwave cooking and light go on until the time expires, which will result in the bell ringing for a while before finishing silent. You can then explore adding several units of time and opening the door when the cooking is going on to see that the cooking is paused.

3.3.2 Verification of properties

The microwave is an example where many properties can be tested. Some LTL properties are checked using NuSMV with

```
make test-ltl-properties
```

ToDo: Miguel said he would check these as some come out false, and it may be that returning false is correct to show others are not true due to vacuity. Similarly, we can test some other CTL properties

```
make test-ctl-properties
```

ToDo: Same issue as before, Miguel said he would check these as some come out false, and it may be that returning false is correct to show others are not true due to vacuity. And with TLA+ we can also check some properties

```
make test-tla-properties
```

With TLA+ (and using `tlc`) we can prove desirable properties such as

> *if the bell is ringing, it will eventually become silent*

or

> *if at the turn of the motor control the doors is open, then eventually the microwave will stop cooking.*

However, since TLA+ is a subset of LTL that does not have `X` (next) and `U` (until), it seems difficult to say that there is a bound for the *eventually*. However, with NuSMV the two types of real-time properties can be expressed, and we can check them. The target

```
make test-realtime-properties
```

verifies as true the LTL property

```
G (G [0,9] Sensor.door_Open -> F [0,19] !Effector.motor_on)
```

This establishes that:

> *globally (in all future states S_0), if the door is open for at least nine consecutive states, then in no more than 19 states, for all states, cooking will stop.*

Thus, we ha proven a real-time property of the upper bound type.

To prove a property of the lower-bound type the earlier target also includes the LTL property

```
G ((Bell_Control.At_RINGING &  X Bell_Control.At_SILENT)
   ->  H [0,29] Bell_Control.At_RINGING)
```

This establishes that

In every state S_0 where the bell is ringing and the next state the bell is silent, then in the previous 29 states, the bell was ringing.

However, a property that speak of the machine and not the effector may seem unsatisfactory. So the same target also includes the following property.

```
X ( X ( X ( X ( X (
     G ((Effector.sound_on &  X !Effector.sound_on)
     ->  H [0,29] Effector.sound_on))))))
```

This property says the following.

Provided the software has run for 5 steps, then in any further state S_0 such that the sound is on, but in the next state the sound is off, it is the case that the sound has been on for the previous 29 steps.

We need the software to run for 5 steps, because the software may start with the sound on, and the software turns the sound off as soon as possible in this case. If we take away the next (`X`) operator, it is not always true that the sound is always on for a long time.

There is a slight challenge with real-time properties that requires attention. Properties may be true by vacuity. That is, the property may be true because there is never a state S_0 with the sound on and the next state with the sound of. So we must also show that it is possible to reach states with the sound on. Alternatively, we show that not all paths never have the sound off. The target

```
test-false-ctl-properties
```

uses NuSMV to verify CTL properties in the file `ctl-false-properties.nusmv`. Among these properties we have the following.

```
AG !Effector.sound_on
```

Not only the model checker indicates the property is false, but provides a specific path to a state where the sound is on. Since the property appears in the fifth position in the file `ctl-false-properties.nusmv`, the path is saved in `logProp4.txt`.

The following property is more explicit about the existence of a path where the sound is on, and later, eventually, off.

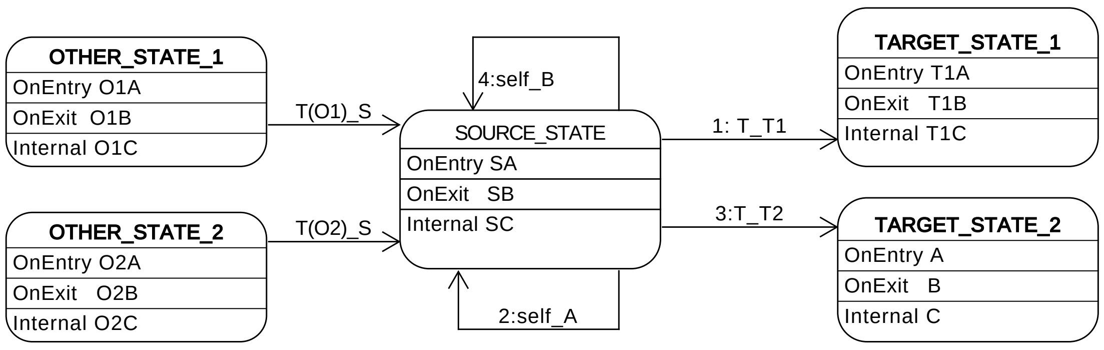
```
AG !(Effector.sound_on & EX !Effector.sound_on)
```

Since the model checker indicates the property is false, it shows that the real-time property above is not true by vacuity, but effectively true. Also, we can check the trace produced in the output.

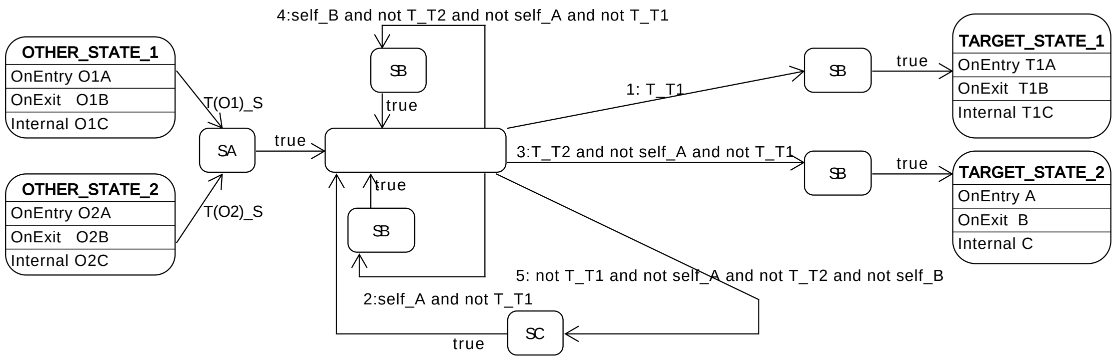
3.4 The semantics of LLFSMs

Translating arrangements of LLFSMs to NuSMV or TLA+ is providing a semantics for LLFSMs. Historically, LLFSM's states were presented as having sections [1, 2]. The idea behind including sections inside states was to facilitate migration of those familiar with state charts in UML to LLFSMs [3]. Although LLFSMs could also be introduced to those familiar with UML's state charts as analogous, but only *guards* are used to label transition, this analogy is an over-simplification. Even in this case, the semantics of UML with only guards is controversial: for instance, when transitions in UML's state charts are labelled by guards alone, some suggest [4] that such transitions fire as soon as they become true (which raises all issues of concurrent execution).

Thus, the first tools that we developed for translating arrangements of LLFSMs to nusmv used ATL [3, 5] and considered states with sections (OnEntry, OnExit, and Internal). The intuition is that the actions



(a) A `SOURCE_STATE` with sections.



(b) Semantics of a "`SOURCE_STATE` with sections" with states that have no sections.

Figure 3.8: The semantics of sections in state should be understood as semantic sugar for a complex group of states and transitions.

in the OnEntry are executed upon arrival to the state (when the state is the target of a transition that fired), those actions in the OnExit are executed when the state is the source of a transition that fires. While those actions in the Internal section happen when the turn arrives to the machine and evaluating all transitions out of the current state does not result in any transition firing. The term *ringlet* was used [1] for defining what happens when the round-robin turn returns to a machine and which actions of the current state (or target state) execute at this turn. However, we have noticed that this description results in a series of semantic variants or alternatives that complicate the understanding translation [6]. For instance, a machine must remember if it has executed or not the OnEntry of its current state. Which typically means that the OnEntry actions are not in the ringlet (turn) when a transition fires. Many more variants appear when we consider self-transitions. One possibility is for this to execute the OnEntry, the OnExit, both or none, but always skip the Internal section.

Thus, we would argue that is better to formalize the semantics of LLFSMs by specifying it for states without sections. Thus, in this case a ringlet is much simpler and uniform. All transitions with the current state as source state (including self transitions) are evaluated in order (they are always in sequence). The first one to fire causes the execution of the actions in the target state of such transition and the current state is updated.

If sections are to be used, they are simply syntactic sugar for a more complex group of states as illustrated by Figure 3.8.

ToDo: Include a folder of examples that illustrate some of this semantic issues.

Moreover, the execution of a ringlet is atomic. That is, no other LLFSM in the arrangement has the turn, so the firing of the transition and the execution of the OnEntry in the target state happens without any changes of any action in any other LLFSMs in the arrangement. If an arrangement of LLFSMs is to model concurrency in a multi-threaded system or a distributed system, and thus some actions may be executed in more than one possible order, then this is simply a non-deterministic update of the scheduler allocating the turn to the LLFSMs in the arrangement.

Since a ringlet consists of the sequential evaluation of the Boolean expression leaving the current state, it is important to consider the context of this evaluation. Since the meta-model does not allow assignment to variables as part of a Boolean expression, evaluating a transition can not have side-effects in the evaluation of the next. However, if a sensor variable appears in more than one position in the sequence of Boolean expressions out of a source state we consider unsuitable a semantics where the environment could provide a different value. Therefore, the semantics of this situation is as follows. All sensor variables in the sequence of Boolean expressions out of a source state are given values (are read) unto a snapshot (a context), and under this context, all the Boolean expressions are evaluated.

This is consistent with a "snapshot semantics" that was given earlier [5] to LLFSMs where execution of a ringlet consisted of a preliminary step to any action execution copying all values of variables (whether shared [whiteboard], read [sensor], written [actuator] or local) to be used while running the ringlet. At completion of all the actions of the ringlet, the context (which now may contain updates) is copied back into the arrangement to reflect such updates. The spirit of this semantics is that a system should not depend on the possibility of the status of a sensor changing during the infinitesimal duration of the execution of a ringlet. This would either let trough systems with a fault that occurs with extremely low probability, but would be intrinsically faulty, or complicate enormously the formal verification when the snapshot semantics solves this.

Note that the semantics proposed here (that a context of all variables to be involved in the actions of running the ringlet be configured and used during the execution of the ringlet) implies either a construction at compilation time of and ADT dictionary of variables and associated values, or the construction at runtime (if LLFSMs are interpreted) with the corresponding run-time cost of dynamic memory allocation.

ToDo:Check that translations to C and MIPS do this for evaluation transitions and the states, the code progresses for transitions only in C (but not to states)

Chapter 4

Creating a model using ECLIPSE and EMF

4.1 The necessary SDKs

Since our meta-model for Logic-Labelled Finite-State Machines (LLFSMs) is represented in the Eclipse Modeling Framework (EMF), it is possible to use

1. EMF - Eclipse Modeling Framework SDK, and
2. Ecore Diagram Editor (SDK)

to create instances of arrangements on LLFSMs.

The following procedure shall set you up. It is based in the following tutorial

www.vogella.com/tutorials/EclipseEMF/article.html

1. Install Eclipse. We started using a version from 2019, and at the moment we are using 2022-03. Follow standard instructions for this tasks.
2. Install in your Eclipse the two pieces mentioned before.
   (a) EMF - Eclipse Modeling Framework SDK, and
   (b) Ecore Diagram Editor (SDK)

   This may require that you add the path to find them. In Eclipse, this means going to the `Help` tab (usually rightmost) in the top bar, and when the menu drops, chose `Install New Software`. Then a new panel will show and then, select `Add`. A further panel shows, and in location type

   `http://download.eclipse.org/releases/mars`

   Alternatively, the two SDKs above may also be in

   `https://download.eclipse.org/releases/latest`

   which you may find pressing the down-button in the field `Work with:`. Once the choosing panel with `Name` and `Version` has some content, you open the inside of `Modeling` and select the two SDK's. Click all the `Next` and `Finish` buttons to install which may require you to relaunch Eclipse.

There are two ways you can get the standard editor of the ecore modeling frameowrk for building your own models as ẋmi files. One way is use the already built Ecore Modelling Project (see Section 4.2) with Eclipse (once the SDKs are installed). The second way means building the Ecore Modelling Project (see Section 4.3). Both options needs the above SDKs installed and the file with the meta-model, namely the file `newGITmetamodelwithtypes.ecore`.

To find this file, you will find that on executing the Docker image, your current directory is

`/opt/Examples`

You will see that

```
ls ..
```

will show you that one level above you have the file

```
EclipseProjectWithMetaModelForLLFSMs.tar
```

Take this file to the shared directory with the host.

```
mv ../EclipseProjectWithMetaModelForLLFSMs.tar ../results
```

Now, you can use the host to unpack this file:

```
tar -xvf EclipseProjectWithMetaModelForLLFSMs.tar
```

in the `results` directory of the host and see the file structure of Figure 4.1 and where the file

```
newGITmetamodelwithtypes.ecore
```

is.

4.2 The supplied Ecore Modelling Project

The released Ecore Modelling Project provides the fewest number of files to release the meta-model of the LLFSMs, and start it as a project inside a workspace for Elipse. In particular, the file `newGITmetamodelwithtype` is not provided (see Section 4.3 from how to produce it).

After moving the tar file and unpacking, start Eclipse with the workspace pointing to inside the shared directory `results` and in particular to the directory `EclipseProjectWithMetaModelForLLFSMs`. Currently the project was created with Eclipse 2022-06. Older version of Eclipse will complain. See Section 4.3 to build the project with an older version of Eclipse.

Once you do this a ".medata" directory will be created in the workspace. Never commit from here (too large and ECLIPSE workspaces are not supposed to be on version control, only projects).

In Eclipse, use the "`FILE`" tab for a menu to drop down, and then select "Open project from file system".

Click the "`Directory`" button, and then select and open the folder `newGITmetamodelwithtypes`. Click "`Finish`" to close the dialog box.

In the package explorer panel of Eclipse (left most), you should see the project. You can then inspect the content of the project, find the `model` subdirectory, and inside model you can find the meta-model: the file `newGITmetamodelwithtypes.ecore`.

You can right-click this file and open it with `Sample Ecore Model Editor`; once opened, the editor panel of Eclipse shows the content. You can start opening the hierarchy until you find the class `Arrangement`.

Right click in the class `Arrangement` to have a drop menu and select `Create Dynamic Instance`. You are on your way to edit a LLFSM arrangement in a file of type `.xmi` with the name of your choice.

4.3 Building the Ecore Modelling Project

1. Make a new `Ecore Modelling Project`. You only have the Meta-Model in the file `newGITmetamodelwitht`
   In the earlier section you skip this step because you use the released Ecore Modelling Project. This sections shows you how to build the Ecore Modelling Project. If you want to see screen shots of what is to come, we recommend the part titled `Create an Ecore Modeling Project` from the following link:

   `https://wiki.eclipse.org/Sirius/Tutorials/DomainModelTutorial`

   Simply, we now go to `File` in the top tabs (usually leftmost), the menu drops, and we chose `New`, followed by `Other`. We use the wizard to open `Ecore Modeling Framework` and select `Ecore Modeling Project`. **It is important that you name your project `newGITmetamodelwithtypes`.** This is because once you click `Finish`, a file named `newGITmetamodelwithtypes.ecore` and others will be created, one or two levels down from a directory named `newGITmetamodelwithtypes` in

```
newGITmetamodelwithtypes
└─model
   ├─newGITmetamodelwithtypes.ecore
   └─newGITmetamodelwithtypes.genmodel
```

Figure 4.1: The directory structure of the `Ecore Modeling Project`.

```
newGITmetamodelwithtypes
├─Arrangement
└─StateMachine
```

Figure 4.2: meta-model tree for arrangements of LLFSMs.

your chosen Eclipse workspace. Use the right-click in Eclipse to obtain a menu to delete the files `newGITmetamodelwithtypes.ecore` and `newGITmetamodelwithtypes.genmodel`. Figure 4.1 shows you the directory structure of the `Ecore Modeling Project` so you can locate these files. Exit Eclipse and place our provided `newGITmetamodelwithtypes.ecore` where the deleted one was (you may want to use directory exploration tools outside Eclipse to find the path of the one to delete so you know where it is being replaced).

2. Regenerate `newGITmetamodelwithtypes.genmodel`. Since we deleted this one, we need to create it from the meta-model in `newGITmetamodelwithtypes.ecore`. So right click on `newGITmetamodelwithtypes` and chose `New` followed by `Other` and then under the `Eclipse Modeling Framework` select `EMF Generator Model` and push the button `Next`. Accept the defaults by clicking `Next` again. About two screen down you will see that the `ecore Import` is our `newGITmetamodelwithtypes.ecore`. Just chose `Load` and `Next`. Then in the alst panel click `Finish`. You should have `newGITmetamodelwithtypes.g` restored. To check that it worked, but is not absolutely necessary, open it with the default. Then in its tab in the editor you will see the root of the classes of the metamodel. Right-click and select `Generate All`. This generates `java` classes for out metamodel (which we included to build our translators).

3. **Build an arrangement of LLFSMs**. Go back to the tab of `newGITmetamodelwithtypes.ecore`. Open the tree of the meta-model. Figure 4.2 shows the tree of the meta-model. You can build parts of LLFSMs, but if you want to translate them, run them and/or verify them as demonstrated in earlier sections of this document, they must be in an `Arrangement` (even if it is only one machine. This, right-click in `Arrangement` and half-way down the appearing menu you will see `Create Dynamic Instance`. Select this and you will see you are about to edit a file of type `.xmi` as those we provided in the examples. By default it comes as `Arrangement.xmi` but you can chose a different name.

Chapter 5

Limitations

5.1 Data types

The data types implemented so far are `Integer`, `Boolean` and `Set of Integer`. Even for this types, not all aspects of these types may be implemented. Particularly some `Set of Integer` operations may not be implemented. Also for `Set of Integer`, some variables may not be implemented. For instance, effector variables of type `Set of Integer` are still missing.

It is desirable to have arrays and sequences. That is, the type `array` of `Integer` would be desirable to make more succinct LLFSMs for some of the schedulers, and handle all thread uniformly. Similarly, variables of type `sequence` of `Integer` would be useful to represent protocols like Paxos. Arrays of integers are functions with domain $\{0, 1, \ldots, n\}$ and range the integers, where the n is fixed. However, "sequences of integers" are functions with domain $\{0, 1, \ldots, n\}$ and range the integers, but the n is not fixed, so operations that append an element exist and enlarge the sequence by one. Sometimes "sequences of integers" are also called "lists of integers".

5.2 Translations

Not all models translate to TLA+. If an example does not mention the translation to TLA+, it is probably that there are elements not implemented.

Bibliography

[1] V. Estivill-Castro, D. A. Rosenblueth, Model checking of transition-labeled finite-state machines, in: T. Kim, A. H., H. Kim, H. Kang, K. J. Kim, K. Akingbehin, K. B. H. (Eds.), Software Engineering, Business Continuity, and Education - International Conferences ASEA, Vol. 257 of Communications in Computer and Information Science, Springer, 2011, pp. 61–73. `doi:10.1007/978-3-642-27207-3\_8`.

[2] V. Estivill-Castro, R. Hexel, C. Lusty, High performance relaying of c++11 objects across processes and logic-labeled finite-state machines, in: D. Brugali, J. F. Broenink, T. Kroeger, B. A. MacDonald (Eds.), Simulation, Modeling, and Programming for Autonomous Robots, Springer International Publishing, Cham, 2014, pp. 182–194.

[3] M. Carrillo, V. Estivill-Castro, D. A. Rosenblueth, Model-to-model transformations for efficient time-domain verification of concurrent models by NuSMV modules, in: S. Hammoudi, L. Ferreira Pires, B. Selic (Eds.), Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, SCITEPRESS, 2020, pp. 287–298. `doi:10.5220/0008910202870298`.

[4] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach, 2nd Edition, IEEE Press, Wiley, Hoboken, NJ, 2015. `doi:10.1002/9781118989166`.

[5] M. Carrillo, V. Estivill-Castro, D. A. Rosenblueth, Verification and simulation of time-domain properties for models of behaviour, in: S. Hammoudi, L. Ferreira Pires, B. Selic (Eds.), Model-Driven Engineering and Software Development Revised Selected Papers, Vol. 1361 of Communications in Computer and Information Science, Springer, 2020, pp. 225–249. `doi:10.1007/978-3-030-67445-8\_10`.

[6] V. Estivill-Castro, R. Hexel, Resolving the asymmetry of on-exit versus on-entry in executable models of behaviour, in: S. Hammoudi, L. Ferreira Pires, B. Selic (Eds.), Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD, SciTePress, 2019, pp. 49–61. `doi:10.5220/0007323300490061`.