Instruction Manual: Computing optimal schedules for SDF graphs on limited resources using STARS (<u>S</u>cheduling and <u>T</u>emporal <u>A</u>nalysis on limited <u>R</u>esources for <u>S</u>DF graphs) Tool-Chain

Introduction

STARS tool-chain consists of following steps.

1. Utilising SDF3 for generating and analysing SDF graphs.
2. Automatically transforming SDF3 models to UPPAAL models.
3. Utilising UPPAAL for producing traces for SDF graphs on limited resources .
4. Using Python to parse traces produced by UPPAAL to generate Excel sheets, representing the optimal schedules.

This document has been prepared to provide a detailed guideline on using SDF3 and UPPAAL for the aforementioned steps. Thus, this document serves following main purposes:

1. To provide a guide on how to use the SDF3 tool to analyse SDF graphs for various properties, in the context of STARS tool-chain.
2. To provide a guide on how to generate UPPAAL models from an SDF3 model, combined with an heterogeneous hardware platform. The hardware platform defines the processor types, number of processors of each type, and the mapping of the SDF actors to processors.
3. To provide a guide on how to use UPPAAL to compute the throughput-optimal schedule of the SDF3 model on a given number of processors.

Global setup of STARS tool-chain

Input:

1. SDF graph *G* in SDF3 input format with following additional information.
    a. Processor types.
    b. Mapping of actors to processor types.
2. Hardware Platform with following aspects
    a. Processor types.
    b. Number of processors of the same processor type represented by “n_{pt}”.
    c. Total number of processors represented by “n”.
    d. Mapping of actors to processor types.

Output: Excel sheet containing the throughput-optimal schedule on “n” number of processors.

Method: STARS tool-chain implements methodology from the work in [1], i.e., transforming an SDF graph in SDF3 input format to Timed Automata (TA) models. The TA models are analysed using UPPAAL tool, for temporal guarantees and generating optimal schedules on the given number of processors.

Installation

1. Install SDF3 from the following link.
   http://www.es.ele.tue.nl/sdf3/download/
2. Download Eclipse from the following link.
   https://eclipse.org/downloads/
3. Download UPPAAL from the following link.
   http://www.uppaal.org
4. Add the following folder in the “workspace” of Eclipse.
   https://github.com/utwente-fmt/STARS/tree/master/epsilon/sense
5. Download the Python script “parse_results_delay.py” from the following link.
   https://github.com/utwente-fmt/STARS/tree/master/PythonScript

Step-by-Step Procedure

Following are the step-by-step instructions for using STARS tool-chain. We use “SDF3file.xml” to represent the name of the SDF3 file. We refer to the execution of an actor as an (actor) firing.

1. Checking Syntactic Well-formedness of the SDF graph

The first step is to check the syntactic well-formedness of an SDF graph *G* in the SDF3 format, using the following commands.

a. Check if the SDF3 file is deadlock free.

```
sdf3analysis-sdf -- SDF3file.xml --algo deadlock
```

b. Check if the SDF3 file is consistent.

```
sdf3analysis-sdf -- SDF3file.xml --algo consistency
```

2. Computing the Repetition Vector of the SDF graph

The second step is to compute the repetition vector of an SDF graph *G* in the SDF3 format, using the following command.

```
sdf3analysis-sdf -- SDF3file.xml --algo repetition_vector
```

The repetition vector looks like, a list of the actor names and positive integers representing the corresponding repetition vector entries. For each actor “a”, we represent its repetition vector entry by rep(a).

If an SDF graph is not consistent in Step 1b, the repetition vector shows 0s against each actor name.

3. *Computing the Self-timed throughput of the SDF graph*

The third step is to compute the throughput of the self-timed execution of an SDF graph *G* in SDF3 input format, using the following command.

```
sdf3analysis-sdf -- SDF3file.xml --algo throughput
```

The above command shows a decimal number representing the throughput, and the time needed (ms) to compute it.

If an SDF graph is not consistent in Step 1b, the command above shows 0 as a throughput.

4. *Printing the self-timed execution of the SDF graph*

The last step in the SDF3 environment is to generate the self-timed schedule of an SDF graph *G*, using the following commands.

```
sdf3print-sdf      --graph      SDF3file.xml      --format
selfTimedSchedule --output SDF3file.cpp
```

```
g++ SDF3file.cpp -o SDF3file
```

```
./ SDF3file  'time' --> SDF3file.txt
```

Here, “time” represents any numerical value and the self-timed schedule will be printed up to that time stamp.

The printed self-timed schedule shows the time stamps and the actors executed at those time stamps. We have to process this schedule manually as follows.

At some time stamp “j”, we observe that all actors have fired according to the repetition vector, with respect to some earlier time stamp “i”. This signifies that the SDF graph has entered the periodic phase. The difference between “i” and “j” represents the length of the periodic phase. The throughput of the SDF graph *G* is equal to the reciprocal of the length of the periodic phase, which should be equal to the throughput computed in Step 3.

After processing the printed self-timed schedule as explained above, we also determine how often each actor “a” fires in the transient and periodic phase, represented by the positive integers “x(a)” and “y(a)” respectively. As each actor “a” fires according to its repetition vector in the periodic phase, y(a)= rep(a).

Using the self-timed schedule printed in Step 4, we manually determine the number of processors required for self-timed execution, represented by “req_proc”. This is done by checking the number of simultaneous executions at each time step. The largest number of simultaneous executions at any time stamp during the entire execution gives us req_proc.

SDF3 to UPPAAL Transformation

5. Add your SDF3 file to the Folder “models” in the Eclipse workspace.
6. From the “sense” directory, run the following command.

```
ant      -lib      lib      -f      ant/sdf2uppaal.xml      -
Dinput=../models/SDF3file.xml
```

7. The above command automatically generates the UPPAAL model with the name "SDF3file.uppaal.xml".

Differences between SDF3 and UPPAAL models

SDF3 only includes the information about processor types, but does not hint about the number of processors of each type. Furthermore, an actor can be mapped to maximum one processor type. On the other hand, our methodology [1] requires the information about the number of processors of each type. Furthermore, we consider that an actor can be mapped to multiple processor types. To account for the missing information in SDF3, we generate UPPAAL model containing separate templates for the SDF graph *G*, and each processor type. The processor type templates are parametrised with the actor names, representing which actors can be mapped to that processor type. For each processor in the hardware platform, an instance of its processor type's template represented by "processor_type_template" is created. By default, STARS tool-chain creates two instances of each processor type. Each processor has a unique identifier represented by "processor_id". Same is for each actor represented by "actor_id". There is an integer variable in "Declarations" of UPPAAL model to count the number of processors and actors, represented by "N" and "M" respectively.

a. Depending on the number of processors of each type, i.e., "n_{pt}", create or remove the instances, and update the variable "N". The instances in UPPAAL can be written in "System declarations" by writing the following lines.

   const int *id_of_the_processor_instance* = processor_id;

   name_of_the_processor_instance =
   processor_type_template(id_of_the_processor_instance, actor_ids);

   And then add *name_of_the_processor_instance* after "System " keyword in "System declarations".

b. Similarly, in the UPPAAL model by default, an actor can be mapped only to the processor type that is specified in the SDF3 model. If an actor can be mapped to more than one processor type, the user has to add this information manually in the UPPAAL model. This is done by adding the actor_id as a parameter to each *name_of_the_processor_instance* onto which the actor is mapped in "System declarations", as follows.

   name_of_the_processor_instance =
   processor_type_template(id_of_the_processor_instance, actor_ids);

Using UPPAAL for throughput-optimal scheduling on a given number of processors "n"

8. We calculate $k(a) = \left\lceil \frac{x(a)+y(a)}{y(a)} \right\rceil$ for each actor "a". Let us suppose that $l = max(k(a))$ for all actors "a".
9. Let us suppose that the given number of processors "n" is equal to *req_proc*. We have a variable in the UPPAAL model named "counter_a" for each actor "a", to count how many times each actor has fired. We run the following query in UPPAAL, and ask for the fastest trace.

   E<>(counter_a==$l.rep(a)$&& counter_b==$l.rep(b)$&& ...)

the option to obtain the fastest trace in GUI of UPPAAL can be found in "Diagnostic Trace" in Options dropdown menu. Following command can be run to ask for the fastest trace using command line.

```
verifyta –C –t2 SDF3file.uppaal.xml SDF3file.uppaal.q 2>
SDF3file.uppaal-diagnostictrace.txt
```

In the above command, "-C" selects "Difference Bound Matrices (DBM)" as the state-space representation. If we don't use "-C", UPPAAL chooses "Minimal Constraint Representation" by default. DBMs are often fast, but for models with many clocks they require a lot of memory. On the other hand, Minimal Constraint Representation reduces significant memory consumption at the expense of speed. Furthermore, "-t2 " generates the fastest trace. Moreover, "SDF3file.uppaal.q" stores the earlier mentioned query, and "2>" denotes the writing of the trace to standard error (stderr). The generated trace in Step 9 is stored in "SDF3file.uppaal-diagnostictrace.txt".

Generating schedules in Excel from UPPAAL trace

SDF3file.uppaal-diagnostictrace.txt has the following format.

- "State: …" showing the current state, and
- "Transitions" showing the transitions taken from a certain state. "States" tags are always followed by the "Transitions" tags, representing the source state of the transitions. Similarly, "Transitions" tags are always followed by the "States" tags, representing the target states of the transition.

  If the automaton stays in the current state for x time units., it is represented by "Delay: x". "Delay" tags appear in between, two "State" tags representing before and after the incurring of the delay respectively.

In the "State" tag, we can see if a processor is currently idle or executing a certain actor. If the processor is idle and a certain actor is going to be mapped to it, we can see it in the "Transition" tag via the action label on the transition. The resulting "State" tag shows that the processor is occupied by that actor. Now the processor will stay in current "State" until the execution time of the actor ends. This is shown via the "Delay" tag. Afterwards, there will be another “Transition” tag, denoting the finishing of the actor firing, and the resulting “State” tag shows that the processor is idle now.

10. Run the Python script using the following command to translate the trace generated by UPPAAL to the schedule in Excel sheet.

```
cat SDF3file.uppaal-diagnostictrace.txt | python parse_results_delay.py > SDF3file.uppaal-diagnostictrace.csv
```

This script reads the “States” and “Delay” tags, by which it generates the comma-separated value (csv) file with the name SDF3file.uppaal-diagnostictrace.csv. The csv file can be changed to the Excel sheet, by opening Microsoft Excel, and selecting “”From Text” from the “Data” drop-down menu. Afterwards, select the csv file. If asked for “Choose the file type that best describes your data:”, select “Delimited”, and press “Next”. Then, choose “Other”, enter “;” in the white box in front of it, and deselect all other options. Then select “General” and press “Finish”. When prompted for “Where do you want to put the data?”, keep the default value “=A1”.

The Excel sheet contains the schedule of which actor is executed on which processor (columns in Excel sheet) at what time (rows in Excel sheet). This schedule is same as the self-timed schedule derived in Step 4. Each actor “a” fires “x(a)” times in the transient phase, “y(a)” times in the periodic phase, and $l.rep(a) - y(a) - x(a)$ times in the final phase. Please note that the final phase takes the execution back to initial state, thus finishing another period. Hence, we have an intermediate period with the shortest accumulated time (periodic phase) within the longer period (from the initial state back to it). The length of the periodic phase (with the shortest accumulated time) represents the maximum throughput. We ignore the final phase in our method.

Throughput-optimal scheduling for fewer number of processors

11. If the given number of processors n is less than req_proc, we start with the query E<>(counter_a==$l.rep(a)$&& counter_b==$l.rep(b)$&& ...) and ask for the fastest trace.

The generated trace is manually interpreted as a schedule, following Step 10.

If the number of available processors is less than req_proc, it could happen that in the fastest trace, the order of actor firings is different in the transient phase, than the self-timed execution. As a result, the SDF graph may enters the intermediate periodic phase (with the shortest accumulated time) later than the self-timed execution. In this case, the value of l may not be sufficient to find this intermediate periodic phase. Thus, the result of the above query is the trace without any transient and intermediate periodic phases, and the graph only finishes the longer periodic phase directly, i.e., for each actor “a”, $x(a) = 0$, and $y(a) = l.rep(a)$.

We gradually start adding l by 1, until we find the intermediate periodic phase with the shorter accumulated time, if any.

Bibliography

[1] W. Ahmad, R. de Groote, P. K. F. Hölzenspies, M. Stoelinga, and J. van de Pol, Resource-Constrained Optimal Scheduling of Synchronous Dataflow Graphs via Timed Automata," in *14th International Conference on Application of Concurrency to System Design (ACSD)*, Tunisia, 2014.