An Introduction to the Pthales Domain of Ptolemy II

Remi Barrere
Eric Lenormand
Dai Bui
Edward A. Lee
Christopher Shaver
Stavros Tripakis

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-32
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-32.html

April 26, 2011

An Introduction to the Pthales Domain of Ptolemy II[1]

Rémi Barrère, Eric Lenormand
Thales, France

Dai Bui, Edward Lee, Chris Shaver, Stavros Tripakis
University of California, Berkeley

April 26, 2011

[1]This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

Introduction

This is an introduction to *Pthales*, one of the *domains* implemented in Ptolemy II. For an introduction to Ptolemy II, including documentation and information on the general architecture and concepts of Ptolemy II, see `http://ptolemy.eecs.berkeley.edu/`.

Pthales is still undergoing development, and this document is expected to evolve. Pthales is defined and implemented by the authors of this document. This project is done as part of Thales' industrial membership as Affiliate of the University of California's, Berkeley, Center for Hybrid and Embedded Software Systems (CHESS) – `http://chess.eecs.berkeley.edu/`.

The semantics and behaviour of Pthales are derived from SpearDE, a co-design environment developed by Thales, partly based on Ptolemy Classic. SpearDE uses a *multidimensional synchronous dataflow* (MDSDF) model of computation (MoC) based on the ArrayOL language [4, 5, 1, 2]. MDSDF can be seen as an extension of synchronous (or static) dataflow (SDF) [7, 12]. In SDF, actors communicate through FIFO queues that carry streams of mostly scalar tokens. In MDSDF each token is conceptually a multidimensional array. MDSDF has appropriate uses in advanced signal processing applications such as radars.

The main motivations for building Pthales have been the following:

- First, to extend the SpearDE MoC. The MoC realized by SpearDE supports multidimensional signal processing algorithms with relatively static structure. This means two things. First, the structure of the algorithm, that is, the number and type of actors and their interconnections, do not change during execution. Second, the parameters of the actors such as *repetitions* (the number an actor is fired to produce or consume a single multidimensional array), *patterns* (the part of the array consumed at each firing), *tilings* (how patterns are "moved" from one firing to the next), and so on, do not change during execution. This is a limitation since many applications require a change in these two elements. One of the goals of the Pthales project is to define dynamic extensions of the model which do not suffer from the above limitations.

- Second, to integrate the Pthales domain within the rest of the Ptolemy II domains. This means that one should be able to connect Pthales models to other models, or embed Pthales models within other models, and vice-versa. Other models could include modal models [10, 8], process networks [6], or even continuous-time models [11, 9].

  The motivation for such an integration comes from Thales' broader modeling needs. SpearDE is an environment used in a number of applications at Thales that involve compute-intensive signal-processing. An example is radar applications. SpearDE is targeted at the compute-intensive signal-processing part of these applications, and provides advanced parallelization techniques to compile SpearDE programs onto parallel architectures. However, signal processing is only one part of an application such as radar. Other important parts include the control logic of the radar, i.e., its decision-making component. We envisage that Ptolemy II can be used to model these parts as well as the signal-processing parts. The fact that Ptolemy II is able to integrate heterogeneous MoCs into a single model facilitates this. We also envisage that Ptolemy II can be used to integrate aspects beyond the system, for instance, by modeling also the environment within which the system operates. In a radar system, this environment could for example include a set of moving targets, the physical medium (air, water, ...) in which system and targets operate, and so on.

This document has two parts. Chapter 1 is a tutorial of Pthales, aimed at users who wish to build Pthales models. Chapter 2 provides a formal analysis of Pthales semantics.

Chapter 1

Pthales Tutorial

The aim of this chapter is to provide a tutorial to the Pthales domain, for potential Ptolemy II users that wish to build Pthales models. We begin by describing the "static" Pthales domain in Section 1.1. In Section 1.2 we describe the dynamic extensions of Pthales currently under study.

Throughout this chapter we make reference to Ptolemy II models that can be found in the Ptolemy II source repository under the directory `ptolemy/domains/pthales/`. When we refer to a model such as `demo/Illustrative/Illustrative.xml`, this corresponds to the file `ptolemy/domains/pthales/demo/Illustrative/Illustrative.xml`.

1.1 Static Pthales

We first discuss the subclass of "static" Pthales models. These models are static in the sense that their structure and parameters do not change over time (during execution of the model).

1.1.1 A series of "hello world" examples

A series of simple introductory examples can be found under `demo/Illustrative/` as models `HelloWorld0.xml`, `HelloWorld1.xml`, and so on. These examples should be self-explanatory (see comments included in the models as annotations) so we will not discuss them further here. Instead, we discuss a slightly richer model at length:

1.1.2 A simple example: Illustrative

Consider the model `demo/Illustrative/Illustrative.xml`, the top level of which is shown in Figure 1.1. This is a Pthales model with three actors, Source, Sink and Sink2. All three actors are instances of PthalesCompositeActor.

In a nutshell, this model does the following. First, the Source actor produces a 2-dimensional array of size 8×10. In Pthales, dimensions are named, or labeled, so more precisely, the size of the array is `[x=8,y=10]`, where "x" and "y" are the labels of the two dimensions in this case. The contents of this array are the integers from 1 to 80 (at this point, never mind why this is so). The array is depicted below, together with the indices in the x- and y-dimensions:

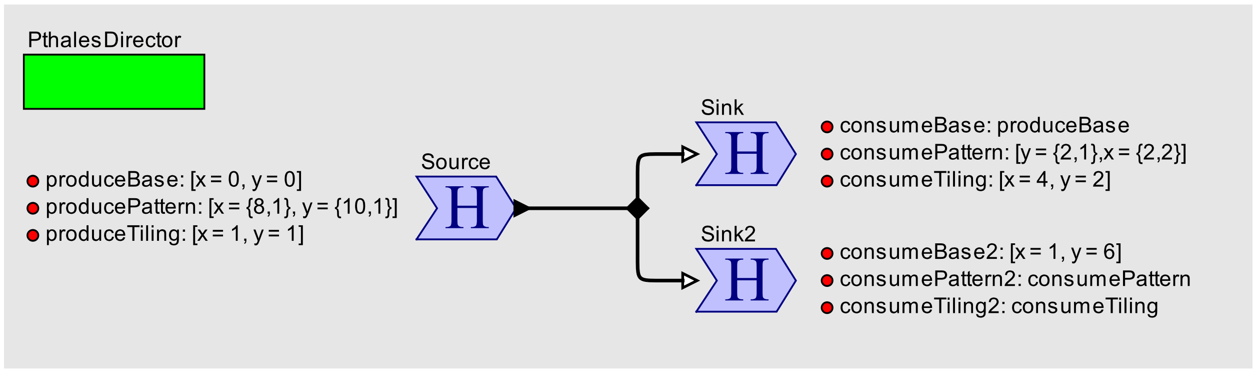

PthalesDirector
Source
produceBase: [x = 0, y = 0]
producePattern: [x = {8,1}, y = {10,1}]
produceTiling: [x = 1, y = 1]
Sink
consumeBase: produceBase
consumePattern: [y = {2,1},x = {2,2}]
consumeTiling: [x = 4, y = 2]
Sink2
consumeBase2: [x = 1, y = 6]
consumePattern2: consumePattern
consumeTiling2: consumeTiling

Figure 1.1: Top level of `demo/Illustrative/Illustrative.xml`.

		x							
		0	1	2	3	4	5	6	7
	0	1	2	3	4	5	6	7	8
	1	9	10	11	12	13	14	15	16
	2	17	18	19	20	21	22	23	24
	3	25	26	27	28	29	30	31	32
	4	33	34	35	36	37	38	39	40
y	5	41	42	43	44	45	46	47	48
	6	49	50	51	52	53	54	55	56
	7	57	58	59	60	61	62	63	64
	8	65	66	67	68	69	70	71	72
	9	73	74	75	76	77	78	79	80

After the Source actor produces the array, the two Sink actors each consume a distinct copy of this array. In this model, the two Sinks contain internally Display actors which display sequences of elements on the array as we will discuss later.

Currently, feedback loops are not supported in Pthales models. This means that all valid Pthales dataflow graphs are acyclic. Because of this, there is a well-defined order (e.g., produced by a topological sort) according to which the PthalesDirector can fire all the actors in the model, from up-stream to down-stream. Source actors first produce arrays, these are consumed by down-stream actors which can themselves produce more arrays, and so on, until the last actors (Sinks). We refer to this as an *iteration* of the model. A model can have one or more iterations.

The Source actor

Every actor in a Pthales model is expected to have a `repetitions` parameter whose value is a n-dimensional vector of positive integers, where n is the dimensions of the arrays that the actor produces or consumes (if the actor has multiple input or output ports, then the dimensions of arrays consumed and produced at these ports must be equal). The `repetitions` parameter of Source is set to `{1,1}` (double-click on the actor to see this). This means that at each iteration, Source is fired a total of $1 \cdot 1 = 1$ times. In particular, Source is fired only once to produce the 8×10 array shown above.

In general, an actor may need to be fired multiple times in an iteration, in order to produce a single output array, or to consume a single input array. For example, actor Sink has its `repetitions` set to `{2,3}` which means that it is fired $2 \cdot 3 = 6$ times at each iteration.

Pthales actors have ports that are instances of PthalesIOPort. These ports have the following parameters:

- `base`: the origin with respect to which other coordinates are computed;

- `pattern`: the "area" of the array that is written to or read from, at each firing of the actor within an iteration;
- `tiling`: how the pattern "moves" or "slides" from one firing to the next, within an iteration (think of a sliding window, but multidimensional);
- `size`: the size of the array produced at output ports (currently optional, as it can be computed as a function of the other parameters, see Chapter 2).

The `base` parameter of the output port of Source is set to `[x=0,y=0]` which explains why the indices of the produced array range from 0 to 7 in the x-dimension and from 0 to 9 in the y-dimension.

The `pattern` parameter of the output port of Source is set to `[x={8,1},y={10,1}]`. This specifies that one pattern (produced at each firing within an iteration) has size 8 in the x-dimension and 10 in the y-dimension. It also specifies that the separation between two adjacent coordinates is 1 in both dimensions. As we shall see below, patterns can have "holes", meaning that the separation can be greater than one. As the case of patterns without holes is frequent, the syntax `[x=8,y=10]` is also allowed for the `pattern` parameter and is equivalent to the one above.

The `tiling` parameter of the output port of Source is irrelevant in this example, since Source only fires once per iteration, therefore, its pattern never has to "slide". The `tiling` parameter is relevant for the Sink actors and will be explained below.

As a result of the above specifications, the 8×10 array produced by Source is computed in a single iteration of the actor, as a single pattern, with the origin set to `[x=0,y=0]`. Internally, Source consists of an SDF graph that produces the stream of (scalar) tokens $1, 2, 3, ..., 80$: open the model and "look inside" the Source actor to see this. The `vectorizationFactor` parameter of the internal SDF Director of Source is set to 80, which means that for each firing of Source, the internal SDF graph is fired 80 times. The output PthalesIOPort converts the stream of 80 tokens to the above array, as specified by the `pattern` parameter of the port. It is interesting to note that the order in which dimensions are specified in the `pattern` parameter matters. Therefore, `[x={8,1},y={10,1}]` is not the same as `[y={10,1},x={8,1}]`. The latter would result in a different array, namely:

		x							
		0	1	2	3	4	5	6	7
	0	1	11	21	31	41	51	61	71
	1	2	12	22	32	42	52	62	72
	2	3	13	23	33	43	53	63	73
	3	4	14	24	34	44	54	64	74
	4	5	15	25	35	45	55	65	75
y	5	6	16	26	36	46	56	66	76
	6	7	17	27	37	47	57	67	77
	7	8	18	28	38	48	58	68	78
	8	9	19	29	39	49	59	69	79
	9	10	20	30	40	50	60	70	80

The Sink actors

We now turn to the consumer actors, Sink and Sink2. As mentioned above, the `repetitions` of Sink is set to `{2,3}`. Thus, Sink is fired $2 \cdot 3 = 6$ times at each iteration. At each of these 6 firings, Sink consumes only a portion of the 8×10 array. This portion is specified by the `pattern` parameter of the input port of Sink, which is set to `[y={2,1},x={2,2}]`. This pattern specifies that a 2×2 sub-array should be extracted from the 8×10 input array, as follows: 2 elements in the y-dimension separated by 1; and 2 elements in the x-dimension separated by 2 (thus, this is an example of a pattern with "holes"). With the origin being `[x=0,y=0]`, the sub-array specified by the above pattern consists of the elements at indices `[x=0,y=0]`, `[x=0,y=1]`, `[x=2,y=0]`, and `[x=2,y=1]`. These elements have values $1, 9, 3$ and 11, respectively (we are considering the array produced using the pattern `[x={8,1},y={10,1}]`, not the one produced using `[y={10,1},x={8,1}]`):

		x	
		0	2
y	0	1	3
	1	9	11

At its first firing Sink will read these four values from the input array and, using its internal Display actor ("look inside" to see this), will print them at the output.

This completes the first out of 6 firings of Sink within the iteration. In the subsequent 5 firings, the pattern that Sink consumes is the same, but the origin changes. It changes according to the `tiling` parameter. In the case of Sink, this parameter is set to `[x=4,y=2]` which means that the origin slides by 4 in the x-dimension and 2 in the y-dimension. In particular, the origin will move across the 6 iterations as follows: `[x=0,y=0]`, `[x=4,y=0]`, `[x=0,y=2]`, `[x=4,y=2]`, `[x=0,y=4]`, and `[x=4,y=4]`. The 6 consecutive patterns consumed by Sink will then consist of the following elements, which are printed out in sequence by the internal Display actor of Sink when the model is run:

- Pattern starting at `[x=0,y=0]`: elements at indices `[x=0,y=0]`, `[x=0,y=1]`, `[x=2,y=0]`, `[x=2,y=1]`, with values $1, 9, 3, 11$, respectively.
- Pattern starting at `[x=4,y=0]`: elements at indices `[x=4,y=0]`, `[x=4,y=1]`, `[x=6,y=0]`, `[x=6,y=1]`, with values $5, 13, 7, 15$, respectively.
- Pattern starting at `[x=0,y=2]`: elements at indices `[x=0,y=2]`, `[x=0,y=3]`, `[x=2,y=2]`, `[x=2,y=3]`, with values $17, 25, 19, 27$, respectively.
- Pattern starting at `[x=4,y=2]`: elements at indices `[x=4,y=2]`, `[x=4,y=3]`, `[x=6,y=2]`, `[x=6,y=3]`, with values $21, 29, 23, 31$, respectively.
- Pattern starting at `[x=0,y=4]`: elements at indices `[x=0,y=4]`, `[x=0,y=5]`, `[x=2,y=4]`, `[x=2,y=5]`, with values $33, 41, 35, 43$, respectively.
- Pattern starting at `[x=4,y=4]`: elements at indices `[x=4,y=4]`, `[x=4,y=5]`, `[x=6,y=4]`, `[x=6,y=5]`, with values $37, 45, 39, 47$, respectively.

It is easier to visualize the sliding of the origin according to the `tiling` parameter in conjunction of the `repetitions` parameter as a nested loop:

```
y := y0;     /* set initial origin for y */
for i=1 to 3 do     /* 3 is the 2nd element of the repetitions vector */
  x := x0;   /* set initial origin for x */
  for j=1 to 2 do   /* 2 is the 1st element of the repetitions vector */
    /* compute sub-array from pattern using (x,y) as origin */
    x := x+4;   /* update x index by 4, as specified in tiling "x=4" */
  end for;
  y := y+2; /* update y index by 2, as specified in tiling "y=2" */
end for;
```

Note that the elements of the repetitions vector specify the bounds on the loops, from the inner-most to the outer-most loop. Also note that the elements of the `tiling` parameter specify which index is updated at each loop, in order, from the inner-most to the outer-most loop. Therefore, as in the case of the `pattern` parameter, the order of elements in the `tiling` parameter matters. For instance, `[x=4,y=2]` is different from `[y=2,x=4]`. The latter results in a different nested loop:

```
x := x0;     /* set initial origin for x */
for i=1 to 3 do     /* 3 is the 2nd element of the repetitions vector */
  y := y0;   /* set initial origin for y */
  for j=1 to 2 do   /* 2 is the 1st element of the repetitions vector */
    /* compute sub-array from pattern using (x,y) as origin */
    y := y+2; /* update y index by 2, as specified in tiling "y=2" */
  end for;
  x := x+4;   /* update x index by 4, as specified in tiling "x=4" */
end for;
```

The operation of Sink2 is similar to that of Sink, with the difference that Sink2 has `repetitions` set to `{2,2}` instead of `{2,3}`, and therefore fires 4 times within an iteration, thus consuming 4 patterns. Also, the `base` of Sink2 is set to `[x=1,y=6]` instead of `[x=0,y=0]`, therefore, the values output by Sink2 are different.

1.1.3 Other examples

A small variation of the model `demo/Illustrative/Illustrative.xml` is the model `demo/Illustrative/Illustrative2.xml`. In this variation, the consumption patterns are modified, resulting in different outputs. Also, the `iterations` parameter of the Pthales director is set to 2, resulting in two iterations (i.e., two 8×10 arrays produced by Source and consumed by the Sinks).

Another simple static Pthales model, that uses a 3-dimensional array, is `demo/HelloWorld/HelloWorld.xml`.

A realistic static Pthales model is `demo/AdaptiveBeamForming/AdaptiveBeamForming.xml`. This model captures an adaptive beam forming radar application.

1.2 Dynamic extensions to Pthales

The MoC realized by SpearDE supports multidimensional signal processing algorithms with relatively static structure. This means two things. First, the structure of the algorithm, that is, the number and type of actors and their interconnections, do not change during execution. Second, the parameters of the actors such as *repetitions* (the number an actor is fired to produce or consume a single multidimensional array), *patterns* (the part of the array consumed at each firing), *tilings* (how patterns are "moved" from one firing to the next), and so on, do not change during execution. This is a limitation since many applications require a change in these two elements. In particular, we can identify two types of applications:

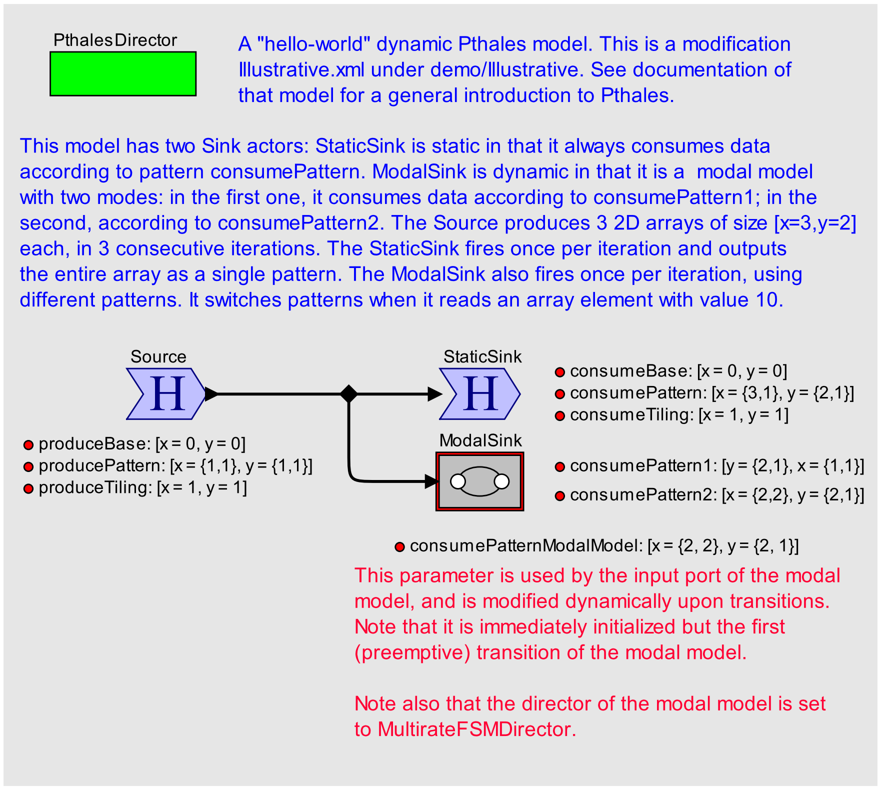
- *Modal* applications where the structure changes dynamically.
- *Parameter-dynamic* applications where the structure does not change, but parameters may change dynamically.

One of the objectives of the Pthales domain is to relax these limitations. For this reason, we describe dynamic extensions to Pthales along the two directions identified above. These extensions are ongoing work.

1.2.1 A Pthales model with a modal model

Modal Models is one of the MoCs implemented in Ptolemy II. It allows arbitrary MoCs to be embedded into a state machine, by refining the states of the state machine. The states of the state machine represent different *modes* of operation of the system (hence the term "modal" models). The transitions between the states of the state machine represent switches between modes. Modal models are themselves composite Ptolemy actors, and can therefore be embedded into other MoCs, to form arbitrary hierarchies. For more details on modal models, see [10, 8].

As a special case, we can use modal models to implement a dynamic Pthales modal with parameter changes. An example is the model `demo/Dynamic/ModalModelPthales.xml`, the top level of which



A "hello-world" dynamic Pthales model. This is a modification Illustrative.xml under demo/Illustrative. See documentation of that model for a general introduction to Pthales.

This model has two Sink actors: StaticSink is static in that it always consumes data according to pattern consumePattern. ModalSink is dynamic in that it is a modal model with two modes: in the first one, it consumes data according to consumePattern1; in the second, according to consumePattern2. The Source produces 3 2D arrays of size [x=3,y=2] each, in 3 consecutive iterations. The StaticSink fires once per iteration and outputs the entire array as a single pattern. The ModalSink also fires once per iteration, using different patterns. It switches patterns when it reads an array element with value 10.

- produceBase: [x = 0, y = 0]
- producePattern: [x = {1,1}, y = {1,1}]
- produceTiling: [x = 1, y = 1]

- consumeBase: [x = 0, y = 0]
- consumePattern: [x = {3,1}, y = {2,1}]
- consumeTiling: [x = 1, y = 1]

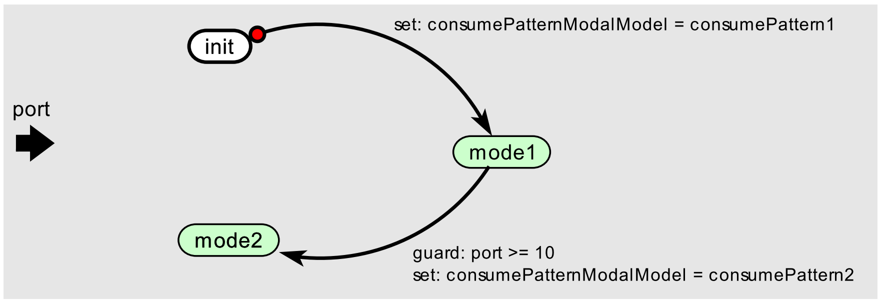
- consumePattern1: [y = {2,1}, x = {1,1}]
- consumePattern2: [x = {2,2}, y = {2,1}]

- consumePatternModalModel: [x = {2, 2}, y = {2, 1}]

This parameter is used by the input port of the modal model, and is modified dynamically upon transitions. Note that it is immediately initialized but the first (preemptive) transition of the modal model.

Note also that the director of the modal model is set to MultirateFSMDirector.

Figure 1.2: Top level of `demo/Dynamic/ModalModelPthales.xml`.



set: consumePatternModalModel = consumePattern1

guard: port >= 10
set: consumePatternModalModel = consumePattern2

Figure 1.3: Top level of ModalSink actor of `demo/Dynamic/ModalModelPthales.xml`.

is shown in Figure 1.2. This model illustrates how parameter changes can be made dynamically. The model has two Sink actors: StaticSink is static in the sense that it always consumes data according to pattern `consumePattern`. ModalSink is dynamic: it is a modal model with two modes. In the first mode, ModalSink consumes data according to `consumePattern1=[y={2,1},x={1,1}]` in the second, according to `consumePattern2=[x={2,2},y={2,1}]`.

The Source actor has `repetitions` set to `{3,2}`. Because of this, and its other parameters as shown in the figure, it produces a 2-dimensional array of size `[x=3,y=2]` at each iteration. The model executes in 3 iterations (specified in the `iterations` parameter of the PthalesDirector) therefore three 2-D arrays are produced in total. They contain the integers $1, 2, 3, ..., 18$.

The StaticSink fires once per iteration and outputs the entire array as a single pattern. The ModalSink also fires once per iteration, but uses different patterns. The state machine of ModalSink is shown in Figure 1.3. The state machine has three states. State `init` is the initial state. This state is only used for initialization: it is immediately exited by the *preemptive* transition leading to `mode1`. This transition serves to initialize the parameter `consumePatternModalModel` to `consumePattern1`. In turn, the `pattern` parameter of the input port of ModalSink is set to `consumePatternModalModel`. Therefore, by setting `consumePatternModalModel` to `consumePattern1` we are effectively setting the pattern of ModalSink to `consumePattern1`.

The reason this "dynamic" initialization is needed, as opposed to a "static" one in the way `produceBase` and all other parameters are initialized, is the following. While the model executes, `consumePatternModalModel` changes. In particular, the state machine switches from `mode1` to `mode2` when an array element with value 10 is read at the input port. At that point, `consumePatternModalModel` is set to `consumePattern2` and maintains that value until the end of execution. If the model is saved at that point, it will be saved with `consumePatternModalModel` having that value. When the model is re-opened, we see that `consumePatternModalModel` has value `[x={2,2},y={2,1}]`, as shown in Figure 1.2. To avoid such problems, we initialize `consumePatternModalModel` explicitly.

States `mode1` and `mode2` are refined (as indicated by their green color). They are refined to SDF models similar to the one refining StaticSink. These SDF models are not shown here ("look inside" the states to see them).

1.2.2 Pthales with Header Information

A common type of parameter-dynamic Pthales models is the case where the pattern and tiling parameters do not change dynamically, but the size of the multidimensional arrays changes dynamically. In particular, the size of the incoming arrays that are to be processed by a certain actor changes dynamically. This implies that the number of firings needed to process an array also changes dynamically. This number is derived from the *repetitions* parameter of an actor, which is more generally a vector of integers specifying the bounds of a set of nested loops.

To address this class of applications, we have extended the Pthales domain with a notion of *header tokens* that can be prepended to the data transferred between actors. Header tokens can be viewed as control tokens, which are different from regular data tokens. Header tokens carry information about the size of the multidimensional array to follow. This information is produced by an up-stream actor and communicated to a down-stream actor. The down-stream actor uses this information to recompute its own repetitions parameter, thus being able to consume multidimensional arrays of dynamically varying sizes.

An example model that uses the header capabilities described above is the model `ThalesDynamic.xml` that can be found under `ptolemy/domains/pthales/demo/Dynamic/`. The top level of this model is shown in Figure 1.4. In this model the Source actor produces a 10×10 array of two dimensions called "dim1" and "dim2" in a single firing (the pattern equals the size of the produced array). The information about the size of the produced array is stored as a parameter called `size` of the input port of the PthalesAddHeaderActor. This information is "packaged" into a header and transmitted to the sink.

The sink is a PthalesDynamicCompositeActor the internals of which are shown in Figure 1.5. The `repetitions` parameter of the sink is initially set to `{}`, that is, empty. This indicates to the Pthales Director that the repetitions for this actor must be computed dynamically, using the size information contained in

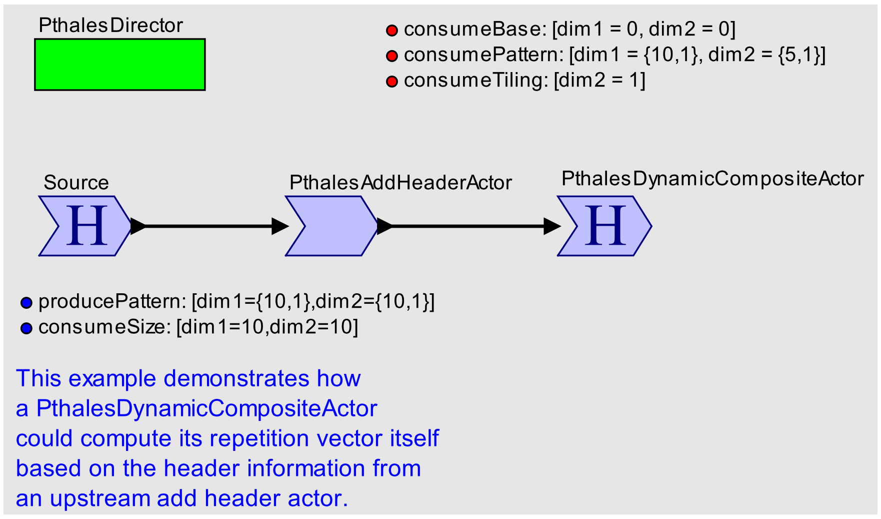

PthalesDirector
consumeBase: [dim1 = 0, dim2 = 0]
consumePattern: [dim1 = {10,1}, dim2 = {5,1}]
consumeTiling: [dim2 = 1]
Source
PthalesAddHeaderActor
PthalesDynamicCompositeActor
H
H
producePattern: [dim1={10,1},dim2={10,1}]
consumeSize: [dim1=10,dim2=10]
This example demonstrates how a PthalesDynamicCompositeActor could compute its repetition vector itself based on the header information from an upstream add header actor.

Figure 1.4: Top level of `Dynamic/ThalesDynamic.xml` model.

the received headers, as well as the statically defined parameters `pattern` and `tiling` of the input port of the sink actor. As can be seen from the specification of these parameters in the figure, at each firing, the sink consumes a 10×5 "chunk" of the input array, and advances by 1 in the dim1 dimension after every firing. Therefore, in total, the sink will need to fire a total of 6 times to consume the entire input array. This is then the value of its `repetitions` parameter which is computed dynamically.

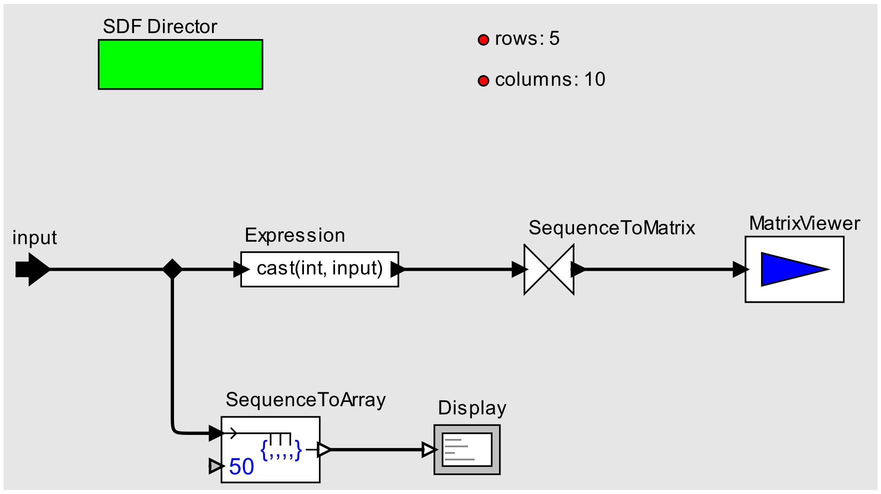
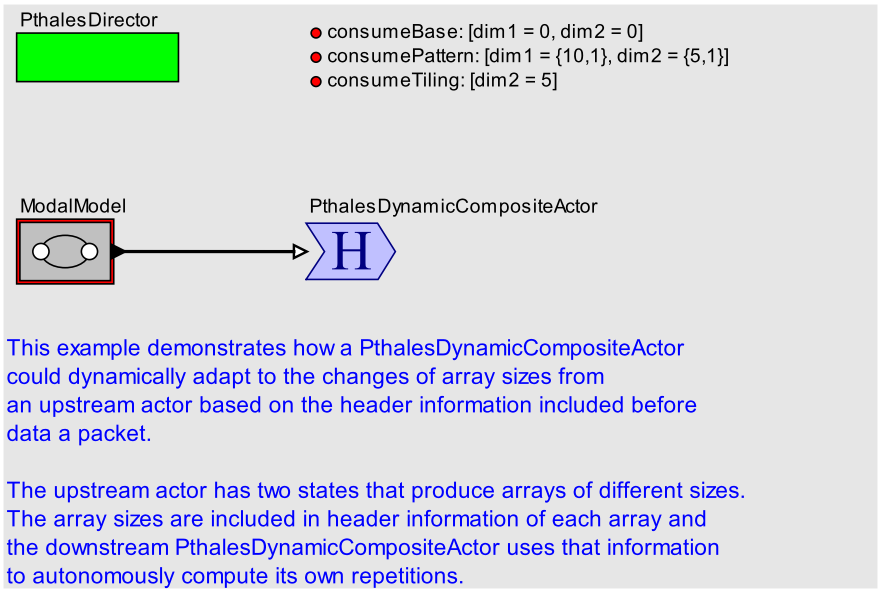


Figure 1.5: Internals of PthalesDynamicCompositeActor of the model of Figure 1.4.

1.2.3 Combination of Modal Models and Headers

The model of Figure 1.4 illustrates the dynamic computation of `repetitions` parameter using header information. However, this model is not completely dynamic in the sense that the Source actor produces only a single multidimensional array, with a fixed size. An extension of that model to illustrate true dynamicity is the model `ThalesDynamicModalModel.xml`, which is found under `ptolemy/domains/pthales/demo/Dynamic/`. The top level of this model is shown in Figure 1.6. In this model, the Source and PthalesAddHeaderActor actors are replaced by a Modal Model actor, the automaton of which is shown in Figure 1.7.


PthalesDirector
consumeBase: [dim1 = 0, dim2 = 0]
consumePattern: [dim1 = {10,1}, dim2 = {5,1}]
consumeTiling: [dim2 = 5]
ModalModel
PthalesDynamicCompositeActor
H

This example demonstrates how a PthalesDynamicCompositeActor could dynamically adapt to the changes of array sizes from an upstream actor based on the header information included before data a packet.

The upstream actor has two states that produce arrays of different sizes. The array sizes are included in header information of each array and the downstream PthalesDynamicCompositeActor uses that information to autonomously compute its own repetitions.

Figure 1.6: Top level of `Dynamic/ThalesDynamicModalModel.xml` model.

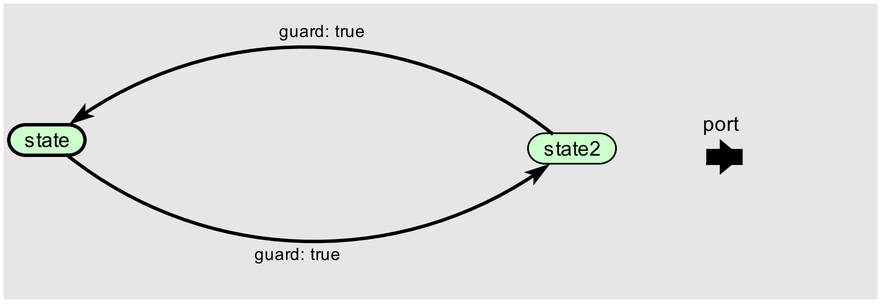


Figure 1.7: State machine of the ModalModel of Figu re 1.6.

The Modal Model alternates between two states, and in each of the two states it produces an array of different size: an array of size 10×10 at state “state” and an array of size 10×5 at state “state2”. The refinements of “state” and “state2” are shown in Figures 1.8 and 1.9. In each refinement, a PthalesAddHeaderActor is used to package the information about the size of the produced arrays using headers.

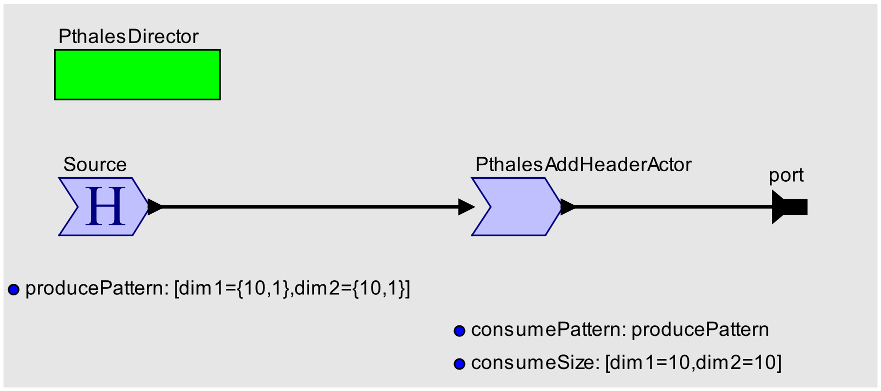

PthalesDirector
Source
H
PthalesAddHeaderActor
port
producePattern: [dim1={10,1},dim2={10,1}]
consumePattern: producePattern
consumeSize: [dim1=10,dim2=10]

Figure 1.8: Refinement of the initial state of the state machine of Figure 1.7.

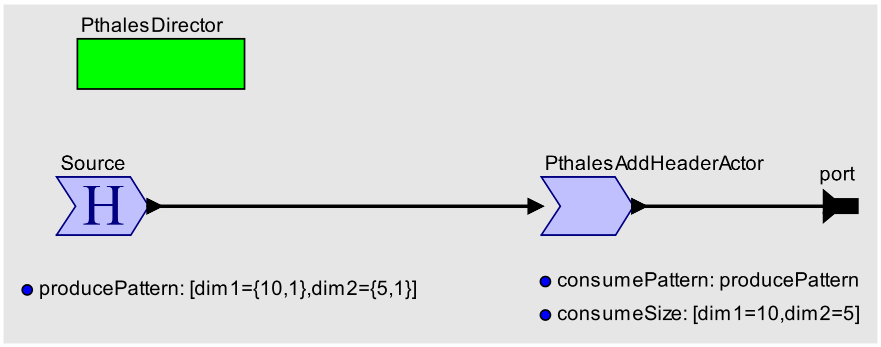

PthalesDirector
Source
H
PthalesAddHeaderActor
port
producePattern: [dim1={10,1},dim2={5,1}]
consumePattern: producePattern
consumeSize: [dim1=10,dim2=5]

Figure 1.9: Refinement of “state2” of the state machine of Figure 1.7.

Chapter 2

Semantics of Array Specifications

2.1 Introduction

In order to specify consumption and production behavior of actors in the Pthales model, a specification language must be developed with a clear and complete semantics. Such a language must allow all semantically relevant details regarding consumption and production to be derived without ambiguity. Beyond these semantics should only be implementation details, which given a particular model with an expected behavior can vary while maintaining this behavior uniquely. Such a language must also be compatible with the existing SpearDE platform, expanding on its semantics to allow the extension of its capabilities into dynamic cases.

2.2 Array-OL Specification Language

The language specified by [2] as a basis of the Array-OL computational model provides a semantics for consumption and production behavior in terms of affine operations on index spaces. With small modifications, extensions, and appropriate constraints, this language can be used to clearly represent parameters of Pthales domain models as well, and allow the evolution of its current rectilinear behavior into more general index transformations.

The Array-OL language also suggest a mathematical means of reasoning about scheduling and compositional properties. Particularly in the dynamic Pthales case, it is useful to have a clear mathematical representation of the way data is mapped between spaces and iterated over. Such a representation may give an efficient way to quickly derive dependent parameters from ones given explicitly or which change dynamically. This kind of a representation can also be useful in allowing parameters to be calculated from given formulas in an expression language.

2.3 Local Semantics of Pthales

The semantics of multidimensional dataflow, in the sense of [12], [2], and the Pthales model, determine how a producing or consuming actor interacts with data structured into multidimensional arrays. These arrays can generally be thought of as discrete spaces with some number of finite dimensions. However, in Array-OL a meaning is given to one particular dimension being unboundedly large.

The interaction of actors that read from and write to these abstract data-spaces can be treated in a way similar to, but not necessarily commensurate with, dataflow when this process can be characterized as writing and reading streams of tokens containing regions of this space (*patterns*). The compatibility of these semantics with those of dataflow necessitates that the reading and writing of regions at any port be strictly ordered in a causal manner, although strict compatibility may not be an aim of these semantics in which case this constraint can be loosened. However, the semantics of Array-OL for instance are not themselves dataflow [3], but rather are single-assignment on the array spaces within iterations. While MDSDF and Array-OL make certain constraints, those of Pthales might be altogether different.

For abstract spaces of finite dimensions, an actor can be treated as having completed a full iteration when all of the constituting operations have been performed over the array, usually covering the space with a series of regions. The operation of a model then would consist, coarsely, of a series of such iterations. In the case of Array-OL, where a dimension is potentially infinite the corresponding iteration may not represent a finite process.

2.3.1 A Trivial Semantics

This basic concept of actors writing to and reading from abstract arrays can trivially be treated from the perspective of the actors as carrying out in a firing the entire set of operations on array spaces constituting an iteration. At this coarse level this trivial case, a model can be treated as having HSDF semantics where each token represents the entire space handled within an iteration, either being written to or read. As is noted above, in Array-OL this could mean a course grain iteration is never actually finished if one of the dimensions is unbounded.

The sizes of the spaces could vary from port to port, but would certainly have to be shared between connected ports. Amongst connected ports, the addressing of this space would have to be consistent. Any traversal or regional projection of the space would have to be done in the actors themselves, explicitly referring to coordinates within this space that share common meaning.

At this trivial level, the most general parameter that would be required to carry out this process would be the shape of the shared space. In the most general case, this shape would have to include both dimensions and topological characteristics. While the former of these is clear, the latter could for instance consists of information about how to treat the boundaries of the space. This difference does exist across different forms of multidimensional models of computation.

Specifically, Array-OL treats data spaces as toral in all finite dimensions, so addressing out further than the size of any dimension aliases back into the space. In other words, coordinates in Array-OL are modular with respect to the dimensions. Other multidimensional models do not allow the meaningful access of coordinates outside the space. This difference is important in cases where ordered traversals over the space are performed; starting from the inside of a space and traversing past the boundary back around to the starting point is used in Array-OL as a means of offsetting strides.

Basic Parameters

Assuming that the abstract space is a rectilinear array, the most basic parameter is a vector of positive integers representing its dimensions, which will be denoted $\vec{m}$. The coordinates of this space then are integer vectors of $dim(\vec{m})$ where for any such coordinate $\vec{x}$, $0 < x_k \leq m_k$. This will be notated further as $\vec{0} \sqsubseteq \vec{x} \sqsubset \vec{m}$, emphasizing the lack of an explicit traversal of this range. Such an explicit traversal totally orders the elements of the array space, usually a dictionary total ordering is imposed where this is useful.

At this level, a question can be raised regarding inconsistent specifications across actors that share a relation, whether they are permitted in the semantics, and if so, how they are handled. Each of these options may have particular advantages and offering them as options might allow for a broader set of cases to be handled

- Inconsistent sizes on a relation could be disallowed, raising an exception. But if this is done in a dynamic model it could be difficult to maintain this consistency.
- Producing specifications could set the size to the smallest space containing all of the produced sizes, while consuming specifications could read from any space equal to or smaller than this produced space.
- Consuming specifications could read from larger sizes but would be filled with null or zeroed elements. The subject of null/zero elements will arise anyways when considering reading decimated areas or areas that have differing offsets if these are allowable.

Clearly, these cases are decreasingly strict, and the less strict the assumptions the broader and more flexible the specifications will be. In the case of dynamic semantics, the size of the space may be in part determined by upstream actors dynamically. Repetitions (as discussed below) are set to match the size, however other factors might prevent the sizes from being consistent in the stricter cases, for instance if multiple inputs are mutually determining the space, and consequently models would either have to be rejected or admit weaker assumptions.

2.3.2 Full Semantics

Within an iteration, multidimensional dataflow can specify a series of constituting firings, each consuming or producing a *pattern* with the full space covered by an iteration. Making this clear involves specifying the patterns themselves, and how the space is traversed in terms of these patterns; this is known a *tiling* the space.

In terms of these semantics, an actor produces and consumes streams of patterns. Since the semantics of the model specify how these patterns are contextualized into the larger abstract spaces, the actors themselves no longer need to internally know the context of these patterns nor do they need to be aware explicitly of the coordinate system of the abstract space. Actors simply produce patterns, with their own coordinates, and the pattern coordinates can be transformed into the coordinates of the abstract space given the semantics of the tiling.

Affine Index Transformation

In producing and consuming *patterns* over *tilings*, the index space as viewed by the actor, which will be called the *local index space*, is transformed into that of the array space, which will be called the *array index space*. This transformation between local and array index spaces corresponds to the location of the same data in both spaces corresponding to different coordinates during a firing. The implementation of this transformation could conceivably mean the data is actually copied from (or to) locations as addressed by the local index space to (or from) corresponding locations addressed by the array index space. But other implementations are certainly possible such as the actors using directly transformed addresses to act on the common data, or something altogether different.

The transform used in both *patterns* and *tilings* is assumed in this case to be affine transforms between rectilinear index spaces. Hence, each of these transforms, addressed in further detail below, can be characterized in general by an *affine index mapping*. Such a mapping between local index space $Ind_L = \mathbb{N}^N$ and array index space $Ind_A = \mathbb{N}^M$ can be described as $\mathcal{A}(b,\, \sigma,\, s) : Ind_L \to Ind_A$, where the parameters of the transform are as follows:

b	$\mathbb{N}^M$	– base index
σ	$\mathbb{N}^M_N$	– $M \times N$ integer matrix, linear step
s	$\mathbb{N}^N$	– index space size (domain size)

which forms the transform:

$$\mathcal{A}(b,\, \sigma,\, s;\, x) = b + \sigma x,\ 0 \sqsubseteq x \sqsubset s \tag{2.1}$$

It is convenient not only to refer to the range of this transform, given the domain is specified by s, but also the smallest rectilinear box in the codomain containing the range of the transform, which will be called the *bounding box*. This box, $\mathbf{B}(\mathcal{A})$, in the array index space can be represented by two opposing corner indices (inclusively).

Dimension-wise Affine Index Transformation

A simpler subset of affine index transformations are those that perform affine transformations in each dimension independently. Rather than a full matrix equation, such a transformation can be formulated as a series of one-dimensional affine transformations:

$$\mathcal{A}_{\mathcal{D}}(b,\, \mathbf{a},\, s;\, x_i) = \begin{cases} b_i + a_i x_i & \text{if } i < dim(\mathbf{a}) \\ b_i & \text{otherwise} \end{cases} \qquad 0 \leq x_i < s_i \tag{2.2}$$

where $\mathbf{a}$ is only a vector of dimension N, rather than an $M \times N$ matrix. If M and N are different, that is the local index space is of lower dimension than the array index space, the higher-order coordinates are fixed by the base index. This simpler transform can be embedded into (2.1) by embedding N dimensional vector $\mathbf{a}$ in the diagonal of an $M \times N$ matrix σ_a. This embedding allows the dimension-wise mappings of Pthales to be described in Array-OL's matrix notation.

Patterns

The atomic element of these full semantics for multidimensional are the patterns produced and consumed by actors, which refer to a mapping between an array handled in the actor and a region of the abstract space. In most general terms, this mapping requires a specification of the space from the perspective of the actor on a particular port and a function mapping the points in this space (locations of data) to points in the pattern that is part of the abstract space constituted on a relation between actors.

In Array-OL, this mapping of the local index space to patterns in the array index space is an affine index transformation $\mathcal{A}(\vec{x}_p,\, \mathbf{P},\, \vec{d})$, where $\vec{d}$ is the size of the local index space of the patterns and $\mathbf{P}$ is the pattern transformation matrix (in the Array-OL literature this is given the name $\mathbf{F}$ and called the *fitting* matrix, but $\mathbf{P}$ is used here to be consistent with the terminology of the Pthales model). The origin of the pattern is $\vec{x}_p$ denoting the position of the pattern within the array index space; it is this parameter that usually changes iteratively to traverse the array index space.

In the Pthales model, contrastingly, the pattern transformation is only dimension-wise affine, though the Pthales model will likely be expanded to support broader classes of transformations. Following the notation of SpearDE, the pattern is specified by a series of named parameters for each dimension of the form

$$\textbf{pattern} = [a = x_a,\, b = x_b, \ldots]$$

where (a, b, ...) are names of dimensions and (q_a, q_b, ...) are associated data. The data here for each of q_n are of the form $\{d_n,\, p_n\}$. Parameters d_n are the sizes of the local index space in each dimension n, hence d_n form a vector $\vec{d}$ as in the above Array-OL notation, given that the dimension names are enumerated in some consistent way. The p_n parameters are the step sizes in each dimension n of the pattern in the array index space, hence, as a dimension-wise affine index transformation, p_n form the diagonal entries of matrix $\mathbf{P}$, with all other entries being 0. In terms of the pattern transformation, the Pthales specification gives an affine index transformation. The order of names in the specification however also carries an additional piece of information that will be discussed below.

Tiling

The tiling of each atomic pattern produced or consumed iterates the origin of patterns over the space covering it in some fashion with patterns. Another local index space corresponds to the origins of each tile containing a pattern and is traversed over a series of actor firings. The size of this space, and hence the number of iterations in each dimension, corresponding to nested loops, are notated $\vec{q}$ and referred to as *repetitions*.

Since these repetitions correspond to firings of the actors, they are usually constrained to be consistent over all producing and consuming ports as a parameter of the actor as a whole. The semantics of the repetitions are usually interpreted as the atomic actions of the actors on all incoming patterns to produce all outgoing patterns, will part of the actor's state being the particular coordinate in the local index space of repetitions. Because of such a semantics, the parameters of this space of repetitions is important to reason about with respect to the other parameters that are usually more predetermined. On the other hand, a less restrictive semantics with a cyclostatic series of firings could decouple the repetitions on different ports.

As with the patterns, in Array-OL the tilings are affine index transforms from the repetition space to the array index space: $\mathcal{A}(\vec{x}_0, \mathbf{T}, \vec{q})$, where $\vec{q}$ is the size of the repetition space as described above and $\mathbf{T}$ is the tiling matrix (in the Array-OL literature this is given the name $\mathbf{P}$ and called the *paving* matrix, note this so as not to confuse $\mathbf{P}$ here with that in the conventions of Array-OL notation). The origin of the tilings is known as the offset parameter $\vec{x}_0$.

Likewise, in the current Pthales model the transformation is only dimension-wise affine. The same SpearDE specification notation (2.3.2) is used, but the data associated with each dimension consists of only a single value t_n for the tiling step in dimension n. As in the case of the Pthales pattern parameters, these values can be embedded into the Array-OL formulation as the diagonal entries of $\mathbf{T}$. The size parameter here, which would correspond to the repetitions index space size, is not given with the step parameter because repetitions are semantically associated with the whole actor rather than with a particular port in Pthales.

The total transformation between the indices of the local and repetition spaces and the indices of the array space is then a composition of the pattern and tiling transformations. This transformation is given as follows:

$$\begin{aligned}\mathcal{F}(\vec{x}_0, \mathbf{T}, \vec{q}, \mathbf{P}, \vec{d}; \vec{u}, \vec{k}) &= \mathcal{A}(\mathcal{A}(\vec{x}_0, \mathbf{T}, \vec{q}; \vec{u}), \mathbf{P}, \vec{d}; \vec{k}) \\ &= \vec{x}_0 + \mathbf{T}\vec{u} + \mathbf{P}\vec{k}, \quad \vec{0} \sqsubseteq \vec{u} \sqsubset \vec{q}, \quad \vec{0} \sqsubseteq \vec{k} \sqsubset \vec{d}\end{aligned} \tag{2.3}$$

Iterative Ordering

Up to the iterative ordering necessary to specify the traversal of tilings, specification (2.3) gives the semantic specification for multidimensional consumption or production at a port of an actor. This final detail of iterative ordering can arguably be fixed to the order of the dimensions as specified in the context of the $\vec{d}$ and other specifications. This is enough since the $\mathbf{T}$ matrix can simply permute its rows to achieve different orders of dimensions in the index space.

Nevertheless, there may be a reason why specifying this explicitly is advantageous. Supposing in the dynamic case that this order changes it might be simpler to perserve the $\mathbf{T}$ matrix and consequently perserve the computations which do not use ordering in the scheduling algorithm. In the specification syntax for Pthales models, this ordering is derived from the order of the named dimensions. But in using a formalization closer to that of Array-OL, $\mathbf{T}$ could be considered to have itself a fixed ordering while another permutation matrix factor γ could be used on the right to permute the indicies, so that the step is $\mathbf{T}\gamma\vec{u}$. There are other possibilities for how this can be specified explicitly.

2.3.3 Specification

For a given port producing or consuming patterns the parameter set

$$\mathbf{M} = (\mathbf{x}_0, \mathbf{T}, \mathbf{q}, \mathbf{P}, \mathbf{d}, \mathbf{m}), \tag{2.4}$$

possibly with an iterative ordering γ and a given topology for the abstract space, specifies a the behavior of multidimensional dataflow in a general way that accommodates linear patterns and affine tilings. The patterns are not so much themselves "affine" since the offset of a given pattern is the origin of its tile.

From $\mathbf{M}$, a transform $\mathcal{F}_\mathbf{M}$ of the form (2.3) can be specified from the local index space and repetition space to the array index space. That is, at a particular port $(\vec{k}, \vec{u}) \xrightarrow{\mathcal{F}_\mathbf{M}} \vec{x}$, where the actor addresses elements of the patterns at the port with $\vec{k}$ and the actor has a repetition index of $\vec{u}$. This parameter set is equivalent to the one used in Array-OL with the constraint that the topology of the array index space is fixed as toral.

2.4 Calculating Specification Parameters

Although the specification $\mathbf{M}$ can fully determine the behavior of a port in a model, certain additional constraints can make parts of $\mathbf{M}$ computed from others given as specified. In Array-OL for instance, $\vec{\mathbf{q}}$ is calculated from the other parameters given the constraint that the whole space be covered by tilings. Following the analogy with SDF models, in MDSDF this repetitions vector can also be calculated along with the size $\vec{\mathbf{m}}$ to satisfy rate relationships between producers and consumers on a relation so that their traversals repeatedly cover the same 'least common region'.

Another possibility is that some of the parameters are specified in terms of others in the form of an expression or relation. In general, if a function can be given that maps to the entire parameter set from some alternative set, the alternative can be used as a specification. This behavior could be used so that dynamic changes in some parameters imply the new values of others.

2.4.1 Bounding Boxes

As described in 2.3.2, the affine index transformations that make up the patterns and tilings can be given rectilinear bounding boxes in the array index space corresponding respectively to the minimum rectilinear index space containing the image of a pattern, $\mathbf{B}(\mathcal{A}_P)$, and the minimum rectilinear index space containing the origins of tilings, $\mathbf{B}(\mathcal{A}_T)$. These boxes can, as discussed before, be described by the index vectors of opposite corners.

Given an affine index transformation $\mathcal{A}(b, \sigma, s)$, the corners (a_-, a_+) of bounding box $\mathbf{B}(\mathcal{A})$ can be calculated from the corners of the index space $\vec{0}$, $s - \vec{1}$. Let σ_+ and σ_- be the matrices containing only the positive elements of σ, and all the negative elements respectively, with all other entries set to 0. In terms of these two matrices, and shifted by the offset b

$$a_\pm = \sigma_\pm(s - \vec{1}) + b \tag{2.5}$$

Pattern Bounding Box

The pattern bounding box is described by opposite corners (p_-, p_+) calculated from given parameters $(\mathbf{P}, \vec{\mathbf{d}})$. The bounding box origin is considered to simply be the origin of the pattern and itself is given no offset, though in a generalization one might extend these computations to accomodate such an offset. This computation is done with (2.5).

Example 1 For instance, consider the pattern specified as

$$\mathbf{P} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}.$$

Since $\mathbf{P}_- = 0$, $p_- = \vec{0}$, which will generally be the case if the linear transformation matrix is non-negative. The index space is stretched or sheared forward. Then,

$$p_+ = \mathbf{P}_+(\mathbf{d} - \vec{1}) = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ 9 \end{bmatrix}.$$

That is, the bounding box has then corners at $[0,\ 0]^T$ and $[13,\ 9]^T$.

Example 2 Another example would be the following

$$\mathbf{P} = \begin{bmatrix} -1 & 3 \\ -2 & -2 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} 6 \\ 3 \end{bmatrix}.$$

In this case, there are non-trivial $\mathbf{P}_-$ and $\mathbf{P}_+$:

$$p_- = \mathbf{P}_-(\mathbf{d} - \vec{1}) = \begin{bmatrix} -1 & 0 \\ -2 & -2 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \end{bmatrix} = \begin{bmatrix} -5 \\ -14 \end{bmatrix} \tag{2.6}$$

$$p_+ = \mathbf{P}_+(\mathbf{d} - \vec{1}) = \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \end{bmatrix}. \tag{2.7}$$

That is, the bounding box has then corners at $[-5, -14]^T$ and $[6, 0]^T$.

The corners of the tiling bounding box can be computed in the same fashion giving one a pair (t_-, t_+) from given parameters $(\mathbf{T}, \mathbf{q})$ and shifting with $\mathbf{x_0}$.

Minimum Array Index Space Bounding Box

The third bounding box that one can derive from the first two is that of the minimal array index space, $\mathbf{B}(\mathcal{A}_M)$, with corners at (a_-, a_+), containing all of the mapped indices from the local index space over all repetitions. Since the bounding box of the tilings form the space of origins of patterns, the minimal array space box can be found by increasing the tiling bounding box to fit patterns at the extreme opposite corners. In the case of all positive $\mathbf{P}$ only the greater-valued corner needs to be increased since all pattern indices are mapped equal to or greater than the index of the origin of the pattern. However, in the case of negative elements in $\mathbf{P}$, the offset of the tiling must accomodate patterns that map indices behind the origin.

Given the pattern bounding box, the corners of which are at (p_-, p_+), either of (a_-, a_+) or (t_-, t_+) can be calculated from the other. The constraint that must be met by the size $\mathbf{m}$ is

$$\vec{0} \sqsubseteq t_- + p_-,\ t_+ + p_+ \sqsubset \mathbf{m} \tag{2.8}$$

which means that

$$a_\pm = t_\pm + p_\pm \tag{2.9}$$

where for a minimally sized array index space $a_+ = \mathbf{m} - \vec{1}$, and since the local index space should be non-negative the additional constraint can be imposed on $\mathbf{x_0}$ that it be appropriately accommodating such that $\vec{0} \sqsubseteq a_-$.

2.4.2 Calculating Size

Using (2.5) with the pattern and tiling bounding boxes, and combining them with (2.9), the parameter $\mathbf{m}$ can be calculated from a specification $(\mathbf{x}_0, \mathbf{T}, \mathbf{q}, \mathbf{P}, \mathbf{d})$ as

$$\mathbf{m} = \mathbf{x_0} + \mathbf{P}_+(\mathbf{s} - \vec{1}) + \mathbf{T}_+(\mathbf{q} - \vec{1}) + \vec{1} \tag{2.10}$$

and must satisfy the constraint

$$\mathbf{x_0} \sqsupseteq -\mathbf{P}_-(\mathbf{s} - \vec{1}) - \mathbf{T}_-(\mathbf{q} - \vec{1}) \tag{2.11}$$

from which $\mathbf{x_0}$ can be calculated as well as the equality case of (2.11).

Example 3 An example of calculating size and minimum offset would be as follows. Given

$$\mathbf{T} = \begin{bmatrix} 1 & 3 \\ 2 & -2 \end{bmatrix},\ \mathbf{q} = \begin{bmatrix} 4 \\ 4 \end{bmatrix},\ \mathbf{P} = \begin{bmatrix} -1 & 3 \\ -2 & -2 \end{bmatrix},\ \mathbf{d} = \begin{bmatrix} 6 \\ 3 \end{bmatrix}.$$

and using (2.11) first to find the offset

$$\mathbf{x_0} = -\begin{bmatrix} -1 & 0 \\ -2 & -2 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 20 \end{bmatrix} \tag{2.12}$$

the size $\mathbf{m}$ can be computed

$$\mathbf{m} = \begin{bmatrix} 5 \\ 20 \end{bmatrix} + \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 24 \\ 27 \end{bmatrix} \tag{2.13}$$

2.4.3 Calculating Repetitions

Taking formula (2.10) and constraint (2.11) and solving for $\mathbf{q}$ gives a method to calculate repetitions from ($\mathbf{x}_0$, $\mathbf{T}$, $\mathbf{m}$, $\mathbf{P}$, $\mathbf{d}$), which in contrast with the above takes the size to be given. These formulas become

$$\mathbf{T}_+\mathbf{q} \sqsubseteq \mathbf{T}_+\vec{1} + \mathbf{m} - \mathbf{x_0} - \mathbf{P}_+(\mathbf{d} - \vec{1}) - \vec{1} \tag{2.14}$$

and the constraint

$$\mathbf{T}_-\mathbf{q} \sqsupseteq \mathbf{T}_-\vec{1} - \mathbf{x_0} - \mathbf{P}_-(\mathbf{d} - \vec{1}) \tag{2.15}$$

Here, inequality is used since the value of $\mathbf{q}$ must be an integer vector, and given a general set of parameters this equality is not necessarily met by an integer vector. In the most general cases, finding a positive $\mathbf{q}$ that is maximum given these constraints forms an integer linear program, but in simple cases this computation can be reduced. In particular, for the case of diagonal positive $\mathbf{T}$ and $\mathbf{P}$ corresponding to a rectilinear lattice, that is a dimension-wise affine index transformation, the expressions can be given as follows:

$$\mathbf{q} = \lfloor \mathbf{T}^{-1}(\mathbf{T}\vec{1} + \mathbf{m} - \mathbf{x}_0 - \mathbf{P}(\mathbf{d} - \vec{1}) - \vec{1}) \rfloor$$

Example 4 An example of calculating the repetitions from the known size with for a dimension-wise parameter set is as follows. Given

$$\mathbf{T} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}, \ \mathbf{m} = \begin{bmatrix} 40 \\ 29 \end{bmatrix}, \ \mathbf{P} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \ \mathbf{d} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}, \ \mathbf{x}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

the repetitions vector $\mathbf{q}$ can be computed

$$\mathbf{q} = \lfloor \begin{bmatrix} 1/2 & 0 \\ 0 & 1/3 \end{bmatrix} \left(\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 40 \\ 29 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \rfloor = \begin{bmatrix} 19 \\ 7 \end{bmatrix} \tag{2.16}$$

Conclusions and Future Work

This document presented preliminary work on a multidimensional model of computation, called Pthales, implemented in the heterogeneous modeling framework Ptolemy II. Pthales follows the SpearDE specification and code generation environment developed by Thales. Pthales extends SpearDE towards a more dynamic model of computation via modal models and a type of control tokens (headers).

Future work can be classified among multiple directions:

- System-level modeling: As mentioned in the introduction, one of the goals of this project is to reach a point where a full system-level application can be modeled in Ptolemy. This application will include signal processing, but also other parts of the system, including control logic, as well as parts outside the system per se, namely, the environment in which the system operates. We believe that this modeling challenge will be highly inspiring and provide multiple opportunities, including integration of Pthales with other domains in Ptolemy, as well as potential extensions to the Ptolemy modeling framework itself.
- Control tokens: We feel that the current approach of control tokens via headers is not the only possible approach, and it would be interesting to examine alternatives and compare the pros and cons of each. One possibility in particular is to distinguish explicitly control and data channels, and insist that control and data tokens only "travel" on their corresponding channels, but are not mixed together as is currently done.
- Code generation: The primary element of model-based design is modeling and model analysis (say, by simulation), however, another crucial element is automatic code generation from models. In the context of Pthales and Ptolemy, one direction is towards highly optimized parallel code generation from the signal-processing part of the application. Another direction, perhaps more interesting, is control logic code generation. The latter part in fact represents currently a limitation for tools such as SpearDE which provide little or no support for it.

Bibliography

[1] Abdelkader Amar, Pierre Boulet, and Philippe Dumont. Projection of the array-ol specification language onto the kahn process network computation model. In *ISPAN '05: Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks*, pages 496–503, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Abdelkader Amar, Pierre Boulet, and Pierre Dumont. Projection of the Array-OL Specification Language onto the Kahn Process Network Computation Model. Research Report RR-5515, INRIA, 2005.

[3] P. Boulet. Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing. Research Report RR-6467, INRIA, 2008.

[4] Alain Demeure and Yannick Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis Conf. on Micro-Electronics (SAME 98)*, 1998.

[5] Philippe Dumont and Pierre Boulet. Another Multidimensional Synchronous Dataflow: Simulating Array-OL in Ptolemy II. Technical report, DART - INRIA Futurs - INRIA - CNRS : UMR8022 - Université des Sciences et Technologies de Lille - Lille I, 2005.

[6] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.

[7] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, January 1987.

[8] E. A. Lee and S. Tripakis. Modal Models in Ptolemy. In *EOOLT 2010 – 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, October 2010. Voted Best Paper. Available at: http://chess.eecs.berkeley.edu/pubs/700.html.

[9] E.A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proc. 7th ACM & IEEE Intl. Conf. on Embedded software*, pages 114–123. ACM, 2007.

[10] Edward A. Lee. Finite State Machines and Modal Models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009. Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html.

[11] J. Liu and E.A. Lee. On the causality of mixed-signal and hybrid models. In *HSCC*, pages 328–342, 2003.

[12] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, July 2002.