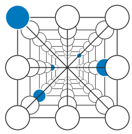


Project no. 248828

ADVANCE

Strategic Research Partnership (STREP)
ASYNCHRONOUS AND DYNAMIC VIRTUALISATION THROUGH PERFORMANCE
ANALYSIS TO SUPPORT CONCURRENCY ENGINEERING

Statistical Analysis
D20

Due date of deliverable: April 30th, 2012
Actual submission date: May 30th, 2012

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP4
Task number: WP4b

Responsible institution: USTAN
Editor & and editor's address: Kevin Hammond
University of St Andrews
School of Computer Science
KY16 9SX St Andrews, Scotland

Version 0.1 / Last edited by Philip Hölzenspies / November 19, 2013

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP4b	Develop static analyses	USTAN*, TWENTE

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Chapter 1

Definitions

In this chapter, we introduce more precisely the terminology required to discuss observations and statistical models of the corresponding behaviour. Since we express temporal behaviour in terms of throughput, latency and jitter, we first introduce definitions related to these concepts. We go on to discuss different types of dependencies in stream programming concurrent systems that are the purview of the ADVANCE project.

1.1 Latency

Even within the confines of the ADVANCE project, there are different interpretations of what constitutes latency. In figure 2.5, we illustrate three different types of latency in an event trace of the VRNET depicted in figure 1.1.

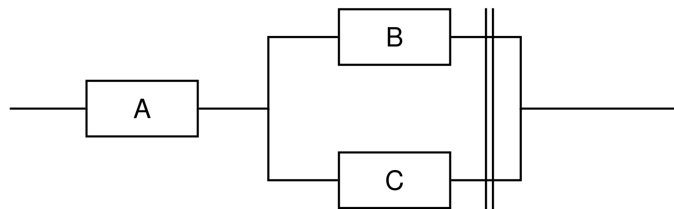


Figure 1.1: Example VRNET program, viz. A..B||C

The program consists of three boxes (A, B, C). All input to the program is first fed into A. Depending on the types of the output records of A, results are fed to either B or C. The output of the parallel composition $B \| C$ is synchronised, i.e. all output in response to a record fed to B can not be produced on the output of the composition until all responses to all prior records—whether fed to B or C—are produced.

Since events are logged per box, the trace in figure 2.5 is expressed with per-box time-lines. A grey area under a box' time-line depicts an execution of

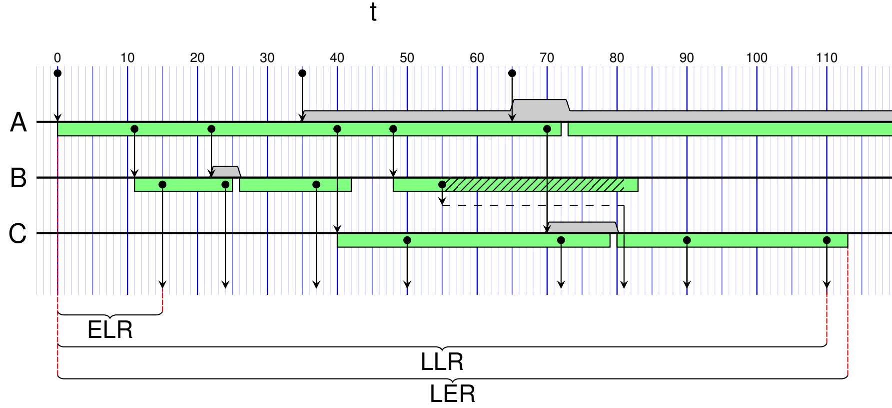


Figure 1.2: Example event trace

that box. The grey areas above the time-lines depict the state (degree of filling) of the corresponding box' input buffer. An arrow from a black solid circle indicates the production of a record from an execution to an input buffer. When a box is not executing and input is available, it can begin an execution. When there is a delay between the availability of input and the start of an execution, therein the scheduler shows through. Scheduling events are themselves not logged and can thus only be observed implicitly from the event traces. If a box is ready to produce a record but is blocked on its output (when either the target buffer is full or, as in the case for B at $t = 55$ in this example, due to a synchronisation constraint, i.e. having to wait for other boxes to finish execution), it is suspended. Blocked executions are shown as diagonally shaded areas (here, for B between $t = 55$ and $t = 81$). As shown in the example, unblocking may not occur instantly upon the output becoming available, but also depends on scheduling.

Because we can reconstruct which execution produced which record and which record triggered which execution, we can establish correspondence between different executions of different boxes. In other words, we can determine the total amount of work incurred by a specific record being input to a network. The record produced at $t = 0$ to A in figure 2.5 triggers the first execution of A, which triggers three executions of B and two of C. At the output of the network, eight records are produced in response to that initial input record. This leads to the following definitions.

Definition 1.1 (Corresponding execution)**.** *For record r, the ith execution of box B, denoted e_i^B, is called an r-*corresponding execution*, iff*

- *r was input to B and triggered e_i^B; or*
- *e_i^B was triggered by record r', which was produced by an r-corresponding execution.*

Definition 1.2 (Corresponding record)**.** *For records r and r', r' is called an r-*corresponding record*, iff it is produced by an r-corresponding execution.*

Definition 1.3 (Earliest Latency Response—ELR)**.** *The earliest latency response of network N to record r is the time between the moment r is fed into N and the moment the* first *r-corresponding record is produced on the output of N.*

Definition 1.4 (Latest Latency Response—LLR)**.** *The latest latency response of network N to record r is the time between the moment r is fed into N and the moment the* last *r-corresponding record is produced on the output of N.*

Determining whether a record is going to be the last record produced by an execution of an opaque box is impossible[1]. Therefore, the LLR can only be determined post-hoc, at the earliest time at the end of a box execution. Since a box execution may produce no records at all, the LLR of a network may be *less* than that of one of its subnetworks.

Definition 1.5 (Latest Execution Response—LER)**.** *The latest execution response of network N to record r is the time between the moment r is fed into N and the moment the last r-corresponding execution of any network element in N terminates.*

The LER of a box is only precisely observable when the assumption holds, that the scheduler can only preempt an execution when this execution gets blocked on I/O. If consecutive executions of a box occur in one scheduler assigned time slot, the start of a new execution is observable by the consumption of a new record from the box' input buffer. If such an aggregated execution of a box can be preempted right after a single execution terminates, but before the next input is read, without logging the event of the termination of the former execution, the LER of any network can be an overestimate.

1.1.1 Buffer delays

The time that records sit idle in buffers inside a network are largely due to waiting times by predecessor records and by resource sharing (scheduling effects). This idling time is aggregated into ELR, LLR and LER. These dependencies are discussed in more detail in section 1.3.

1.1.2 Synchrocells

In the determination of correspondence, synchrocells are treated like boxes. In other words, when records are consumed for synchronisation in a synchrocell, we simply model zero-multiplicity. There is no perceived difference between a record that passes through a synchrocell unchanged and one that is the result of synchronisation.

If future findings indicate that such information is useful for the prediction of latencies, further latency types must be defined. It seems reasonable to assume

[1] In the general case, this is the Halting Problem.

the existence of different classes of persistence of a record in the accumulated state, where one class has all previously accumulated records remain in the state and another can bound (the time of) the influence a record has on a state (e.g. in MPEG, the influence of one Independent Frame is up until the next Independent Frame and the influence of any Progressive Frame is only until the next Independent Frame). However, whether these classes have different predictive value remains to be seen.

1.2 Throughput and jitter

As shown above, individual observations are expressed most naturally in terms of latency. Definitions for throughput are hard to provide such that they are simultaneously meaningful and generally observable in asynchronous dataflow networks. A reasonable definition for throughput is the number of records input (or, adversely, output) over a bound interval T. However, the throughput defined as such is a function of both network properties and *available* input. When networks are not given input, their throughput goes down. Since the distribution of records over routes through asynchronous dataflow networks is not fixed, a meaningful benchmark can not simply be produced. The same problems arise in trying to formulate compositional models of throughput. This is why we do not model throughput explicitly in our statistical analysis.

In *synchronous* dataflow, jitter is often perceived as the variance of latency, where the latency can either be concretely determined or, at least, bound. Since, in our case, latencies are only concrete for individual observations, but generalised to networks by aggregating these observations into distributions of latency, jitter is implicitly captured in those same distributions.

1.3 Dependencies

In the context of the statistical performance model described in this deliverable, a dependency constitutes a correlation between two observable values. As discussed above, the main observations are those of latencies. In this section, we describe different types of dependencies between latencies of different box executions and between those latencies and predictors.

1.3.1 Data/Record dependency

The latency of a box execution e_i^B can be dependent on the (values contained in the) n^{th} input record r to box B. If any r-corresponding executions' latencies of box C are similarly dependent on the (values contained in the) r-corresponding records on its input, then B's latency should (inversely) correlate to that of C.

1.3.2 Stream dependency

Some applications or subnets of applications have (partially) determined structures in their in- or output streams. As a typically example, consider the Independent and Progressive frames (resp. I-frames and P-frames) in MPEG video coding. I-frames are a complete frame, so quite large, but require very little processing. P-frames are 'deltas' from the previous frame, i.e. they describe only what changed from the previous frame. The application dictates, that after an I-frame, there is *always* a P-frame. Typically, the next frame is also, as is the next, etc. The chance of an I-frame occurring immediately after another I-frame, therefore, is nil. For every following frame, the chance of it being an I-frame, rather than a P-frame increases.

Whether a record represents (part of) an I-frame or a P-frame could be part of the type and, thus, influence its routing. In other words, some boxes may be used only for I-frames and some only for P-frames. Alternatively, boxes dealing with both types of frames may still have quite different (latency) behaviour for different types of frames. Thus, this dependency is between consecutive executions of boxes and/or latencies of a (sub)network in response to consecutive records on its input.

1.3.3 State dependency

State dependency is where a box' latency depends on whatever state has accumulated in the application, i.e. the latency of a response to a record depends on the records that came *before* it. As an example, consider applications that perform search-and-track, i.e. they look for a pattern in one sample and—assuming some form of locality in consecutive samples—follow it in consecutive samples. This is a common application pattern in, for example, wireless baseband processing and feature tracking in video.

In terms of the last example, feature tracking in video, the behaviour of search-and-track can be described as follows. Initially, the application has no indication of where the desired feature(s) is in the video frame. Therefore, it searches the entire frame to see whether it can detect the desired feature. When it fails to detect the feature, it searches the next frame. When it succeeds, the assumption is that the feature will reside in roughly the same area in the next video frame. Thus, the application does not need to search the entirety of the next frame, but rather *tracks* the feature through consecutive frames, i.e. it searches with the location from the previous frame as a starting point.

Naturally, tracking is a considerably faster operation than searching. Whether or not the application is currently tracking (i.e. knows where the feature was previously) is not determined by the input, but rather by the accumulated application state.

Other common state dependencies show in (di- or) convergent behaviour. In other words, the latency of (subnets of) an application (diverting from or) converging to a stable (noisy/stochastic) point.

1.3.4 Resource dependency

The sharing of resources is not visible on the VRNET-level. Sharing is determined by the scheduler. However, sharing of, for example, a processor clearly influences latencies. Therefore, resource dependency is between two or more simultaneous (in wall-clock terms) executions of boxes. Since this document deals with static analysis, resource dependency is not treated in the remainder of this text.

1.3.5 Predictors

The dependencies discussed above can be used to relate latency behaviour of different box executions in an application. However, it is sometimes possible to indicate explicitly information that predicts some of this behaviour. This information is not currently observed by the available tooling, but future incarnations of LPEL will have a monitoring system that allows for the logging of the types of records and/or values of their tags.

Logging types is expected to be particularly helpful to determine stream dependencies. Numeric values of record tags can similarly be informative, especially for data dependencies—even if only one box is dependent on this data—but also for state dependencies.

Where an application's performance is insufficiently predicted by the observations provided, specialised *predictor generating boxes* can be introduced. These are normal boxes from a compiler/run-time system point of view, but they have no functional purpose for the application. Rather, they specifically extract properties from records, such as array sizes or heuristic estimations of data complexities, and expose them by adding observable tags to these records.

Chapter 2

Example: Blob analysis

2.1 The application

The application analysed for this example is the Blob Analysis application as described in [28].

2.1.1 Application outline

Image processing in industrial environments, especially in the field of quality inspection, e.g. for the production of foils, industrial woven fabrics, or stainless steel plates for end-consumer devices, has to cope with a complex phenomenology of textures and defects and in addition with real time requirements on high speed installations, e.g. with an achievable scanning speed up to 300 m/min, i.e. about 80 MB/sec per camera system. This requires the application of advanced cost-intensive algorithms for image processing as well as machine learning, the use of high-performance computational hardware like GPUs or multi-core systems, and the exploitation of parallelisation potentials.

Especially in the field of quality inspection, blob analysis is an elementary step in defect detection (module "Candidate Detection" in figure 2.1) to extract features for defect classification, e.g. using support vector machines (module "Defect Classification" in figure 2.1). Furthermore, for huge data sets the statistical evaluation of distinct regions in images is a time consuming task.

One of the ADVANCE visions is to realize both an automatic and optimal distribution of workload on heterogeneous platforms to achieve the best performance gain during run-time, especially if the amount of input data is changing during inspection, e.g. due to varying numbers or sizes of regions of interest (ROIs). To analyse such a workload behaviour we have developed a simple and easy to understand blob analysis tool (see figure 2.2) to simulate an industrial use case.

First, this tool reads images from an image database and preprocesses them applying an anisotropic filter [24, 27]. Second, an "ImageLabeler" is used to

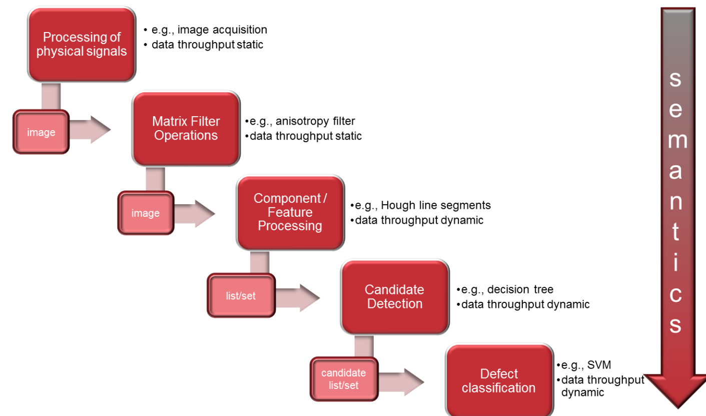


Figure 2.1: General image processing pipeline for quality inspection

identify the regions of interest. These two steps must be executed in sequence only once. In contrast, the statistical blob analysis itself (calculation of perimeter, area, centroid, compactness, and moments) must be performed for each labelled region. Separate labelled regions are mutually independent, so their analyses can be performed in parallel. Since some modules rely on the output of others, e.g. to calculate the compactness, the results of perimeter and area are necessary, synchronization points are needed, depicted as vertical lines in the lower dashed-bordered rectangle in figure 2.2. Finally, the results of all modules are written to the output.

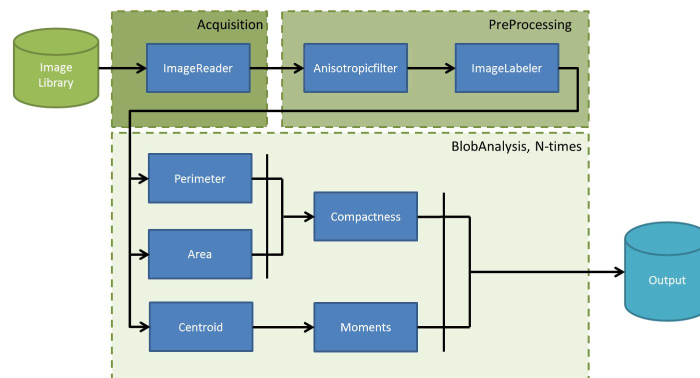


Figure 2.2: Abstract blob analysis tool for statistical performance analysis

The workloads of individual modules as shown in figure 2.2, as well as the workload of the used CPU/GPU units, are not predictable ahead of time. This means that in most industrial use cases the occurrence, the size, and the complexity of such regions of interest have an arbitrary behaviour. Additionally,

the processing times of the single modules in the blob analysis tool differ because of the varied mathematical complexity.

2.1.2 Use Cases

This section gives a brief theoretical introduction on the blob analysis tool and its modules. As shown in Figure 2.2 the tool has a simple design, based on elementary modules, which are commonly used in the field of binary blob analysis. A region of interest R, as shown in figure 2.3, is characterised by different features. One is the perimeter P of R, which is defined as the number of pixels of its contour C. Another is the area A, which is defined as the number of pixels in region R (see figure 2.2 and equations 2.1 and 2.2). To calculate the coordinates x_s and y_s of the centroid S, the pixels in both the x and y directions have to be summed up separately and normalized by A (equations 2.3 and 2.4). The second moments (equations 2.5, 2.6, and 2.7) describe the rotation of the region in the defined direction, where σ_{xx}, σ_{yy}, and σ_{xy} are calculated using the standard deviation. Finally, the compactness K in equation 2.8 defines the roundness of a region, where $K = 1$ denotes a circle and $K > 1$ denotes a line.

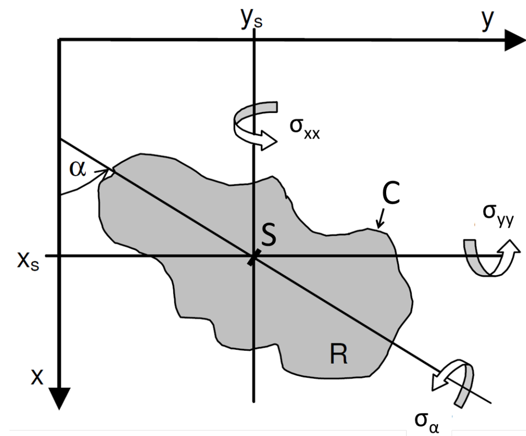


Figure 2.3: blob analysis

$$P = |C| \tag{2.1}$$
$$A = |R| \tag{2.2}$$
$$x_s = \frac{1}{A} \sum_{(x,y) \in R} x \tag{2.3}$$
$$y_s = \frac{1}{A} \sum_{(x,y) \in R} y \tag{2.4}$$
$$\sigma_{xx} = \frac{1}{A} \sum_{(x,y) \in R} (x - x_s)^2 \tag{2.5}$$
$$\sigma_{yy} = \frac{1}{A} \sum_{(x,y) \in R} (y - y_s)^2 \tag{2.6}$$
$$\sigma_{xy} = \frac{1}{A} \sum_{(x,y) \in R} (y - y_s) \cdot (x - x_s) \tag{2.7}$$
$$K = \frac{P^2}{4 \cdot \pi \cdot A} \tag{2.8}$$

2.1.3 S-NET Source Code

The SNET implementation of the blob analysis tool is as follows:

Source code of blob analysis tool using S-NET

```
1  #define REPLICATION_NUMBER 2
2
3  net bloblanalysis
4  {
5    box ImageReader( (imagePath, ⟨imageReplicationCnt⟩) ->
6                    (image, ⟨imageNum⟩) );
7    box PrintStats( (area, perimeter, compactness, moments,
8                    centroidX, centroidY, ⟨blobNum⟩) -> );
9
10   net PreProcessing
11   {
12     box AnisotropicFilter( (image) -> (filteredImage) );
13     box ImageLabeler( (filteredImage, ⟨blobReplicationCnt⟩) ->
14                       (blobList, ⟨blobCount⟩) );
15   } connect
16     (AnisotropicFilter!⟨imageNum⟩) ..
17     (ImageLabeler!⟨imageNum⟩);
18
19   net Analysis
20   {
21     box SplitBlobList( (blobList, ⟨blobCount⟩) -> (blob, ⟨blobNum⟩) );
22     net Properties
23     {
24       box AreaComputation( (blobCalculationCopy) -> (area) );
25       box PerimeterComputation( (blobCalculationCopy) -> (perimeter) );
```

```
26        box CompactnessComputation( (areaCopy, perimeterCopy) →
27                                    (compactness) );
28      } connect
29        [{} → {⟨area⟩}; {⟨perimeter⟩}] ..
30        (
31          ([{⟨area⟩}        → {}] .. AreaComputation      ) ||
32          ([{⟨perimeter⟩} → {}] .. PerimeterComputation)
33        ) ..
34        ([| {area}, {perimeter} |] * {area, perimeter}) ..
35        [{area, perimeter} → {area, perimeter, areaCopy = area,
36          perimeterCopy = perimeter}] ..
37        CompactnessComputation;
38      net Geometry
39      {
40        box MomentsComputation((blobCalculationCopy) → (moments));
41        box CentroidComputation((momentsCopy) → (centroidX, centroidY));
42      } connect
43        MomentsComputation ..
44        [{moments} → {moments, momentsCopy = moments}] ..
45        CentroidComputation;
46    } connect
47      SplitBlobList ..
48      [{blob, ⟨blobNum⟩} → {blob, blobCalculationCopy = blob, ⟨blobNum⟩}] ..
49      (
50        [{blob, blobCalculationCopy, ⟨blobNum⟩} → {blob, blobCalculationCopy,
51          ⟨blobNum⟩, ⟨k = blobNum % REPLICATION_NUMBER⟩}] ..
52        ((
53          [{} → {⟨properties⟩}; {⟨geometry⟩}] ..
54          (
55            ([{⟨properties⟩} → {}] .. Properties) ||
56            ([{⟨geometry⟩}    → {}] .. Geometry   )
57          ) ..
58          ( [| {area, perimeter, compactness},
59               {moments, centroidX, centroidY} |] *
60               {area, perimeter, compactness, moments,
61               centroidX, centroidY} )
62        ) ! ⟨k⟩)
63      );
64  } connect ImageReader .. PreProcessing .. Analysis .. PrintStats;
```

As depicted in figure 2.2 the tool consists of several modules (blue boxes) which are connected either sequentially or in parallel according to their dependencies to the results of other modules. Each of these modules corresponds to a box definition (e.g. ImageLabeler in lines 5–6) in the SNET network. Those modules/boxes hold the sequential code for image reading, filtering, labelling and calculating properties of blobs. Furthermore the boxes are grouped into networks the same way as the modules in figure 2.2.

The boxes `AnisotropicFilter` and `ImageLabeler` build up the `Preprocessing` network and are connected sequentially inside of this net. The output of this net is a list of all blobs found in the given image. The `Analysis` network consists of all boxes responsible for analysing a blob. This network is split into a `Properties` network, which calculates the area, perimeter and compactness of a blob, and a `Geometry` network, to compute the moments and centroid. This way those two sub networks can easily be connected for parallel execution, because those

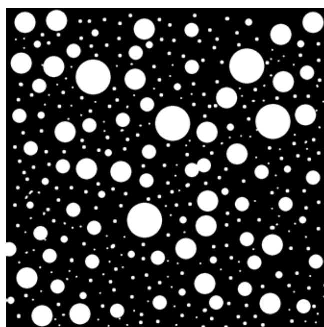


Figure 2.4: Test image used

values do not rely on each other. Inside of the `Properties` network the boxes for area and perimeter computation are also connected parallel. The output of the `Analysis` network is the complete set of properties of one blob.

After each parallel execution of networks or boxes a synchronization box is placed to synchronize the computed properties of a box, so that values of different blobs do not get intermixed. The box `SplitBlobList` is an additional box between preprocessing and analysis, which takes the list of blobs produced by the ImageLabeler box and writes the single blobs to the output stream. Since those blobs are independent they can be analyzed in parallel. This is done using the parallel replication combinator '`!`', which replicates the whole Analysis network dynamically during runtime. The maximum replication number is defined by `REPLICATION_NUMBER` (defined in the line 1 of the listening).

2.1.4 Execution

For this example, we used one image (see figure 2.4), to illustrate the different types of latency described above. The different-sized blobs cause variations in workload incurred by boxes. Furthermore, to exploit the nature of a stream processing pipeline and to simulate the continued acquisition and inspection process of a real-world application the image is processed five times in a loop.

The benchmarks demonstrates the different execution times of different boxes using SNET. The measurements were performed on a Sony VAIO™ PCG-81112M with an Intel® Core™ i7-740QM processor, 8 GB RAM and a NVIDIA GeForce GT 425M graphics card. The operating system was Ubuntu 10.10.

2.2 The execution trace

We want to establish the latency between `SplitBlobList` and the network output. In this application, all boxes between `SplitBlobList` and the network output have no multiplicity, i.e. for every record they consume, they produce precisely one output record. This means that the ELR and the LLR are the same.

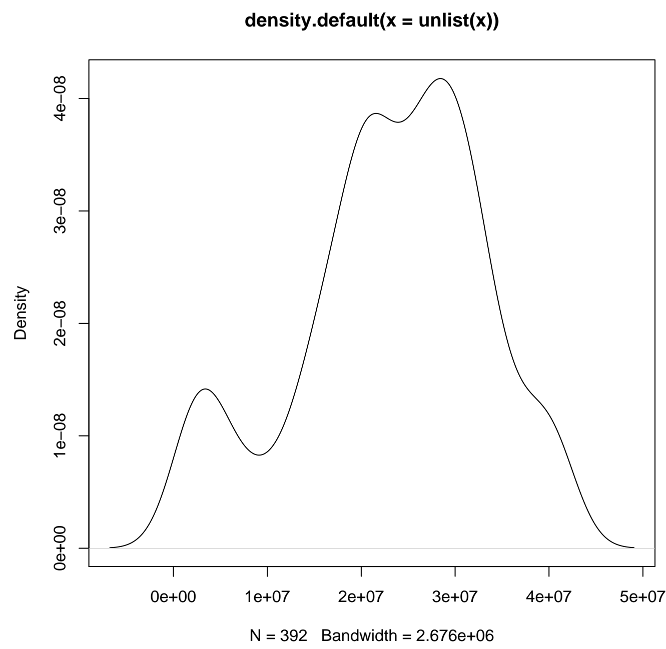


Figure 2.5: Histogram of latencies between `SplitBlobList` and the network output.

The histogram depicted in figure 2.5 gives us the measured latency in nanoseconds. This granularity of measurement is not very reliable, considering we ask the Linux kernel for wall-clock time in nanoseconds, for which the accuracy is known to be undefined. Because of this hard-to-estimate accuracy, we resort to determining the histogram through kernel density estimation. This explains why the histogram suggests execution times below zero. The consequences of such estimation artifacts for our further analysis are future work. For sufficiently large data sets, we treat this histogram as the probability density function for the latency.

Chapter 3

Statistical performance analysis of synchronous dataflow graphs

Synchronous dataflow (SDF) [19] is a paradigm to model deterministic stream processing applications. The SDF model is decidable, i.e., questions regarding deadlock-freeness may be answered by analysing the model.

In the SDF paradigm, a streaming application is modelled by an SDF *graph* (see Figure 3.1). In this graph, vertices (called *actors*) represent tasks (at a certain granularity) and edges (called *channels*) represent data dependencies between these tasks. Channels may be marked with multiple *tokens*, which resolve cyclic data dependencies (these cycles may for example model stateful computations, where tokens signify an initial state, or first-in-first-out buffers, where tokens signify the buffer's capacity).

Dataflow graphs are typically used to model real-time embedded systems, where issues such as worst-case timing analysis need to be addressed at design-time. In worst-case timing analysis, each actor is assigned a worst-case execution time, using which the graph's worst-case makespan (or its inverse, throughput) may be computed. In this chapter we address the case where the execution time of an actor is non-deterministic, but is in fact assumed to be sampled

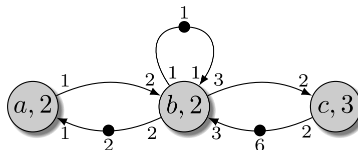


Figure 3.1: Example SDF graph

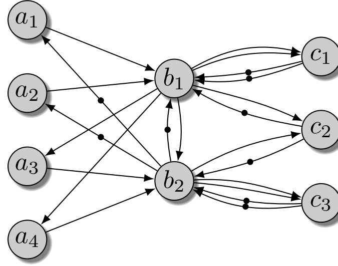


Figure 3.2: Example equivalent homogeneous SDF graph of the example graph shown in Figure 3.1. Note that this graph is a multigraph.

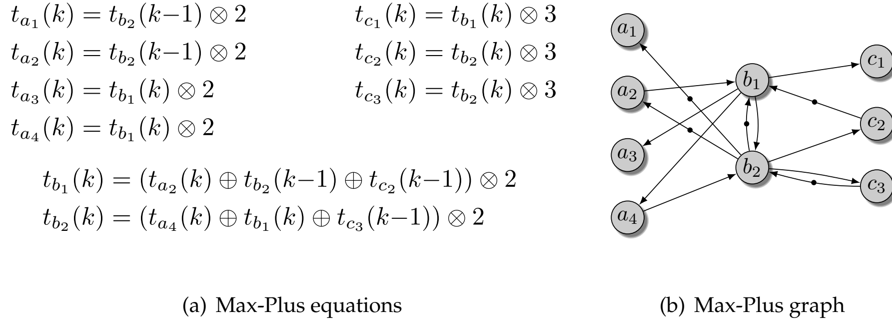
from a stationary, ergodic probability distribution. We furthermore assume that no interdependencies between actor execution time probability distributions exist. As will become clear in the following sections, even under these naive assumptions performance analysis of SDF graphs is prohibitive.

3.1 Timing Analysis

Timing analysis of SDF graphs aims at finding performance characteristics such as latency (or latencies) and throughput. A straightforward approach to timing analysis of SDF graphs is by *simulating* a *self-timed execution* of the graph, which involves firing an actor as soon as it is enabled. Whenever actor execution times are deterministic (e.g., in worst-case timing analysis), however, a mathematical analysis is more suitable. Mathematical analysis requires the SDF graph to be transformed into a graph in which all production and consumption rates are one. The resulting graph is commonly called a *homogeneous* SDF (HSDF) graph and is essentially a *timed event graph*. Synchronisation and duration of events in this graph may be elegantly described using *max-plus algebra*. In case execution times are non-deterministic, the same algebra may be applied. Obtaining the (non-deterministic) throughput or latencies, however, is for the non-deterministic case, even under the most naive assumptions, not straightforward. The following details will discuss the application of Max-Plus algebra to SDF graphs and non-deterministic timing analysis in more detail.

3.1.1 Max-Plus algebraic system model

If one considers the time at which actors fire for the kth time as the state of a system, then one may view the times at which these actors fire for the $(k+1)^{\text{th}}$ time as a *state transition*. For a homogeneous SDF graph, we may describe the



(a) Max-Plus equations

(b) Max-Plus graph

Figure 3.3: Max-Plus equations for the SDF graph shown in Figure 3.1, and its graphical representation.

evolution of actor firing times with a linear, time-invariant system:

$$x(k+1) = \bigoplus_{i=0}^{M} A_i \otimes x(k-1), \tag{3.1}$$

where M is the order of the system. Consider for example the HSDF graph depicted in Figure 3.3(b), which is the homogeneous representation of the SDF graph shown earlier in Figure 3.1. This HSDF graph may be described by a system of recurrent max-plus expressions of order 1 (the order is simply the maximum number of tokens on a single channel in the graph).

The higher-order system (3.1) may (if the graph is *deadlock-free*) on its turn be transformed in the following first-order time-invariant system:

$$x(k+1) = A \otimes x(k). \tag{3.2}$$

The *eigenvalue* of matrix A in (3.2) is equal to the *makespan* (the duration of a single iteration) of the graph, and the eigenvalue's multiplicative inverse is the graph's *throughput*. Efficient methods (see for example [6], [8], [29]) exist to determine the eigenvalue of matrix A.

Note that modelling an SDF graph by a linear, time-invariant max-plus system requires the SDF graph to be homogeneous (an SDF graph may be described by a linear, *time-variant* max-plus system). If this is not the case, the SDF graph needs to be transformed into an HSDF graph, which in the worst case may result in an exponentially larger graph.

3.1.2 Lyapunov exponent

Whenever the entries a_{ij} in matrix A in (3.2) are stochastic variables, the eigenvalue will be stochastic as well. The non-deterministic analogue of the eigenvalue is the *Lyapunov exponent*. The numerical value of the Lyapunov exponent is the fastest rate at which the system can operate *on average*.

Apart from straightforward simulation of the Lyapunov exponent several methods are proposed in the literature to characterise the Lyapunov exponent. These methods are based on approximating the Lyapunov exponent either by Taylor series [2] or by simulating its bounds [13].

3.2 Computational issues

Dataflow graphs where actors have (stationary, independent) probabilistic execution times have been studied by Olsder et al. [23]. The authors show that the behaviour of such a system may be described by a discrete-time Markov chain, with a finite state space. The average throughput can then be derived from the Markov chain's stationary distribution. The problem of this analysis, however, is the potentially huge size of the state space of the Markov chain. Even for unrealistically simple stochastic models, computation of exact solutions is prohibitive.

When actor execution times are modelled by *exponential* distributions, then the system can be analysed using continuous-time Markov chains [20]. This approach exploits the memorylessness of the exponential distribution. The number of states for such a continuous-time Markov chain is equal to the number of reachable token configurations of the dataflow graph. Unfortunately, the number of reachable token configurations can again be exponential in the size of the dataflow graph. Furthermore, assuming exponentially distributed execution times for actors in a dataflow graph is generally a rather unrealistic simplification and will hence yield inaccurate results.

Chapter 4

Statistical methods: copulas and vines

Our aim is to produce statistical bounds on execution time for S-Net applications, and we are attempting to do this by combining statistical information about the performance of individual components. We are using statistical methods related to multivariate probability distributions, in particular the theory of *copulas*. In this chapter we will describe the statistical background and discuss possible applications to S-Net.

4.1 Multivariate distributions and copulas

A multivariate probability distribution gives a description of several random properties of a single individual. There are discrete and continuous versions of multivariate distributions, but we shall be concerned exclusively with the latter here.

A continuous *joint distribution* or *multivariate distribution* can be thought of as a sequence of $(X_1, \ldots, X_d)$ continuous random variables on some probability space, or equivalently as a *random vector* associating elements of $\mathbb{R}^d$ with members of a sample space. Analogously to single-variable distributions, a multivariate distribution has a (*cumulative*) *distribution function* (CDF)

$$F(x_1, \ldots, x_d) = P(X_1 \leq x_1, \ldots, X_d \leq x_d) \tag{4.1}$$

and, under suitable continuity assumptions, a *probability density function* (PDF)

$$f(x_1, \ldots, x_d) = \frac{\partial^d F}{\partial x_1 \cdots \partial x_d}$$

so that

$$P(a_1 \leq X_1 \leq b_1, \ldots, a_d \leq X_d \leq x_d) = F(b_1, \ldots, b_d) - F(a_1, \ldots, a_d)$$
$$= \int_{a_d}^{b_d} \cdots \int_{a_1}^{b_1} f(x_1, \ldots, x_d)\, dx_1 \cdots dx_d \tag{4.2}$$

See [14] or [9] for instance for more information on the basic theory of multivariate distributions.

In recent years *copulas* have become a popular tool in the study of multivariate distributions. There is a considerable literature on copulas: [22] and [17] are standard references.

Definition 1. Let $d \in \mathbb{N}$. A d-dimensional copula (or *d-copula*) is a multivariate distribution on the unit cube in $\mathbb{R}^d$ with uniform marginal distributions.

□

More concretely, a 2-copula is a function $C : [0,1] \times [0,1] \to [0,1]$ such that

(i) $C(u,0) = C(0,v) = 0$ for all $u, v \in [0,1]$

(ii) $C(u,1) = u, C(1,v) = v$ for all $u, v \in [0,1]$

(iii) If $(u_1, v_1), (u_2, v_2) \in [0,1] \times [0,1]$ with $u_1 \leq u_2$ and $v_1 \leq v_2$ then

$$C(u_2, v_2) - C(u_2, v_1) - C(u_1, v_2) + C(u_1, v_1) \geq 0$$

The definition of a d-copula for $d > 2$ can be stated in similar terms, but with a considerably more complex version of (iii).

The significance of copulas is that they capture the dependency between the individual variables of a multivariate distribution and allow us to express the entire distribution in terms of the marginal distributions and a copula. A precise statement is given by

Theorem 1 (Sklar's Theorem)**.** *Let H be a d-dimensional distribution function with marginal distributions $F_1, \ldots, F_d$. Then there exists an d-copula C such that for all* $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$,

$$H(x_1, \ldots, x_d) = C(F_1(x_1), \ldots, F_d(x_d))$$

Proof. See [22, Theorem 2.3.3] □

Sklar's Theorem allows us to use copulas to model multivariate distributions. Suppose we have a set of samples $\{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ drawn from some d-dimensional distribution. Let $\pi_j : \mathbb{R}^d \to \mathbb{R}$ be the j-th coordinate projection; then $S_j = \{\pi_j(\mathbf{x}_i) : 1 \leq i \leq k\}$ is a sample drawn from the j-th marginal distribution. We can now model the marginal distributions F_j either by fitting some specific distribution function to the data (for example, we may have reason to believe that the marginal data is normally distributed, and then we can use the mean

and variance of S_j as approximations to the parameters of the distribution), or by using the empirical cumulative distribution function.

In order to model the entire multivariate distribution it now suffices to find a copula which matches (or at least approximates) the copula given by Sklar's Theorem. Subsequent sections will address this problem.

4.1.1 2-copulas

In the 2-dimensional case copulas are fairly well understood, and computational methods have been developed for dealing with them.

There are copulas derived from certain well-known distributions, such as the *normal copula* (or *Gaussian copula*) and the *Student t-copula*. These are useful for studying data which is appropriately distributed, but for more general situations *Archmidean copulas* are popular.

Theorem 2. *Suppose that $\phi : [0,1] \to [0,\infty)$ is a continuous, strictly decreasing, convex function. If we put*

$$C(u,v) = \phi^{[-1]}(\phi(u) + \phi(v))$$

where

$$\phi^{[-1]}(t) = \begin{cases} \phi^{-1}(t) & \text{if } 0 \leq t \leq \phi(0) \\ 0 & \text{if } t > \phi(0) \end{cases}$$

then C is a 2-copula, called the Archmidean copula with generator ϕ.

It is easy to find functions ϕ with the properties necessary to be a generator, and this allows the construction of copulas modelling a wide range of dependency properties. Furthermore, entire *families* of generators ϕ_θ can be found, where θ is a parameter lying in some interval such as $[0,\infty)$; these lead to corresponding parametric families of copulas. There is a large variety of such families: for example, Nelsen [22, Table 4.1] gives a list of 22 one-parameter families; there are also families with two parameters.

There are techniques which aid in the selection of an appropriate copula family to model a sample of empirical data, and others (eg, maximum likelihood estimation) which allow one to select a parameter θ giving the member of a family which has the best fit to a sample: see [11, 12, 15] for example. Techniques of this type have been implemented in statistical packages such as **R** [25], making practical application quite feasible.

4.1.2 Archmidean copulas in higher dimensions

In higher dimensions, the situation is less well understood. Archimedean copulas still exist, with a similar definition:

$$C(u_1,\ldots,u_d) = \phi^{[-1]}(\phi(u_1) + \cdots + \phi(u_d))$$

(for suitable ϕ), but the commutativity of addition in $\mathbb{R}$ implies that

$$C(u_1, \dots, u_d) = C(\pi(u_1), \dots, \pi(u_d))$$

for any permutation π of $\{1, \dots, d\}$. This introduces a high degree of symmetry which makes Archimedean copulas unsuitable for realistic modelling of complex high-dimensional distributions.

4.2 Pair-copula Decompositions and Vine Structures

An approach to studying higher-dimensional distributions which has been the subject of considerable research in recent years is to decompose the distribution hierarchically using 2-copulas. The structure of the decomposition is managed by using a graphical structure known as a *vine*. In order to explain this, we first require some definitions.

4.2.1 Marginal and conditional distributions

Suppose that $f(x_1, \dots x_d)$ is the density function of some d-dimensional continuous probability distribution. Let $\{i_1, \dots, i_k\}$ be some subset of $\{1, \dots, d\}$ and let $\{j_1, \dots, j_l\} = \{1, \dots, d\} \backslash \{i_1, \dots, i_k\}$. The *marginal distribution* $f_{i_1 \cdots i_k}$ is the k-dimensional distribution with density function

$$f_{i_1 \cdots i_k}(x_{i_1}, \dots, x_{i_k}) = \int \cdots \int f(x_1, \dots, x_d)\, dx_{j_1} \cdots dx_{j_l}$$

where the integrals are over the whole of $\mathbb{R}$.

If $f_{i_1 \cdots i_k}(x_{i_1}, \dots, x_{i_k}) \neq 0$ then the *conditional distribution* $f_{j_1, \dots, j_l | i_1, \dots i_k}$ is defined by

$$f_{j_1, \dots, j_l | i_1, \dots i_k}(x_{j_1}, \dots, x_{j_l}) = f(x_1, \dots, x_d) / f_{i_1 \cdots i_k}(x_{i_1}, \dots, x_{i_k}) \quad (4.3)$$

For fixed values of $x_{i_1}, \dots, x_{i_k}$, $f_{j_1, \dots, j_l | i_1, \dots i_k}(x_{j_1}, \dots, x_{j_l})$ describes the distribution of $(j_1, \dots, j_l)$. The process of forming the conditional distribution $f_{j_1, \dots, j_l | i_1, \dots i_k}$ is sometimes referred to as *conditioning on* $\{x_{i_1}, \dots, x_{i_k}\}$, or just *conditioning on* x_i if only one variable is involved.
Rearranging (4.3), we have

$$f(x_1, \dots, x_d) = f_{i_1 \cdots i_k}(x_{i_1}, \dots, x_{i_k}) f_{j_1, \dots, j_l | i_1, \dots i_k}(x_{j_1}, \dots, x_{j_l}) \quad (4.4)$$

For example, in $\mathbb{R}^5$ we have

$$f_{145}(x_2, x_3) = \iiint f(x_1, x_2, x_3, x_4, x_5)\, dx_1 dx_4 dx_5$$

and

$$f_{23|145}(x_2, x_3 | x_1, x_4, x_5) = f(x_1, x_2, x_3, x_4, x_5) / f_{145}(x_1, x_4, x_5).$$

4.2.2 Copula densities

Since a d-copula C is just a special type of d-dimensional cumulative distribution function, it has (under suitable differentiability assumptions which will apply in all the cases which we will consider) an associated density function, the *copula density* c, given by

$$c = \frac{\partial^d C}{\partial x_1 \cdots \partial x_d}.$$

Sklar's Theorem can be rephrased in terms of copula densities.

Theorem 3 (Sklar's Theorem for density functions)**.** *Let g be the density function of a d-dimensional distribution with margins $F_1, \ldots, F_d$ and corresponding marginal density functions $f_1, \ldots, f_d$. Then there exists a d-copula density c such that for all $(x_1, \ldots, x_d) \in \mathbb{R}^d$,*

$$f(x_1, \ldots, x_d) = c(F_1(x_1), \ldots, F_d(x_d)) f_1(x_1) \cdots f_d(x_d).$$

Proof. This follows from the Chain Rule for partial derivatives. □

4.2.3 Pair-copula decompositions

The ingredients described above allow us to decompose an arbitrary d-copula into a product of 2-copulas and univariate marginal distributions. This construction originated in [16]. The general process is fairly complex and we will give an illustrative example rather than attempting a complete explanation.

A simple corollary of Theorem 3 will be helpful.

Lemma 1. *Suppose $f_{ij}(x_i, x_j)$ $(i \neq j)$ is a bivariate distribution. Then there exists a 2-copula c_{ij} such that*

$$f_{i|j}(x_i|x_j) = c_{ij}(F_i(x_i), F_j(x_j)) f_i(x_i) \tag{4.5}$$

and

$$f_{j|i}(x_j|x_i) = c_{ij}(F_i(x_i), F_j(x_j)) f_j(x_j) \tag{4.6}$$

Proof Theorem 3 provides a 2-copula c_{ij} with

$$f_{ij}(x_i, x_j) = c_{ij}(F_i(x_i), F_j(x_j)) f_i(x_i) f_j(x_j)$$

By definition,

$$f_{i|j}(x_i|x_j) = f_{ij}(x_i, x_j)/f_j(x_j),$$

and this gives (4.5). Similarly, the definition of $f_{j|i}$ leads to (4.6). □

Example Consider the density function $f(x_1, x_2, x_3, x_4)$ of a four-dimensional distribution with cumulative distribution function $F(x_1, x_2, x_3, x_4)$. Using (4.4) repeatedly we can write

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= f_{124|3}(x_1, x_2, x_4|x_3) f_3(x_3) \\ &= f_{12|34}(x_1, x_2|x_3, x_4) f_{4|3}(x_4|x_3) f_3(x_3) \\ &= f_{2|134}(x_2|x_1, x_3, x_4) f_{1|34}(x_1|x_3, x_4) f_{4|3}(x_4|x_3) f_3(x_3) \end{aligned} \tag{4.7}$$

whenever all of the conditional distributions are defined.

We have chosen to decompose f here by successively conditioning on x_3, x_4 and x_1. This is an arbitrary choice: there are many other ways we could decompose f.

Now Lemma 1 provides a 2-copula c_{34} with

$$f_{4|3}(x_4|x_3) = c_{34}(F_3(x_3), F_4(x_4)) f_4(x_4)$$

and substituting this into (4.7) gives

$$f(x_1, x_2, x_3, x_4) = f_{2|134}(x_2|x_1, x_3, x_4) f_{1|34}(x_1|x_3, x_4) c_{34} f_3(x_3) f_4(x_4).$$

(We< have omitted the arguments of c_{34} for brevity, and will use similar abbreviations in the sequel.)
Now we will decompose $f_{1|34}$. First, consider $f_{13|4}$. Using Theorem 3 again we obtain a 2-copula $c_{13|4}$ (more precisely, this is a family of 2-copulas paramaterised by x_4) such that

$$f_{13|4}(x_1, x_3|x_4) = c_{13|4}(F_{1|4}(x_1|x_4), F_{3|4}(x_3)|x_4) f_{1|4}(x_1|x_4) f_{3|4}(x_3|x_4))$$

it is easy to see that $f_{1|34}(x_1|x_3, x_4) = f_{13|4}(x_1, x_3|x_4)/f_{3|4}(x_3|x_4)$, so we obtain

$$f_{1|34}(x_1|x_3, x_4) = c_{13|4}(F_{1|4}(x_1|x_4), F_{3|4}(x_3|x_4)) f_{1|4}(x_1|x_4). \tag{4.8}$$

another application of Lemma 1 gives a 2-copula c_{14} with

$$f_{1|4}(x_1|x_4) = c_{14}(F_1(x_1), F_4(x_4)) f_1(x_1)$$

so that (4.8) becomes

$$f_{1|34}(x_1|x_3, x_4) = c_{13|4} c_{14} f_1(x_1)$$

Substituting into (4.7), we now have

$$f(x_1, x_2, x_3, x_4) = f_{2|134}(x_2|x_1, x_3, x_4) c_{14} c_{34} c_{13|4} f_1(x_1) f_3(x_3) f_4(x_4) \tag{4.9}$$

We can similarly decompose $f_{2|134}$ by rewriting it in terms of $f_{12|34}$ and $f_{1|34}$ and then using Theorem 3 again (three times) to replace bivariate conditional

distributions with 2-copulas. The details are left to the reader, but eventually we obtain

$$f_{2|134}(x_2|x_1,x_3,x_4) = c_{12|34}c_{23|4}c_{24}f_2(x_2),$$

which we can substitute into (4.9) to get

$$f(x_1,x_2,x_3,x_4) = f_1(x_1)f_2(x_2)f_3(x_3)f_4(x_4)\cdot c_{14}c_{24}c_{34}c_{13|4}c_{23|4}c_{12|34}. \quad (4.10)$$

As claimed above, we have been able to decompose the 4-dimensional distribution f into a product of univariate marginal distributions and 2-copulas.
It is clear that we have made a number of choices in this construction. Other choices are possible, and these give other decompositions. For example, in (4.8) we used $f_{1|4}$ to obtain $f_{1|34} = c_{13|4}c_{14}f_1$, but we could also have used $f_{1|3}$ to obtain a decomposition $f_{1|34} = c14|3c_{13}f_1$, and proceeding with the rest of the calculation as before would yield

$$f(x_1,x_2,x_3,x_4) = f_1(x_1)f_2(x_2)f_3(x_3)f_4(x_4)\cdot c_{13}c_{24}c_{34}c_{14|3}c_{23|4}c_{12|34}. \quad (4.11)$$

4.3 Vines

The computation in the previous section demonstrates that arbitrary d-dimensional distributions can be expressed in terms of 2-copulas, but it is clear that the construction can be quite complex and that many choices can be made. There are also other complications which are not apparent in our example: for example, care must be taken to avoid cyclic dependencies between variables. In order to manage this complexity, Bedford and Cooke [3, 4] introduced a graphical structure called a *vine*.

Definition 2. Let $d \in \mathbb{N}$. A *regular vine tree* (or *R-vine*) on d variables is a sequence $T_1,\dots,T_d$ of trees such that

1. T_i has nodes N_i and edges E_i with $|N_i| = d+1-i$ (and hence $|E_i| = d-i$).
2. For $i = 2,\dots,d-1$ a bijection is specified between the nodes N_i of T_i and the edges E_{i-1} of T_{i-1}.
3. If two nodes in T_{i+1} are joined by an edge then the corresponding edges in T_i share a common node.

Note 1 We have described a vine as a sequence of trees. Another (equivalent) approach is to regard the bijection in condition (2) as an identity map: in this view, a vine is a tree with a hierarchical set of edges, an edge on level $k+1$ joining two edges on level k. This helps to explain the term “vine”, but leads to diagrams which are difficult to draw and interpret.

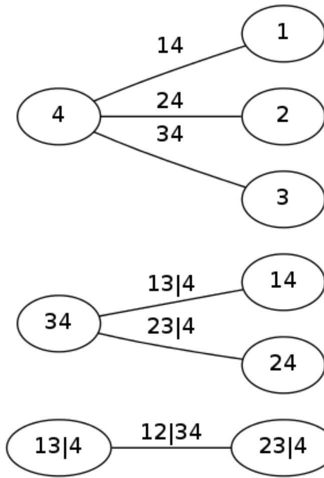


Figure 4.1: C-vine on 4 variables

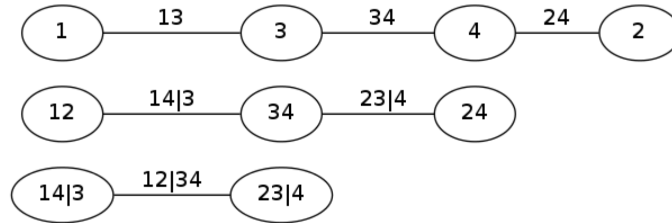


Figure 4.2: D-vine on 4 variables

Note 2 Note that condition (3) (the *proximity condition*) does not say "if and only if". This means that the tree T_{k+1} is not necessarily uniquely determined by T_k: for a given T_k there may be multiple possibilities for T_{k+1}.

C-vines and D-vines

There are two classes of vines which have a particularly regular structure: *C-vines* (*canonical* vines), which consist of a sequence of star-shaped trees, and *D-vines* (*drawable* vines) which consist of a sequence of linear trees. Figure 4.1 shows a C-vine on 4 variables and Figure 4.2 shows a D-vine on 4 variables. In Figure 4.3 we display an R-vine on 7 variables. The labels on these diagrams will be explained in the following section.

Labelled vine structures

The significance of vines is that they can be used to describe the strategy used to decompose copulas into products of 2-copulas; this enables one to generate

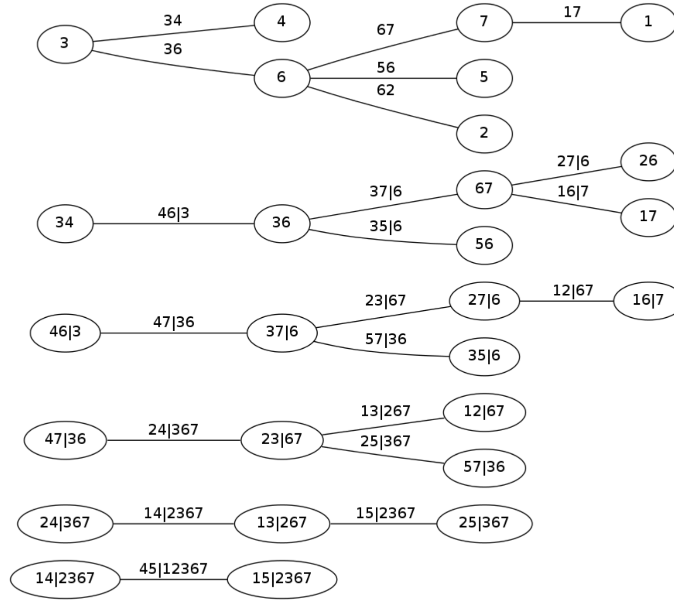


Figure 4.3: R-vine on 7 variables

pair-copula decompositions quite easily from vine diagrams, without having to perform complicated algebraic manipulations.

Given a d-dimensional distribution f and a vine structure on d variables, it is quite straightforward to write down a pair-copula decomposition for f. We do this by inductively labelling the nodes and edges of the trees T_k comprising the vine, starting at $k = 1$. The edges of T_k will be labelled with symbols $ij|D$ with $i < j$ and $|D| = k - 1$. When $k = 1$, $D = \emptyset$ and we just write ij instead of $ij|\emptyset$.

- The d nodes of T_1 are labelled with the integers $1, \dots, d$ in some order. An edge between nodes labelled i and j ($i < j$) is labelled with the symbol ij.
- Once the nodes and edges of T_k have been labelled, the nodes and edges of T_{k+1} are labelled as follows:
  - Condition (2) in the definition of a vine says that the nodes of T_{k+1} correspond bijectively to the edges of T_k. We label the nodes in T_{k+1} simply by copying the labels from the corresponding edges in T_k
  - Now suppose that we have an edge e between nodes labelled $i_1 j_1|D_1$ and $i_2 j_2|D_2$ in T_{k+1}. Condition (3) in the definition of a vine requires that the corresponding edges in T_k must share a node in T_k. Put $V_1 = \{i_1, j_1\} \cup D_1$ and $V_2 = \{i_2, j_2\} \cup D_2$. We label the edge e with

the symbol $ij|D$, where

$$\begin{aligned} D &= D_1 \cap D_2 \\ i &= \min((V_1 \cup V_2) \setminus D) \\ j &= \max((V_1 \cup V_2) \setminus D) \end{aligned}$$

The vine structures in Figures 4.1, 4.2 and 4.3 have already been labelled in this way.

Remark 1 Note that for a given vine structure, the only nondeterministic step in the process above is the initial labelling of the nodes in T_1.

Remark 2 It can be shown that we always have $|(V_1 \cup V_2) \setminus D| = 2$, so the $\max$ and $\min$ operations above are expressing the complement of D in the form $\{i, j\}$ with $i < j$.

Remark 3 Every vine structure on d variables has precisely $\binom{d}{2} = d(d-1)/2$ edges, which correspond to 2-copulas covering all possible dependencies between distinct pairs of variables.

Deriving pair-copula constructions from labelled vines

Once we have a suitably labelled vine structure, it is easy to write down a pair-copula decomposition for a d-dimensional distribution with joint density function $f(x_1, \ldots, x_d)$: we simply simply write down the product consisting of the marginal distributions $f_1 \ldots f_d$ and 2-copulas c_ℓ for all labels ℓ of edges in the labelled vine. There will always be $d(d-1)/2$ such copulas.

The labellings in Figures 4.1 and 4.2 give the decompositions

$$f = f_1 f_2 f_3 f_4 \cdot c_{14} c_{24} c_{34} c_{13|4} c_{23|4} c_{12|34}$$

and

$$f = f_1 f_2 f_3 f_4 \cdot c_{13} c_{24} c_{34} c_{14|3} c_{23|4} c_{12|34}$$

derived earlier. Figure 4.3 gives a decomposition

$$\begin{aligned} f = f_1 f_2 f_3 f_4 f_5 f_6 f_7 &\cdot c_{17} c_{26} c_{34} c_{36} c_{56} c_{67} \\ &\cdot c_{16|7} c_{25|6} c_{27|6} c_{37|6} c_{46|3} \cdot c_{12|67} c_{23|67} c_{47|36} c_{57|36} \\ &\cdot c_{13|267} c_{24|367} c_{25|367} \cdot c_{14|2367} c_{15|2367} \cdot c_{45|12367} \end{aligned}$$

for a 7-dimensional distribution. (The $\cdot$ symbols represent multiplication, as does juxtaposition, and are included merely to make the structure of the product more apparent.)

4.4 Inference

The previous sections describe how to decompose a given d-variable distribution into a product of marginal distributions and 2-copulas, but the problem remains of how to *fit* such a decomposition to a set of empirical data. Suppose we have a sample drawn from some multivariate distribution given by a vector of random variables $(X_1, \ldots, X_d)$. There are three steps involved in fitting a vine-copula model.

- Select a suitable vine structure.
- For each of the $d(d-1)/2$ edges (i, j) in the vine, choose a 2-copula family to represent the dependency between X_i and X_j.
- Estimate parameters to find members of the copula families which give the best fit to the data.

The first step presents some difficulty since there are a very large number of possible vine structures. Ii is shown in [21] that in dimension d the number of C-vines is equal to $d!/2$, as is the number of D-vines. The number of R-vines is

$$\frac{d!}{2} \, 2^{\binom{d-2}{2}}. \tag{4.12}$$

For $d = 10$, this number is 1814400×2^{28}, which is approximately 4.87×10^{14}.

A number of methods have been proposed for choosing a suitable vine structure. In small dimensions (perhaps up to $d = 5$, where the number of vines is 480) it is feasible to examine all possible vine structures and select to one with best fit, but this is clearly out of the question in higher dimensions. In some cases a simple vine structure may be suitable: for example, if there is reason to believe that there is a single variable whose influence dominates the distribution then a C-vine would be a reasonable choice. A more general method is to consider measures of correlation between pairs of variables and build the top level of the vine from the most strongly correlated pairs of variables, with more weakly correlated pairs appearing in lower levels [10]. In many cases the conditional correlation in later levels may be very weak, allowing one to truncate the vine structure and use independence copulas in the lower levels of the vine [5]. Several other methods have been proposed.

There are also various methods for selection of suitable copula families: for example one can use graphical tools to examine the fit of data to various families of copulas, and there several methods which can be used to fit members of copula families to data and test the goodness of fit; one can apply these to (for example) the Archimedean copula families mentioned above to select a copula giving the best description of the data.

Several methods for model and copula selection have been implemented in the **R** package `CDVine` of Schepsmeier and Brechmann [26]. As the name suggests, this deals only with C- and D-vines. Inference for general R-vines is both less well-developed and technically more difficult, but development

of **R** code is currently ongoing, and a package for R-vines will reportedly be available later in 2012.

The problem of modelling and inference of R-vines is currently a very active area, and up-to-date references to both the theory and applications to real-world data can be found at `www-m4.ma.tum.de/en/research/vine-copula-models/`. See also [7] and [1] for an overview.

4.5 Application to S-Net

Our aim is to apply the theory of copulas and vines to analyse and predict the statistical behaviour of S-Net applications. Our approach is to observe the execution times of individual components and then to use copulas to analyse the joint distribution of the interconnected components. The basic justification for using statistical methods is that in real applications, execution time is not deterministic: statistical variations are introduced by many factors: for example, interference from other processes being executed concurrently, I/O delays and cache delays. A further possible source of nondeterministic behaviour is non-uniformity of data: for example, the execution time of an image-processing application may vary according to the level of detail in the image (a medical image, say), and it is a reasonable assumption that the level of detail in candidate inputs will be probabilistically distributed.

4.5.1 Composing subnetworks

Suppose we have an S-Net network N composed of sub-networks $N_1, \ldots, N_d$. For simplicity, we will consider the case where the N_i are individual components. For every record r input to N and for each i we measure the total t_i of the latest latency responses (1.4) for all r-corresponding executions (1.1) of the N_i, and this gives us a vector $(t_1, \ldots, t_d)$. We can also measure the LLR for the composite network N to obtain an overall execution time t. In general t will be strongly correlated with $(t_1, \ldots, t_d)$, but there will be some extra statistical noise contributed by factors such as buffering delays.

There are at least two strategies available to us.

1. Observe the distribution of $(t_1, \ldots, t_d)$ and attempt to use the structure of the network (expressed in terms of S-Net combinators) to predict the distribution of t.

2. Observe the distribution of $(t_1, \ldots, t_d, t)$ of the execution times of components together with the overall execution time, then model the entire distribution and examine that in order to derive properties of the overall distribution.

Strategy 1 is the more powerful method, since it allows prediction of overall system behaviour from observations of individual components. The ideal outcome would be that we would be able to examine models of several different

proposed S-Net implementations (with different network structures) and predict which implementation gives the best performance. However, this may be a rather optimistic vision. As the SDF examples in Chapter 3 demonstrate, statistical variations in the flow of records through a network can give rise to somewhat unpredictable global behaviour. This suggests that overall network behaviour may be determine from the behaviour of individual components: for example, if we have a network of a given shape, small perturbations in the behaviour of a single box (perhaps due to varying implementations of the box code) may cause large changes in the performance of the complete application. An appropriate choice of copula families may be able to accommodate this type of behaviour, but there is a danger that the resulting predictions may include so much uncertainty that the analysis is too imprecise to be useful.

Strategy 2 is less powerful, since it requires us to observe the overall behaviour of a *specific* S-Net implementation instead of predicting the behaviour solely from the properties of the components; however, it does have the advantage that we are able to fit models which take account of actual network behaviour: this allows to combine the behaviour of individual components using copula families which allow sufficient room to take account of emergent network behaviour as well. We hope that this approach will allow us to develop models which provide information about the extreme behaviour of the distribution of the overall execution time: this will potentially allow us to make statements about how long an unusually "bad" execution of a network will take, and how likely such bad behaviour is.

4.5.2 Modelling in R

We are currently examining synthetic models of small S-Net applications in order to evaluate these strategies, using vine copula methods (currently restricted to C- and D-vines) to model multivariate distributions. All analysis is done using **R**, since this is freely available and already has suitable packages for performing the type of analyses we require. In an industrial-strength implementation of out methods it might be desirable to have fast optimised implementations of specific statistical routines, but we believe that **R** is ideal for proof of concept since it enables us to explore complex analysis techniques (and to visualise their output) with minimal implementation effort.

We are optimistic that knowledge of S-Net network structures and possibly also information supplied by the programmer (in the CAL language described in [18]) will provide sufficient information about dependencies between box execution times to enable us to select suitable members of the vast space of possible vines (see Equation 4.12) to construct accurate statistical models and scale up from small artificial examples to realistic S-Net applications.

Bibliography

[1] Kjersti Aas, Claudia Czado, Arnoldo Frigessi, and Henrik Bakken. Pair-copula constructions of multiple dependence. *Insurance Mathematics and Economics*, 44(2):182–198, 2009.

[2] François Baccelli and Dohy Hong. Analytic Expansions of Max-Plus Lyapunov Exponents. *The Annals of Applied Probability*, 10(3):pp. 779–827, 2000.

[3] Tim Bedford and Roger M. Cooke. Probability density decomposition for conditionally dependent random variables modeled by vines. *Annals of Mathematics and Artificial Intelligence*, 32(1-4):245–268, 2001.

[4] Tim Bedford and Roger M. Cooke. Vines - a new graphical model for dependent random variables. *Ann. Statist*, 30:1031–1068, 2002.

[5] E. C. Brechmann, C. Czado, and K. Aas. Truncated regular vines in high dimensions with application to financial data. *Canadian Journal of Statistics*, 40(1):68–85, 2012.

[6] Jean Cochet-Terrasson, Guy Cohen, Stéphane Gaubert, Michael Mc Gettrick, and Jean-pierre Quadrat. Numerical Computation of Spectral Elements in Max-Plus Algebra, 1998.

[7] Claudia Czado. Pair-copula constructions of multivariate copulas.

[8] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):385, 2004.

[9] F. M. Dekking, C. Kraaikamp, H. P. Lopuhaä, and L. E. Meester. *A Modern Introduction to Probability and Statistics*. Springer-Verlag, 2010.

[10] Jeffrey Dißmann. Statistical inference for regular vines and application. Master's thesis, Technische Universität München, 2010.

[11] V Durrleman, A Nikeghbali, and T Roncalli. Which copula is the right one? Working paper, Groupe de Recherche Opérationnelle. Crédit Lyonnais. France, 2000.

[12] Michiels F. and De Schepper A. A new graphical tool for copula selection. Working papers, University of Antwerp, Faculty of Applied Economics, May 2010.

[13] Rob M.P. Goverde, Bernd Heidergott, and Glenn Merlet. A fast approximation algorithm for the Lyapunov exponent of stochastic max-plus systems. In *2008 9th International Workshop on Discrete Event Systems*, pages 49–54. IEEE, 2008.

[14] Robert V. Hogg, Allen Craig, and Joseph W. McKean. *Introduction to Mathematical Statistics, 7th Edition*. Pearson, 2012.

[15] David Huard, Guillaume Évin, and Anne-Catherine Favre. Bayesian copula selection. *Computational Statistics & Data Analysis*, 51(2):809–822, 2006.

[16] H. Joe. Families of m-variate distributions with given margins and $m(m-1)/2$ bivariate dependence parameters. In Berthold Schweizer Ludger Rüschendorf and Michael D. Taylor, editors, *Distributions with fixed marginals and related topics*. Institute of Mathematical Statistics, 1996.

[17] H. Joe. *Multivariate Models and Dependence Concepts*. Monographs on Statistics and Applied Probability. Chapman & Hall, 1997.

[18] Raimund Kirner. ADVANCE project deliverable D4: Report on performance annotations/interfaces, 2010.

[19] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245.

[20] M.K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE Transactions on Computers*, C-31(9):913–917, September 1982.

[21] Oswaldo Morales-Nápoles. *Bayesian belief nets and vines in aviation safety and other applications*. PhD thesis, Technische Universiteit Delft, 2010.

[22] Roger B. Nelsen. *An Introduction to Copulas, 2nd Edition*. Springer, 2006.

[23] G. J. Olsder. Performance analysis of data-driven networks. pages 33–41, October 1990.

[24] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:629–639, 1990.

[25] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[26] Ulf Schepsmeier and Eike Christian Brechmann. *CDVine: Statistical inference of C- and D-vine copulas*, 2011. R package version 1.1-5.

[27] Volkmar Wieser, Clemens Grelck, Holger Schoener, Peter Haslinger, and Bernhard Moser. GPU-based Image Precessing Use Cases: A High-Level Approach. In *Advances in Parallel Computing*. IOS Press, 2011.

[28] Volkmar Wieser, Philip K.F. Hölzenspies, Raimund Kirner, and Michael Roßbory. Statistical Performance Analysis with Dynamic Workload using S-NET. Technical report, November 2011. FD-COMA 2012: Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures.

[29] N Young, R Tarjan, and J Orlin. Faster Parametric Shortest Path and Minimum Balance Algorithms. *ArXiv Computer Science e-prints*, May 2002.