

An Integrated CSP-Based Tool for the Visualisation, Animation and Performance Evaluation of Message Passing Algorithms

Ali E. Abdallah
Centre for Applied Formal Methods
School of Computing, Information and Mathematics
South Bank University
103 Borough Road
London, SE1 0AA, U.K.
Email: A.Abdallah@sbu.ac.uk

Mark Green
The University of Reading
Department of Computer Science
Reading, RG6 6AY, U.K.
Email: Mark.Green@reading.ac.uk

Abstract

This paper presents ongoing research and development on an integrated tool for the visualisation and animation of message-passing communicating systems described in Hoare's CSP (Communicating Sequential Processes). It introduces major new developments to the original VisualNets implemented in C++ and reported in [4]. The new tool is implemented partly in Java and partly in the functional programming language Haskell. Not only does the new tool enjoy greater expressive power and a better user interface, it has new capabilities for aiding the user in animating, analysing, and reasoning about CSP specifications. These includes provisions for nested parallelism within a single node, improved profiling and animation, and the possibility of applying generic timing cost models for performance evaluation.

The paper briefly describes the new features, explains the methods by which they have been implemented and illustrates their use with examples.

Keywords: Communicating Sequential Processes, visualisation, animation, performance evaluation, message-passing algorithms.

1 Introduction

It is known that the use of formal specifications can greatly facilitate the process of system design. Properties of the system may be rigorously proven based on a specification, and the process of constructing the specification can highlight areas in which consideration is required. Automatic animation of formal specifications enables the behaviour of the specified system to be conjectured prior to implementation, enabling the specification to

be tested, validated and becoming increasingly accessible to non-technically oriented end-users.

This paper presents a tool for the visualisation and animation of specifications written in Hoare's CSP notation [9, 14], used for specifying networks of parallel processes that communicate via a message-passing model. The process of developing correct and efficient message passing algorithms is known to be difficult and error prone, and the notation provides a means of formally resolving the complexity introduced by composing processes in parallel, which leads to the need for synchronization, communication and non-determinism. Properties such as specification satisfaction and deadlock freedom can be formally proven, and parallel systems can be constructed from functional specifications using formal refinement rules [1, 3, 2]

The tool generates and animates a visual representation of the process network designed by the user. Visualisation is a valuable aid to the design, analysis and understanding of concurrent systems, and is used by a number of systems in the parallel programming and distributed computing fields ([11, 15] for example). An appropriate visualisation of a parallel system provides an intuitive and clear display of information that may not be obvious from the textual specification details of the system, such as the exact structure of the network and arrangement of connecting channels.

The tool presented here is a greatly enhanced version of the VisualNets tool introduced in [4]. The tool has been completely rewritten since the previous version to considerably improve the range of specifications accepted and enable it to visualise existing specifications in textual forms (the original version allowed only visual creation of networks). The new implementation is written partly in Haskell and partly in Java. It is intended to develop this tool further into an industrial-strength network visualisation application.

ht

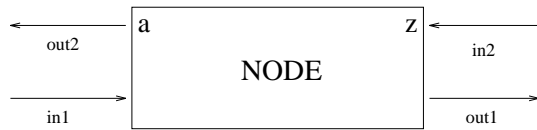


Figure 1. A single node of the multiplication pipeline.

2 General overview of the tool

This section will provide an overview of the functionality of the tool.

2.1 Specification structure visualisation

The tool's functionality is based on a visualisation of the communication and synchronisation structure present in a formal CSP specification. This visualisation is automatically constructed based on the user's input.

The user's specification is entered using a hybrid notation referred to as CSP_F , similar to the CSP_M syntax used by FDR [6]. This notation greatly extends the CSP syntax used by the original version, which supported only prefixing, input and output, prefix choice, and conditionals; the parallelism operator was not supported since all parallelisms had to be defined using the tool's own visual editor. The new version adds support for sequencing, parallelism and nested parallelism, interleaving, hiding, substitution, feeding and pipelining, and interruption, together with the Timed CSP operators: Wait, timeout, and timed prefix. Additionally, the notation supports a wide range of functional expressions, which may be used for performing computation within the defined CSP processes. Functions may also be defined that operate on CSP processes or on sets of CSP processes; by writing functions that input sets of CSP processes and apply the CSP parallelising operators to them as appropriate, it is possible to create skeletons that create standard network structures from user-specified sets of processes.

To illustrate the tool's functionality, consider the CSP formulation [9] of a parallel algorithm for long multiplication of two numbers represented as lists of digits, expressed with number base b , shown below:

$$\begin{aligned}
 NODE(a, z) = & \text{in1?}x \rightarrow \text{out2!}(a \times x + z) \bmod b \rightarrow \\
 & \text{out1!}x \rightarrow \text{in2?}y \rightarrow \\
 & NODE(a, y + (a \times x + z) \text{div } b)
 \end{aligned}$$

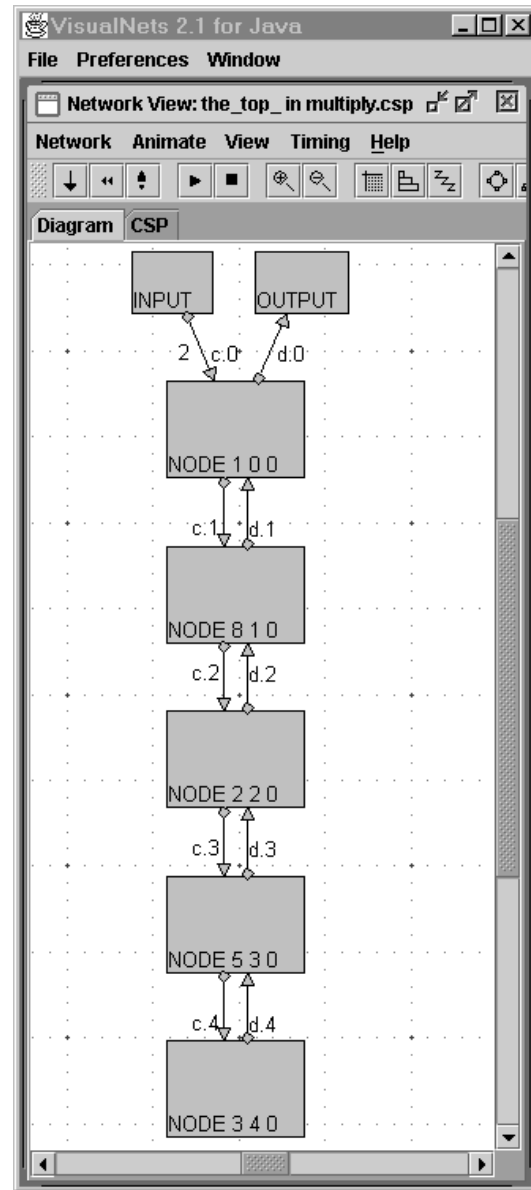


Figure 2. A visualisation of a cyclic multiplication pipeline.

$$N(n) = c_n ? x \rightarrow d_n ! (m_n \times x + z) \bmod b \\ \rightarrow N(m_n, (m_n \times x + z) \text{div } b)$$

$$N(k) = NODE(m_k, 0) [c_k / \text{in}1, c_{k+1} / \text{out}1, \\ d_k / \text{out}2, d_{k+1} / \text{in}2]$$

$$NETWORK = || / [N(k) \mid k \leftarrow [0..n]]$$

Note that the notation $|| / [P_1, P_2, ..P_n]$ stands for the parallel composition of all the processes in the list $[P_1, P_2, ..P_n]$; the notation $P[\text{new}/\text{old}]$ stands for renaming the channel *old* of the process *P* to be *new*.

The above formulation captures a doubly linked pipe network of communicating processes. Each digit of the multiplier $[m_0, m_2, .., m_n]$ resides in one node of the network. The original input comes in, digit by digit, on c_0 and propagates to the right on the *c* channels. The partial answers are propagated leftward on the *d* channels and the final answer is output on channel d_0 . The rightmost node does not have connections to the right. Each node can be pictured as a box with four channels as shown in Figure 1. The graphical visualisation and configuration of the entire network is automatically derived from the textual CSP_F form (shown in Appendix 1) as a dual pipeline of processes as shown in Figure 2. Communication channels common to pairs of processes are represented by links connecting the appropriate boxes.

2.2 Animation

The user is able to directly edit the text of the specification from within the tool. It is worth noting that, unlike the previous version of the tool, each box on the visualisation is not necessarily a sequential process: they may contain internal parallelisms which are not normally displayed on the visualisation. The user is provided with zooming functions to view more detailed internal parallelism.

Once the structure visualisation has been constructed, the user is able to observe the behaviour of the specified network by animating the visualisation.

Animation is based on the interface used by the previous tool; a coloured indicator appears at each end of each line on the visualisation. The colour of the indicator shows the status of the process it is nearest to with respect to the channel represented by the line. The user may advance the state of the network through time by clicking a button. By observing the change in the colours of the indicators as the network runs, the behaviour of the specification can be observed.

A green indicator indicates that the process is ready to communicate on the channel; a red indicator indicates it is

	Current	Last	High	Low	Average	Total
c.0	1	3	3	3	3	30
c.1		0	2	0	0	2
c.2		0	4	0	0	4
c.3		0	6	0	0	6
c.4		2	8	2	2	24
d.0		3	3	1	2	31
d.1		0	0	0	0	0
d.2		0	0	0	0	0
d.3		0	0	0	0	0
d.4		0	0	0	0	0

Figure 4. An automatically generated waits information chart.

not. Where communication takes place, the direction of the communication and value communicated are indicated by an arrow and an indicator on the connecting line; events synchronisations are indicated in the same way without the arrow or value indicator.

If a process encounters a choice which is not resolved by parallel synchronisation, yellow indicators appear on the channel or event lines representing the options, and the user may resolve the choice. (The original version of the tool lacked this capacity and depended on any choices being resolved by parallel synchronisation.) In this way, the user can exercise control over the behaviour of the network, and check it behaves correctly under different conditions.

If the user needs further information on the behaviour or state of the network, the variable environments of the processes may be displayed, either in separate windows or within the process boxes themselves. The user may also view a window displaying the specification of a process with the currently active clause highlighted; this enables the user to identify the location of flaws in their specification.

2.3 Performance analysis

Calculating the predicted performance of a parallel network specification, and providing information based on which we can reason about the performance, can be an extremely valuable function. The level of efficiency which a network will perform is very rarely obvious from the specification and even a very small change to the specification may have a significant impact on the performance of the network. Detecting performance issues before implementation time can significantly reduce development difficulties. The tool automatically gathers performance information as the

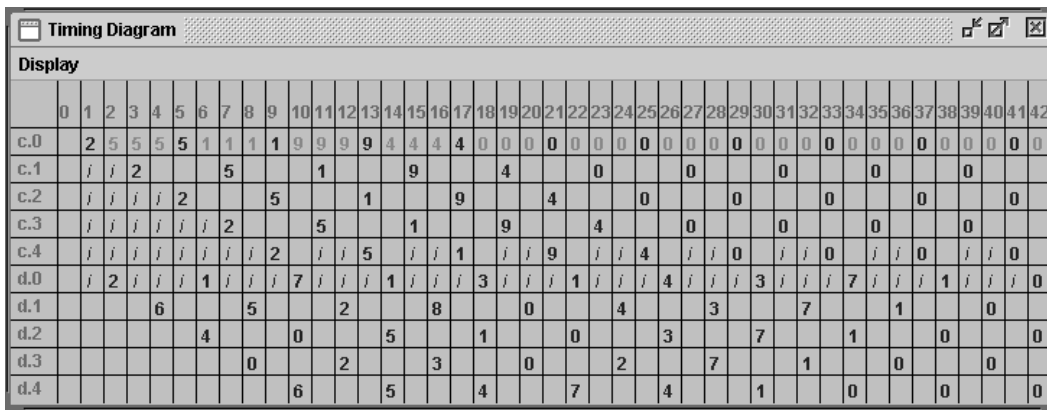


Figure 3. An automatically generated timing diagram.

network is animated, and the user may call up performance information displays at any point during the animation.

The user may view the CSP Timing Diagram of the network, as in Figure 3. This diagram shows all event synchronisations and channel communications that have occurred in the animation of the network so far, and the times at which they occurred. This can be used for performance analysis as well as a useful record of how the network behaved. The timing diagrams produced by the tool have also been enhanced such that they may, at the user's option, show the times at which *failed* attempts to synchronise or receive or transmit messages were made, which can be valuable in detecting bottlenecks or determining the cause of deadlock.

The timing diagram in Figure 3 shows the behaviour of the pipelined multiplication network in the case where the *M* list (the multiplier) is [1, 8, 2, 5, 3], which corresponds to the number 35281, since the least significant digit appears first. The value for multiplication, 49152, can be read as the sequence of values on the *c.0* channel as sent into the pipeline by the *INPUT* process; the result, 1734131712, can be read as the sequence of values on the *d.0* channel as received from the pipeline by the *OUTPUT* process, again with the least significant digit appearing first.

The new version of the tool can also produce a breakdown of the amount of time processes in the network spend waiting for synchronisation, shown in Figure 4. This is a diagram referring to the whole network, rather than to a specific process; hence, a wait is counted whenever one process is trying to communicate on a channel and the other process is not prepared to synchronise. Note that time waiting is not counted simply as time between communications, but only as time when a process is attempting to communicate and cannot immediately obtain synchronisation to do so.

The majority of waiting is on the channels *c.0* and *d.0* as these are connected to the *INPUT* and *OUTPUT* processes respectively; these processes *constantly* attempt to

add new data to the pipeline or retrieve completed results. *c.4* also has a higher wait value than the others, since the *N* process at the end of the pipe has a shorter cycle than the processes within the pipe, and hence begins to attempt to input more rapidly after returning a result. The "highest wait" values on the *c* channels occur during the initial distribution of data when the pipeline is initialising.

Similarly, the user may view a barchart or numerical display of the amount of time processes are spending waiting as against communicating with external processes or dealing with internal communications. Although this information can be derived mathematically from the formal specification, doing so is a time-consuming process and becomes considerably more complex as processes are added to the network. By making use of the tool, performance analysis can be greatly simplified.

2.4 Timing model application

The performance model described above, though useful, is limited by the fact that the actual parallel architecture upon which the specified system is implemented may affect the performance in numerous ways. For example, processors may be arranged in clusters such that intercluster communication may be slower than intracluster communication; the time for a communication may vary with loads on the network; processors and channels may need time to initialise; and different processors may run at different speeds.

It is intended that the tool will also allow a user to construct a timing model for a network architecture and then apply this model to a CSP specification, creating a timed-CSP specification representing that network running on that particular architecture.

A timing model is composed of a set of channel classes and a set of processor classes. The user can assign any channel to be of a specific class, and any process to be running on

a processor of a specific class. When the timing model is applied, appropriate timed CSP Wait statements are added to the specification to represent the timing information added based on the model. This new specification may then be animated or reasoned about by the user to understand the likely real-world behaviour of their specified network; they may animate their specification under several different implementation models in order to establish which would be most suitable for the implementation.

3 Implementation of the tool

3.1 Network specification and animation

The former version of the tool was written entirely in C and used only a simplistic parser. In the new version, the functional language Haskell[10] has been employed for handling calculations related to the structure and state of the network. The CSP_F notation is based on the Haskell language with the CSP operators added and some altered syntax rules to make the syntax more similar to that of standard CSP; CSP_F is read as "CSP (functional)". When the tool is invoked, the user's CSP_F specification is preprocessed into Haskell and the resulting program is combined with a library which provides Haskell definitions of the CSP operators. This method enables the functional flexibility of the Haskell language to be used within the CSP_F notation; it is this flexibility that enables the use of complex functional expressions and skeletons.

The state of a process at any given point is modelled by a Haskell structure containing several parts: the CSP menu (the list of all sets of parallel events that may occur in the next time step), the animation function which inputs one of those sets and returns the state of the network after those events have occurred, and several structures holding information about the derived structure of the network. These structures are constructed by the Haskell library functions that represent the CSP operators.

The visualisation is created based on the structure of a particular process nominated by the user as representing the entire network. The user may visualise the parallel structure of any process they have defined; if the processes within the network themselves include parallelism, the structure of these processes can be visualised in the same way as that of the top level network.

3.2 Network visualisation

The network visualisation and user interfacing is managed by a front end written in the Java programming language[7]. Java was chosen because of its user interfacing capabilities and ease of portability; its platform-independence allows the tool to run on both UNIX and

PC platforms without modification. Interfacing Haskell and Java required the construction of a separate library, the details of which are discussed in [8].

The user's specification is initially accepted by the front-end which preprocesses and compiles it within the Haskell environment, and then constructs the visualisation based on the structure information contained within the Haskell structure representing the initial state of the network. The Java program deals with laying out and presenting the visualisation. When animation is requested, the Java program updates the colours of the channel status indicators based on the sets of events offered by the appropriate processes; whenever a time step is requested, the Java front-end invokes the Haskell system to calculate the state of the network in the new time step.

The Java front-end also deals with gathering performance statistics, which are accumulated based on the state of the network at each time step. The front-end uses these statistics to generate the timing diagram and wait information displays.

3.3 Timing model application

Timing model application is at present still under development; what is discussed here is what has been developed so far.

Processor classes at present are not implemented. Channel classes may be defined using Haskell (to maintain flexibility). A channel class is represented by a structure containing three functions provided by the user. These functions are called to establish the number of time steps taken to perform certain operations on the channel. The operations are: the time it takes for a channel to be ready to input once an input event is encountered; the time it takes for a channel to be ready to output once an output event is encountered; and the time it takes for a particular message to be transmitted on the channel. When a message is transmitted on the channel, the channel may react by changing class after the message has been transmitted, enabling some level of dynamic timing to be implemented (for example, a channel may take a long while to transmit the first message due to setting up, and then may be faster for future messages).

4 Case studies and applications

4.1 Network animation

The animation functions of the tool can be used to examine the structure and functionality of a CSP specification. This may bring to light errors in the specification or flaws in the underlying design of the parallel system and suggest means by which they can be corrected.

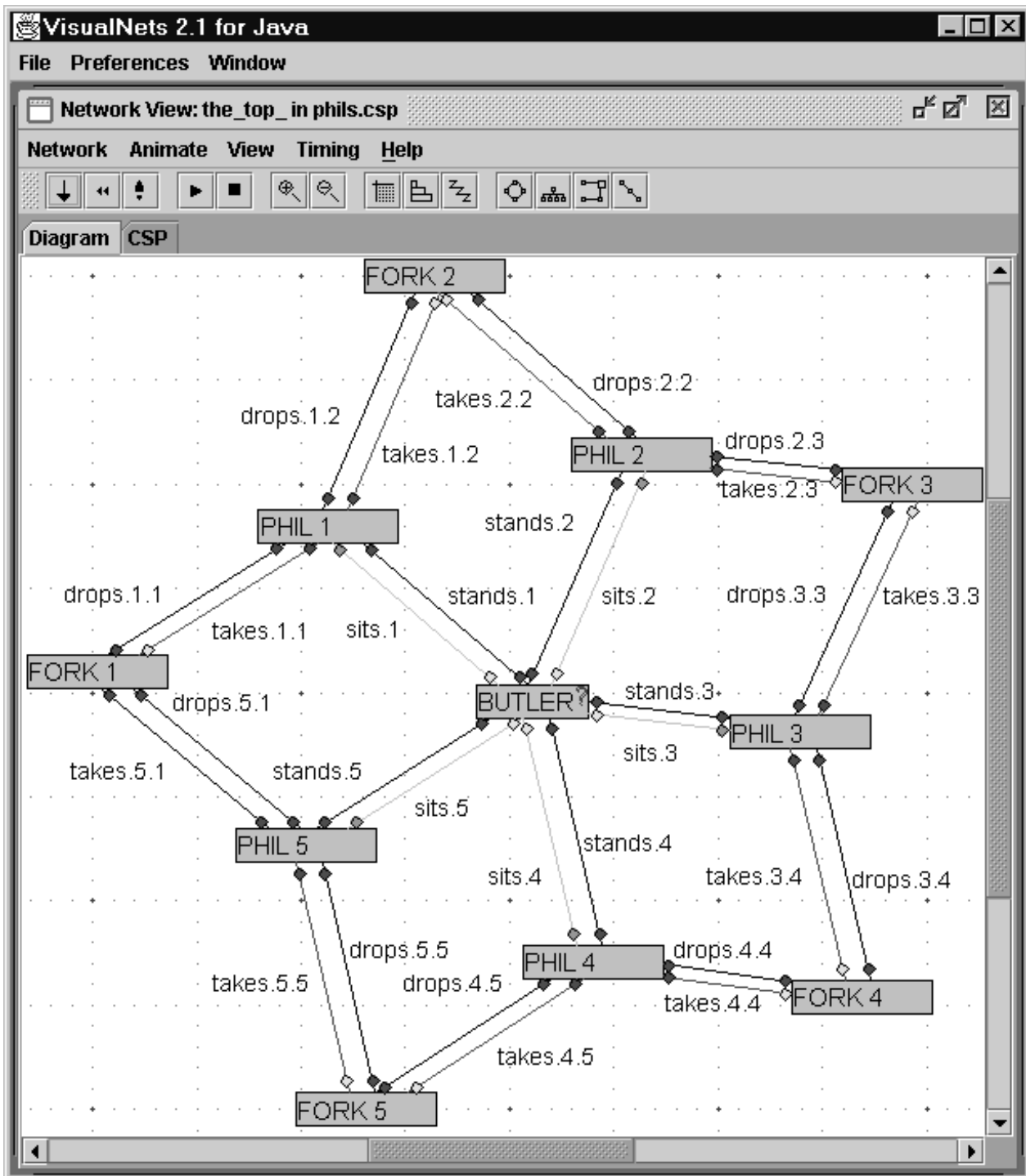


Figure 5. Visualisation of the Dining Philosophers.

```

n = 5
inc x = (x+1) `mod` n
dec x = (x-1) `mod` n

alpha PHIL x = [sits.x, stands.x, eats.x, takes.x.x,
               takes.x.(inc x), drops.x.x, drops.x.(inc x)]
PHIL x = sits.x -> takes.x.x -> takes.x.(inc x) -> eats.x ->
         drops.x.x -> drops.x.(inc x) -> stands.x -> PHIL x

alpha FORK x = [takes.x.x, drops.x.x, takes.(dec x).x, drops.(dec x).x]
FORK x = (takes.x.x -> drops.x.x -> FORK x)
         | (takes.(dec x).x -> drops.(dec x).x -> FORK x)

alpha BUTLER = [sits.x | x <- [0..(n-1)]] ++ [stands.x | x <- [0..(n-1)]]
BUTLER = BUTLER 0
BUTLER 0 = foldr1 (|) [sits.x -> BUTLER 1 | x <- [0..(n-1)]]
BUTLER 4 = foldr1 (|) [stands.x -> BUTLER 3 | x <- [0..(n-1)]]
BUTLER y = (foldr1 (|) [sits.x -> BUTLER(y+1) | x <- [0..(n-1)]]
           | (foldr1 (|) [stands.x -> BUTLER(y-1) | x <- [0..(n-1)]]))

TOP = (foldr1 (||) [PHIL x | x <- [0..(n-1)]] || BUTLER ||
      (foldr1 (||) [FORK x | x <- [0..(n-1)]]))

```

Table 1. The script for the Dining Philosopher problem

To illustrate these functions, consider a solution to the well-known Dining Philosophers problem, presented in [9]. The problem of deadlock is solved by the addition of a “butler” process. The CSP_F specification of the network is described in Table 1. Note the use of the higher order function capabilities of CSP_F to simplify the definitions of the BUTLER and TOP processes. The visualisation produced is shown in 2.4. The structure of the philosophers’ interaction is clearly visible, and the user may step the network through time, observing the interaction in progress. The user may resolve any choices not resolved by the parallelism in the network: hence, the user chooses the order in which the butler seats the philosophers, and should two philosophers attempt to grasp the same fork at the same time, the user can choose who gets it.

If the deadlocking form of the network is used (as above, but with the butler process removed), then the deadlock condition is clearly visible after a single animation step; every philosopher has a green indicator on the *takes* link to the fork on their right, indicating their attempt to lift the fork, but the fork shows a red indicator on this link, showing green only on the *drops* indicator of the philosopher on the right of the work, indicating that it is being held by that philosopher.

4.2 Performance analysis

To illustrate the tool’s capacity for performance analysis, together with the earlier observation that small changes in a network specification can have a dramatic effect on performance, we shall consider the multiplication network presented previously. If we change the order of events in the NODE process as follows:

$$\begin{aligned}
 NODE(a, z) &= in1?x \rightarrow out1!x \rightarrow in2?y \rightarrow \\
 &out2!(a \times x + z) \bmod b \rightarrow \\
 &NODE(a, y + (a \times x + z) \text{div} b)
 \end{aligned}$$

The network will still produce the same results; however it will take considerably more time to do so. Viewing the timing diagram in Figure 6 reveals that the network has become wholly sequential.

Each process is required to receive two inputs before it may return its computed output value. Since each process in the pipeline can provide only one output - the current multiplier value passed along the pipeline - no process can return its value until the value passed down has reached the end node, which is the only node capable of returning an output having received only a single input.

Conversely, using the following redefinition:

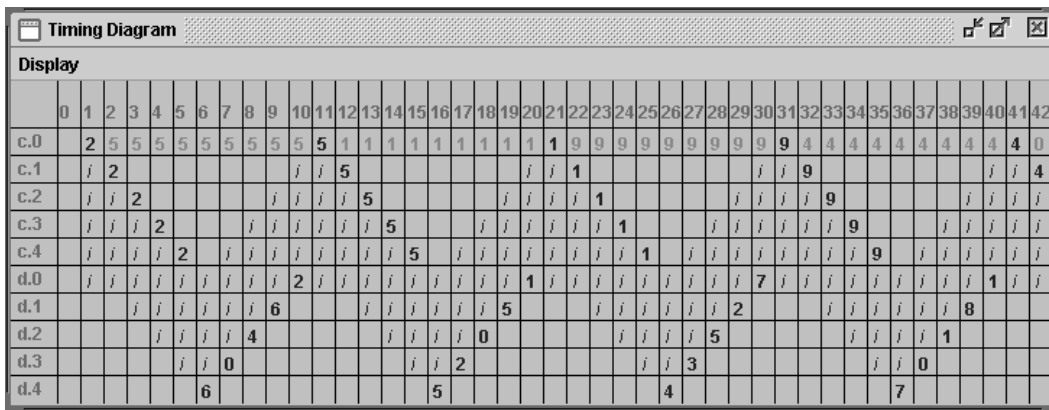


Figure 6. Timings for the sequential multiplication “pipeline”.

$$\begin{aligned}
 NODE(a, z) = & \text{in1?}x \rightarrow \text{out1!}x \rightarrow \\
 & \text{out2!}(a \times x + z) \bmod b \rightarrow \text{in2?}y \rightarrow \\
 & NODE(a, y + (a \times x + z) \div b)
 \end{aligned}$$

Will also produce the same output results. The timing diagram of this network (Figure 7) shows a different pattern to the previous network; this network passes its input value down the pipeline before returning its computed result. The effect of this is that the network takes one extra time step to begin returning results, and then begins returning at the same speed as the previous process.

Examining the waits profile (Figure 8) confirms that the average waiting times for *INPUT* and *OUTPUT* on the *c.0* and *d.0* channels respectively are the same. However, the high waits on the other *c.* channels have been reduced: each process no longer has to wait for the process before it in the pipeline to return its initial output before receiving its own initial input. The effect of *N*'s shorter cycle has also changed: *N* waits, on average, one time step less for its inputs (on *c.4*), since they are transmitted earlier in the cycle: however, it then is required to wait to send its outputs (on *d.4*) since the *NODE* process to which it is connected is initially busy sending its own output back up the pipeline.

This information may not be immediately obvious from the specification, but by the use of visual animation it can be made immediately apparent at specification time without needing any further implementation of the system.

4.3 Education and presentation

The format of the visualisations produced is such that no experience of the CSP notation is required to understand it,

	Current	Last	High	Low	Average	Total
c.0	1	3	3	3	3	30
c.1		0	1	0	0	1
c.2		0	2	0	0	2
c.3		0	3	0	0	3
c.4		1	4	1	1	13
d.0	3	3	3	2	2	29
d.1		0	0	0	0	0
d.2		0	0	0	0	0
d.3		0	0	0	0	0
d.4	1	1	1	1	1	9

Figure 8. Waits information for the modified version of the multiplication pipeline.

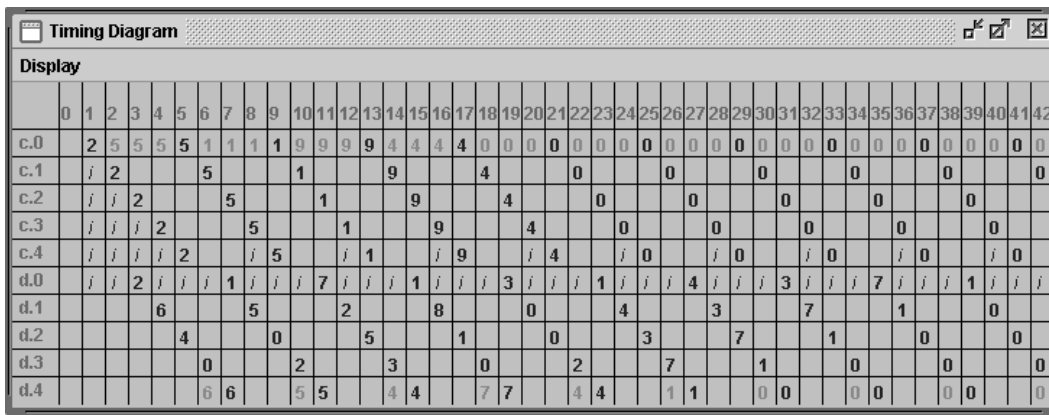


Figure 7. Timings for the modified version of the multiplication pipeline.

and it can be quite easily explained even to persons with little or no experience of parallel systems. This makes the tool valuable for use in education. The graphical layout of the visualisation is simple and can be magnified by the user. The ability to visualise an existing network directly from a textual CSP specification enable users to experiment by specifying systems and immediately viewing their behaviour, rather than being required to use an intermediate implementation language.

5 Related Work

A number of other tools exist within the parallel processing field related to the visualisation and animation of parallel networks. Many of these are based on purely visual notations such as Petri Nets ([16]) or on implementation languages ([11], [5]) rather than on formal notations. Many tools related to the CSP notation have been concerned with model checking, and most CSP animation tools (such as ProBE [13]) produce purely textual, rather than visual, output.

The Deadlock Checker Tool [12] can provide a simple visualisation of the communication structure of a CSP network; however, the visualisation does not include the names of processes or any information on the process's status or behaviour, and it cannot be animated, although some information on process behaviour can be obtained in textual form. Also, this tool is dependant on the commercial (and expensive) tool FDR [6] for compilation of CSP.

6 Conclusion

The original version of VisualNets was a useful tool for visual construction of simple networks and educational presentation. The newly enhanced version retains these ca-

pabilities while enabling the visual animation of existing specifications, more sophisticated analysis of networks, and a considerably stronger animation engine that deals with a far greater range of potential specifications and allows more sophisticated operators to be used in their construction. In addition, the capability for applying timing models will provide a greatly useful additional application.

It is intended to continue developing the tool within the architecture to produce a full industrial-strength network visualisation tool, capable of accepting any well-defined CSP network and visualising and animating it. Frameworks also exist within the tool for additional visualisation modules to be created, enabling custom visualisations to be built for presentation purposes or to better represent the structure of a given network or type of network; the tool structure also may make it possible to enable the system to monitor the parallel behaviour of networks specified in a language other than CSP by replacing the back-end animator.

References

- [1] A. E. Abdallah. Derivation of Parallel Algorithms from Functional Specifications to CSP Processes. In Bernhard Möller, editor, *Mathematics of Program Construction*, LNCS, pages 67–96. Springer Verlag, 1995.
- [2] A. E. Abdallah. Functional Process Modelling. In K. Hammond and G. Michealson, editor, *Research Directions in Parallel Functional Programming*, pages 339–360. Springer Verlag, 1999.
- [3] A. E. Abdallah. Synthesis of Massively Pipelined Algorithms for List Manipulation. In L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert, editor, *Parallel Processing, Euro-Par'96*, LNCS, pages 911–920. Springer Verlag, 1996.

- [4] A. E. Abdallah. A Graphical Tool for the Visualisation and Animation of Communicating Sequential Processes. In D. Pritchard and J. Reeve, editors, *Parallel Processing, Euro-Par'98*, volume 1470 of *LNCS*, pages 165–172. Springer-Verlag, 1998.
- [5] T. Delaitre, M. J. Zemerly, P. Vekariya, G. R. Ribeiro Justo, J. Bourgeois, F. Schinkmann, and S.C. Winter. Edpepps: An environment for the design and performance evaluation of portable parallel software. In *Proceedings of the Workshop on Portable Software Tools for Parallel Applications*, 1997.
- [6] Failures-divergences refinement. <http://www.formal.demon.co.uk/>.
- [7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, 1999.
- [8] M. Green and A. Abdallah. Interfacing Haskell with Java. In G. Michaelson and P. Trinder, editors, *Current Trends in Functional Programming*. Intellect, 2000.
- [9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [10] S. Peyton Jones and J. Hughes. *Report on the programming language Haskell, a Non-Strict, Purely Functional Language*. <http://www.haskell.org/definition/>.
- [11] P. Kacsuk, G. Dozsa, T. Fadgyas, and R. Lovas. The GRED Graphical Editor for the GRADE Parallel Program Development Environment. In *Future Generation Computer Systems, No. 15 (1999)*, pages 443–252, 1999.
- [12] J. M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, The University of Buckingham, 1996.
- [13] Probe. <http://www.formal.demon.co.uk/>.
- [14] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1998.
- [15] J. Schönwälder and H. Langendörfer. Tcl extensions for network management applications. In *Proceedings of the 3rd Tcl/Tk Workshop, Toronto (Canada)*, 1995.
- [16] Harold Störrle. An evaluation of high-end tools for petri-nets. Technical report. <http://www.pst.informatik.uni-muenchen.de/personen/stoerrle/survey.ps.gz>.