Cross Cutting Condensed Graphs *Submission to PDPTA05*

Barry P. Mulcahy, Simon N. Foley, and John P. Morrison
Department of Computer Science,
University College Cork,
Ireland.
{b.mulcahy, s.foley, j.morrison}@cs.ucc.ie

***Abstract*—Condensed Graphs provide a graph based based programming model that unifies availability-driven, coercion-driven and control-driven computing. In this paper we explore the issues of concurrency and synchronization between condensed graph applications. A paradigm for condensed graph synchronization is proposed and an architectural pattern is developed that forms the basis of a novel synchronization mechanism.**

KEYWORDS: Condensed Graphs; Concurrency; Synchronisation; Parallel Programming; Distributed Systems.

I. INTRODUCTION

Like classical dataflow [2], the Condensed Graphs ($\mathcal{CG}$) model [9], [10] is graph-based and uses the flow of entities on arcs to sequence and trigger execution. In contrast, Condensed Graphs are directed acyclic graphs in which every node contains not only operand ports, but also an operator and a destination port. Arcs incident on these respective ports carry other Condensed Graphs representing operands, operators and destinations. Condensed Graphs are so called because their nodes may be condensations, or abstractions, of other Condensed Graphs. Condensed Graphs can thus be represented by a single node (called a *condensed node*) in a graph at a higher level of abstraction.

A number of working prototypes that use Condensed Graphs have been developed, demonstrating its usefulness as a general model of computation. Prototypes include a distributed computing platform that schedules the execution of coarse-grain computations, described as Condensed Graphs, across heterogeneous database middlewares [8], Web-Services [6], hardware [7], clusters and Grid [5], .

Condensed Graphs are inherently parallel; a graph describes a collection of nodes whose (parallel) execution is constrained only by the sequencing of the defining graph. Node sequencing is defined explicitly in terms of node dependencies *within* a given graph. However, explicit sequencing constraints and/or synchronisation between the nodes of *different* graphs has not been considered. In this paper we explore a number of Condensed Graph primitives that can support concurrency and inter-graph synchronisation.

The rest of the paper is organised as follows; Section II provides an overview of Condensed Graphs, and Section III motivates the provision of synchronisation primitives. In Section IV we introduce primitives for cross cutting Condensed Graphs. Section V describes a resource negotiation pattern using the proposed synchronisation primitives. The patterns developed are then applied to the Dining Philosophers Problem in Section VI. Section VII describes how the primitives are implemented for a distributed execution platform.

Presenting author will be B. Mulcahy. Telephone +353214903977; fax +353214274390 Telephone

II. CONDENSED GRAPHS

The basis of the $\mathcal{CG}$ firing rule is the presence of a Condensed Graph in every port of a node. That is, a Condensed Graph representing an operand is associated with every operand port, an operator Condensed Graph with the operator port and a destination Condensed Graph with the destination port. This way, the three essential ingredients of an instruction are brought together (these ingredients are also present in the dataflow model; only there, the operator and destination are statically part of the graph).

Any Condensed Graph may represent an operator. It may be a condensed node, a node whose operator port is associated with a machine primitive (or a sequence of machine primitives) or it may be a multi-node Condensed Graph.

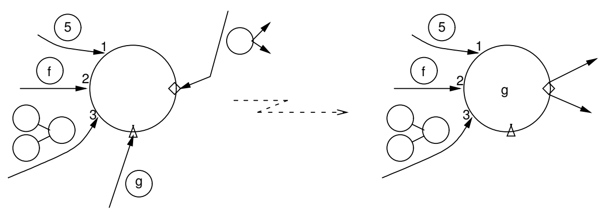

Fig. 1. Condensed Graphs congregating at a node to form an instruction

The present representation of a destination in the $\mathcal{CG}$ model is as a node whose own destination port is associated with one or more port identifications. Figure 1 illustrates the congregation of instruction elements at a node and the resultant rewriting that takes place. We decorate connections to distinguish between different kinds of ports, and use numbers to distinguish input ports.

Executing a Condensed Graph corresponds to scheduling its fireable nodes to run on ancillary processors, based on the constraints of the graph. The nodes in a graph are represented as triples (operation, operands, destination) and are constructed by the *Triple Manager* (TM) as the graph executes. Once a node is ready to fire, the triple manager can schedule it for execution on an ancillary processor. The $\mathcal{CG}$ operators can be divided into two categories: those that are ‘value-transforming’ and those that only move Condensed Graphs from one node to another in a well-defined manner. Value-transforming operators are intimately connected with the ancillary processors and can range from simple arithmetic operations to the invocation of software components that form part of an application system. In contrast, Condensed Graph moving instructions are few in number and are architecture independent. These *TM primitives* include the condensed node evaporation operator and the `ifel` node.

By statically constructing a Condensed Graph to contain operators and destinations, the flow of operand Condensed Graphs sequences the computation in a dataflow manner. Similarly, constructing a Condensed Graph to statically contain operands and operators, the flow of destination Condensed Graphs will drive the computation in a demand-driven manner. Finally, by constructing Condensed Graphs to statically contain operands and destinations, the flow of operators will result in a control-driven evaluation. This latter evaluation order, in conjunction with side-effects, is used to implement imperative semantics. The power of the $\mathcal{CG}$ model results from being able to exploit all of these evaluation strategies in the same computation, and dynamically move between them, using a single, uniform, formalism.

III. Concurrency and Synchronisation

Condensed Graphs are inherently parallel; a graph describes a collection of nodes and/or operations whose (parallel) execution is constrained only by the sequencing of the defining graph. Node sequencing is defined explicitly in terms of node dependencies *within* a given graph. The hierarchical nature of Condensed Graphs also result in limited implicit sequencing between the nodes of different graphs that are explicitly sequenced. However, explicit sequencing constraints between the nodes of *different* graphs has not been considered.

Example 1: Consider the Dining Philosophers Problem [3] where a group of philosophers sit around a table sharing one fork between each pair of philosophers. In this problem the forks are a shared resource and in order to eat a philosopher must have the two forks on either side of him/her. Figure 2 demonstrates how a fork might be modeled as a single independent graph, while Figure 3 models a philosopher as an independent graph. By definition [9], the behaviour of a Condensed

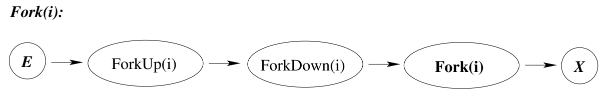

Fig. 2. A Fork Resource

Node such as $\mathsf{Fork}(i)$ is constructed as a Condensed Graph with a single entry node (*E*) and single exit node (*X*). The other nodes in the graph represent the operations that are available: $\mathsf{Up}(i)$ and $\mathsf{Down}(i)$. Arcs represent data paths between the operations which, in this case, may fire (execute) when data arrives at their input ports. For example, when $\mathsf{Up}(1)$ fires, it may, in turn, fire, $\mathsf{Down}(1)$ and so forth. Firing a condensed node such as $\mathsf{Fork}(1)$ *evaporates* it into the graph that defines it, with input available from the *E* node and final output emanating from its *X* node. △

The concurrent composition of multiple philosophers ($\mathsf{Philo}(0)$, $\mathsf{Philo}(1)$ ...), synchronising with their neighbouring forks ($\mathsf{Fork}(0)$, $\mathsf{Fork}(1)$, ...), provides an elegant approach to describing the dining philosophers problem [3]. Viewing the problem as a collection of interacting graphs and/or processes results in a problem specification that is more tractable than attempting to characterise it as a single uniform process/graph [4], [11], [13]. However, this approach requires the construction of explicit sequencing constraints between the nodes of different graphs. Simple node side-effects provide an ad-hoc strategy for achieving such sequencing constraints between the nodes of different Condensed Graphs. In the example above, this synchronisation might be achieved by graph nodes communicating their intentions indirectly with each other, via shared state for example. The disadvantage of this ad-hoc approach is that the inter-graph communication is not explicit in the graph and may result in synchronisation problems. Such problems can be alleviated by having a centralised synchronisation point, such as a semaphore or rendevouz. However, these mechanisms would rely on side effects outside of the graph and could result in unexpected dependencies not explicitly specified at the graph level. A similar problem has been encountered in Workflow Management Systems (WFMSs), where a proposed solution [1] is to associate a signed semaphore with the resource being accessed, however, this approach has the effect of pushing state on the requested resource. In this paper we explore a number of strategies for supporting decentralised synchronisation explicitly within a graph.

IV. Cross Cutting Condensed Graphs

In the previous section we illustrated the need for a concurrency mechanism between graph instances. In this section we introduce a pair of simple synchronisation primitives for Condensed Graphs; the *off-page* and *on-page* connectors. Using these primitives we establish *cross cutting* features for Condensed Graphs. Cross cutting uses pairs of connectors to establish a virtual arc between graphs. This extension of the Condensed Graph model allows a value to be exported from a graph by an off-page connector without exiting through the X node. Matching pairs of connectors coalesce creating a virtual arc between the host graphs, allowing a value to flow on that arc from one graph to another. The act of connector coalescence can be interpreted as a dynamic rewrite of the address of the target destination port by the *Triple Manager* (TM). Once an operand is available to an off-page connector, its corresponding on-page connector can fire, importing the operand into a different graph. The use of connectors to transfer an operand between the nodes of two graphs can be seen in Figure 4.

In order to maintain the firing rules of the Condensed Graph model the connectors have a semantics similar to primitive nodes. The off-page connector has a single operand port and is non-blocking, that is, the node is fireable once a value is available to it. When an off-page connector fires the value is passed as an operand and becomes available to the corresponding on-page connector (in a different graph). If the graph containing the matching on-page connector has not yet been uncovered, the

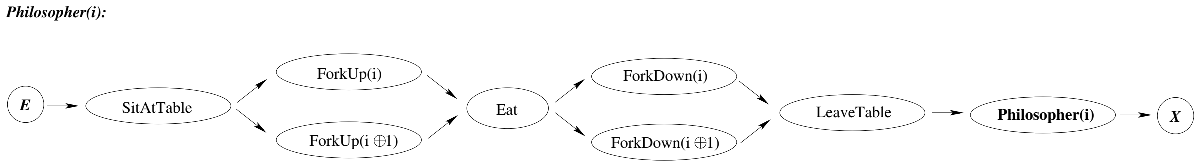

Fig. 3. A Dining Philosopher

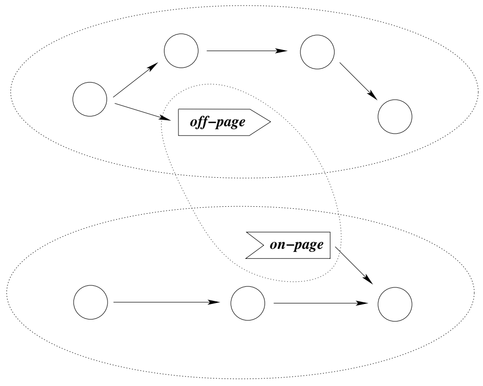

Fig. 4. Cross-Cutting Between Two Graphs

TM is responsible for ensuring the value being passed is stored as an operand for the importing graph, ready for use when the graph is executed. Exporting the operand once it is available allows the source graph to evaporate as normal and the memory used to define the graph can be deallocated.

The on-page connector receives its operand from the off-page connector, but it cannot coerce the off-page to fire. This is in contrast to the Condensed Graph model and as a result the on-page connector blocks until a value is available. This is necessary as the source graph hosting the off-page connector should not be affected by the execution of the graph hosting the corresponding on-page connector.

EXAMPLE 2: Consider two autonomous graphs **A** and **B** as seen in Figure 5. When the node a in graph **A** fires it produces

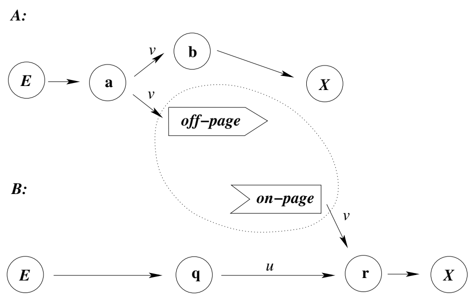

Fig. 5. Synchronisation Between Two Graphs

the value v. This value is passed to the node b and the off-page connector. In graph **B** the node r takes its operands from the node q and the on-page connector. The node r will not fire until the operand v is imported into the graph by the on-page connector. △

Next we consider how to arbitrate synchronisation between a number of graphs. An obvious approach is to expand the semantics of the proposed connectors to allow a many-to-one relationship as depicted in Figure 6. This would allow multiple graphs to attempt to synchronise using the single on-page connector in the target graph **G3**. The on-page receives and queues values from all the off-page nodes, when all values are present it passes on a value from the queue to its host graph. With a many-to-one relationship it is necessary to ensure that all off-page values have been received by the on-page connector for the proper garbage collection of the source graphs. A many-

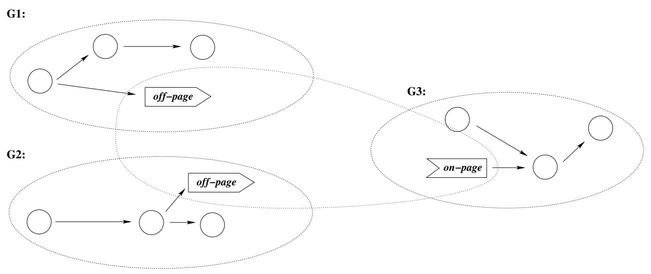

Fig. 6. Many-To-One Connectors

to-one relationship between connectors has the advantage of synchronising all the consumer graphs (containing the off-page connectors) by using a single coordination point. Another benefit is that it simplifies the graphs, however, it pushes responsibility for the managing of connectors onto the resource being requested. Also the mechanism to coordinate the requests is centralised in a single stateful on-page connector, contrary to the inherently decentralised and stateless nature of Condensed Graphs.

A mapping of many-to-many connectors has the same problems associated with a many-to-one mapping, that of centralisation and burdening the resource with the coordination management.

In order to prevent state associated with coordinating synchronisation being pushed onto the producer graph consider a one-to-one mapping between connectors. Once an off-page connector has its operand, the value can be imported by the matching on-page as a request for access to the resource. If a resource has multiple consumers, the graph encoding the resource has an on-page connector for each potential consumer graph. When a resource is requested a non-deterministic choice of the incoming requests is made by the producer graph. The consumers are subsequently notified by the producer graph as to who currently has gained the resource. This notification is accomplished by corresponding sets of connectors from the producer graph to the consumers.

If the on-page connector is stemmed to the next node in the graph it is not required to fire instantly due to the demand-driven dependency, in this case the imported value is stored as an operand ready for use. The operand port of the on-page connector can thus be viewed as a bounded queue of length one. By using pairs of connectors complex concurrency patterns for graphs can be developed. Sets of one-to-one connectors can also simulate a one-to-many relationship by passing a single value to several off-page connectors and having the corresponding on-page connectors in different graphs.

A one-to-one relationship between connectors has the effect of establishing a one-off, asynchronous message transfer between graphs. Since the on-page is stateless and the resource is modeled as a Condensed Graph, the producer graph provides a decentralised synchronisation point. Even though the one-to-one mapping may result in additional graph complexity and connector overhead, the decentralised management of a resource has significant practical implications and is conducive to the Condensed Graph model of computation.

V. Resource Negotiation Pattern

Consider the problem of a producer with one consumer requesting access to a resource. With the proposed cross cutting approach using one-to-one connectors the problem can be modeled as two separate graphs. Figure 7 describes the graph for the producer. The first stage of the producer graph is a wait for us-

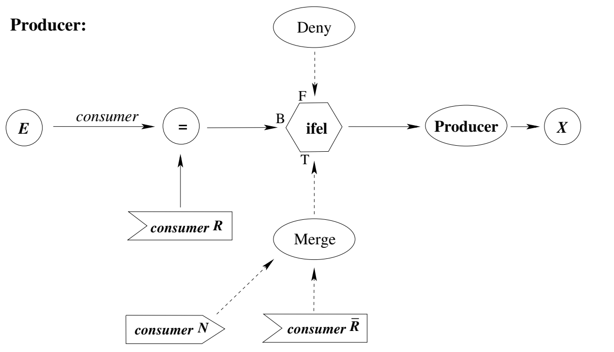

Fig. 7. Producer Graph using One-To-One Connectors

age request by the consumer, the on-page connector *consumer*^{R} will block the graph until the corresponding off-page fires in the consumer graph seen in Figure 8. The event of the off-page

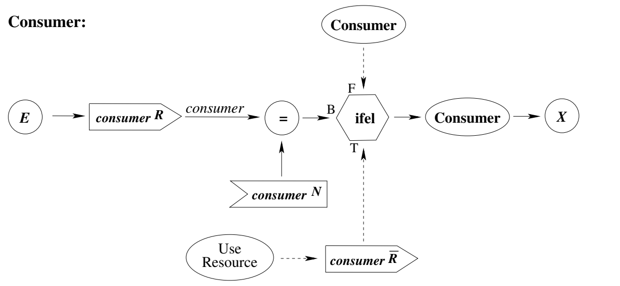

Fig. 8. Consumer Graph using One-To-One Connectors

consumer^R firing is interpreted as a request for access to the resource by the producer graph. Since the producer has only one consumer the incoming request will always originate from the same off-page *consumer* and results in the consumer graph being notified that it has gained access to the resource. In this case the `Deny` node of the producer will never fire (but is included here for the sake of completion). The consumer notification is accomplished by firing the off-page connector *consumer*^{N} in the producer graph. The matching on-page in the consumer graph will block until the notification is received. Should the consumer be notified that the resource has been granted to some other graph, the boolean value for the consumers `ifel` will resolve to False and the consumer graph will recurse attempting another request for the resource. However, in this example (with only one consumer) all requests will be met favourably by the producer and the boolean will be true. This allows the consumer to use the resource and when finished the off-page connector *consumer*$^{\bar{R}}$ will fire, releasing the resource back to the control of the producer. In the producer graph the on-page connector *consumer*$^{\bar{R}}$ will block preventing recursion until the release action is received from the consumer. When the release is given by the consumer graph, the resource is made available again to usage requests by the producer graph.

VI. The Dining Philosophers

Examination of the producer & consumer problem in the previous section reveals a pattern for resource negotiation between graphs, namely that of request, notification and release. Request *(R)* of resource usage by a consumer, notification *(N)* of which graph currently is allowed access by the producer, and release *($\bar{R}$)* once the resource has been used by the consumer. In this section we apply this pattern to solve the resource management issues in The Dining Philosophers Problem [3].

In this problem, forks are considered a resource, each fork is shared between two philosophers (consumers). The graph **Fork(i)** in Figure 9 is modeled as a resource producer as seen in Section V. However, in this case the resource has two poten-

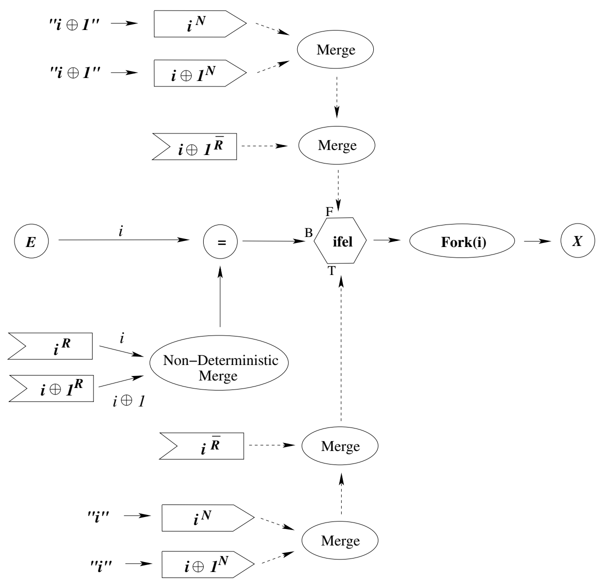

Fig. 9. Dining Philosophers: **Fork(i)**

tial consumers, as such, the producer must 'listen' for both usage requests. The on-page connectors i^R and $i \oplus 1^R$ in the fork graph block until a request is made by a philosopher. Should the philosophers on either side of the fork make simultaneous requests a non-deterministic choice will be made between the two. When a usage request is received by either on-page connector the graph will unblock and a comparison will be made to determine which graph has made the request, philosopher(i) or philosopher(i⊕1). The boolean result of the comparison defines the execution path of the `ifel` node. If it resolves to true, as is the case for a usage request from philosopher(i), the `ifel` will coerce the firing of the lower part of the fork graph. In this case, the off-page connectors i^N and $i\oplus 1^N$ fire, notifying the philosophers on either side of the fork that philosopher *'i'* currently has access to the fork.

The on-page connector $i^{\bar{R}}$ then blocks the fork graph until it receives the release message from philosopher(i). Had the request comparison resolved to false, that is, the request originated from philosopher(i⊕1), then a similar pattern would be followed. However, the subsequent notification would grant the resource to philosopher(i⊕1) and the release on-page $i\oplus 1^{\bar{R}}$ would block until that philosopher was finished with the fork.

When the release is received by the relevant on-page connector the **Fork(i)** graph recurses, making the resource available again to either philosopher.

For clarity the modeling of a philosophers tasks have been broken down into several graphs. Figure 10 shows the different high-level stages of a philosophers actions. After sitting at the

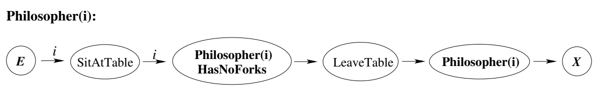

Fig. 10. Dining Philosophers: **Philosopher(i)**

table a philosopher has no forks. Figure 11 demonstrates the extended negotiation pattern (derived from Section V) that a

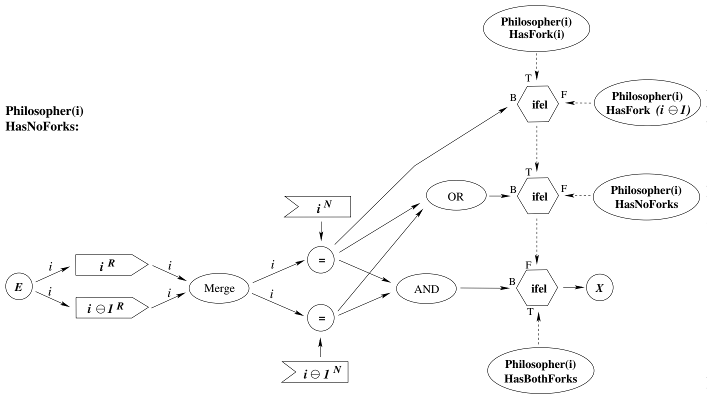
Philosopher(i) HasNoForks:
Philosopher(i) HasFork(i)
T
B
ifel
F
Philosopher(i) HasFork (i ⊖ 1)
T
OR
B
ifel
F
Philosopher(i) HasNoForks
i N
E
i
i R
i
i
i ⊖ 1 R
i
Merge
i
i
=
=
AND
F
B
ifel
X
T
i ⊖ 1 N
Philosopher(i) HasBothForks

Fig. 11. Dining Philosophers: **Philosopher(i) with no forks**

philosopher must engage in to receive both forks. The initial step is a request (made in parallel) to the two forks on either side using the off-page connectors i^R and $i\ominus 1^R$.

The graph **Philosopher(i)HasNoForks** will block until it receives notification from both forks as to their current status. If the notification from the on-page connectors i^N AND $i\ominus 1^N$ is favourable, the philosopher has access to both forks and the graph seen in Figure 13 is fired. If neither fork is available the graph **Philosopher(i)HasNoForks** recurses and attempts to access both forks again.

Should the philosopher gain access to only one fork *i⊖1*, the graph **Philosopher(i)HasFork(i⊖1)** is fired. Figure 12 depicts a similar scenario where the philosopher has one fork *i*, and must again request access to fork *i⊖1.* this graph is similar to the consumer graph seen in Section V, the request and notification are repeated for fork *i⊖1.* If a failure notification is repeated this graph will recurse, and keep doing so attempting

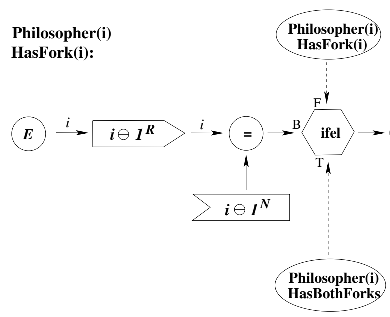
Philosopher(i) HasFork(i):
Philosopher(i) HasFork(i)
F
E
i
i ⊖ 1 R
i
=
B
ifel
X
T
i ⊖ 1 N
Philosopher(i) HasBothForks

Fig. 12. Dining Philosophers: **Philosopher(i) with one fork**

to gain access to the resource. Should the notification prove positive for the philosopher in question then he/she has both forks and the graph in Figure 13 is coerced to fire. Throughout

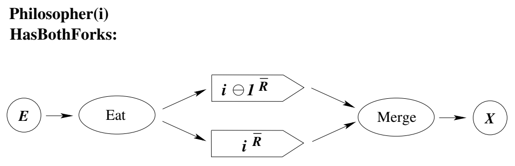

Fig. 13. Dining Philosophers: **Philosopher(i) with both forks**

the period between notification (N) and release ($\bar{R}$), the fork resource is held by the successful philosopher. This includes the time where a philosopher has only one fork and is attempting access to the other fork.

Should one philosopher repeatedly gain access to a fork, the philosopher who loses out may fall behind in his requests for the resource. In this case the failure notifications for the lagging philosopher will be placed as operands to the on-page connectors i^N yet to be uncovered by recursion. A mechanism to quickly cycle through these failure notifications could be incorporated into the Triple Manager to automatically garbage collect the failing graphs, this can be viewed as a flush of the already denied requests. While this approach may seem to cause extra overhead, a decentralised synchronisation mechanism is consistent with the Condensed Graph model. However, the techniques described here are equally applicable to a centralised approach. Should the resources be available to support a centralised approach the implementation of the connectors can be switched to support this, and push the synchronisation state onto the resource.

In Figure 13 we see the philosopher has both forks and can now engage in the action of eating. When finished eating both forks are released by the philosopher using the off-page connectors $i\ominus 1^{\bar{R}}$ and $i^{\bar{R}}$. The philosopher can then leave the table as seen earlier in Figure 10 and repeat the process of sitting eating and leaving the table.

Figure 14 provides the basis for an entire table of `N` philosophers and forks. The two `ForAll` nodes create 0 to N-1 number of philosophers and forks. Each of the instantiated graphs takes a single parameter i=0..N-1, which identifies each graph by an integer value, representing their relative positions around the table. This integer parameter *(i)* ensures that each philosopher graph negotiates with the correct fork graphs using the off-page and on-page connectors.

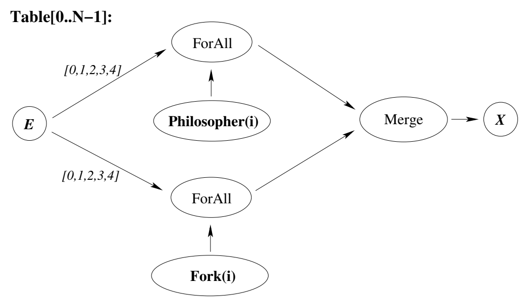

Fig. 14. Dining Philosophers: **Table[0..N-1]**

VII. Implementation

As stated in Section III and Section IV, fragmenting a problem into representative graphs results in a specification that is more tractable than attempting to characterise it as a single uniform process/graph like CCS [4], CSP [13] or Petri Nets [11].

Simple node side-effects using existing mechanisms, such as a semaphore or rendevouz, would provide an ad-hoc strategy for achieving the necessary sequencing constraints between the nodes of different Condensed Graphs. However, these stateful mechanisms would not be consistent with the stateless nature of Condensed Graphs. Also, communication using side-effects that are not explicitly in the graph can result in inconsistencies does to hidden dependencies between nodes in the graphs. In contrast, the coalescing of connectors to cross cut graphs is appropriate for the Condensed Graph model.

Cross cutting Condensed Graphs requires support for the off-page and on-page connectors in the Triple Manager. WebCom [8] is a metacomputer which incorporates a TM for executing Condensed Graphs in a distributed heterogeneous environment. In our implementation the connectors are included in the WebCom node library as TM *primitives*. The off-page primitive has one operand port and one destination port. Once a value is available on its operand port WebCom automatically rewrites the primitives destination port to reflect the location of the corresponding on-page primitive. The on-page primitive also has one operand port and one destination port. If the matching on-page is unavailable due to its host graph not being uncovered, WebCom holds the value as the operand to the on-page primitive. Once the value is scheduled as the operand to the on-page, the primitive can evaporate and be replaced by the operand. Execution of the importing graph then proceeds as normal.

VIII. Discussion and Conclusions

The contribution of this paper is the development of novel Condensed Graph based synchronisation primitives. These primitives are used to specify a pattern for resource negotiation. This pattern provides a general ‘template’ for processes to negotiate and share resources.

Recall from Section II that a Condensed Graph is executed by a Triple Manager, which in turn, can be implemented in a variety of ways, ranging from a simple sequential interpreter to distributed peer-to-peer mechanisms. Thus, the execution of nodes in a condensed graph application can be coordinated in a centralized manner, or distributed across a network according to resource requirements. Therefore, the resource negotiation pattern can be used to implement a decentralized synchronization mechanism. In the case of the Dining Philosophers example, much of the the synchronization overhead (coordination of graph nodes) can be allocated to the Philosophers, reducing the load on forks. Future research will investigate the development of distributed protection mechanisms that use these patterns.

The existing implementation of the Condensed Graph Triple Manager has been extended to support the sychnronization primitives proposed in this paper. A challenge in using these primitives is to be able to properly reference the on-page connector in one graph from the off-page connector of another graph. This is done using the naming system proposed in [12].

References

[1] Gustavo Alonso, Divyakant Agrawal, and Amr El Abbadi. Process synchronization in workflow management systems. In *8th IEEE Symposium on Parallel and Distributed Processing*, October 1996.

[2] Arvind and Kim P. Gostelow. A computer capable of exchanging processors for time. Information Processing 77 Proceedings of IFIP Congress 77 Pages 849-853, Toronto, Canada, August 1977.

[3] Edsger Wybe Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[4] R. Milner. *Communications and Concurrency*. Series in Computer Science. Prentice-Hall International, Hemel Hempstead, 1989.

[5] John P. Morrison, Brian Clayton, David A. Power, and Adarsh Patil. Webcom-g: Grid enabled metacomputing. *The Journal of Neural, Parallel and Scientific Computation. Special Issue on Grid Computing. Editors H.R. Arabnia, G.A. Gravvanis, M.P. Bekakos*, 12(3):419–438, September 2004.

[6] John P. Morrison et al. Architectural neutral glue for COM objects. Internal Note, Center for Unified Computing, University College, Cork, Ireland, 2000.

[7] John P. Morrison, Philip D. Healy, and Padraig J. O'Dowd. Architecture and implementation of a distributed reconfigurable metacomputer. In *Proceedings of the Second International Symposium on Parallel and Distributed Computing (ISPDC 2003)*, pages 153–158, Ljubljana, Slovenia, October 2003.

[8] John P. Morrison, David A. Power, and James J. Kennedy. WebCom: A Web Based Distributed Computation Platform. Proceedings of Distributed computing on the Web, Rostock, Germany, June 21 - 23, 1999.

[9] J.P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Eindhoven, 1996.

[10] J.P. Morrison and M. Rem. Managing and exploiting speculative computations in a flow driven, graph reduction machine. proceedings of PDPTA'99: Las Vegas, USA. June 28-July 1, 1999.

[11] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[12] T.B. Quillinan and S.N Foley. Security in webcom: Addressing naming issues for a web service architecture. In *ACM Workshop on Secure Web Services (ACM-SWS2004).*, 2004.

[13] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.