Scalable Computing: Practice and Experience
Volume 9, Number 2, pp. 83–94. http://www.scpe.org
ISSN 1895-1767
© 2008 SCPE

OBJECT ORIENTED CONDENSED GRAPHS*

SUNIL JOHN AND JOHN P. MORRISON†

Abstract. Even though Object Orientation has been proven to be an effective programming paradigm for software development, it has not been shown to be an ideal solution for the development of large scale parallel and distributed systems. There are a number of reasons for this: the parallelism and synchronisation in these systems has to be explicitly managed by the programmer; few Object Oriented languages have implicit support for Garbage Collection in parallel applications; and the state of a systems of concurrent objects is difficult to determine. In contrast, the Condensed Graph model provides a way of explicitly expressing parallelism but with implicit synchronisation; its implementation in the WebCom system provides for automatic garbage collection and the dynamic state of the application is embodied in the topology of the Condensed Graph. These characteristics free programmers from the difficult and error prone process of explicitly managing parallelism and thus allows them to concentrate on expressing a solution to the problem rather than on its low level implementation. **Object Oriented Condensed Graphs** is a computational paradigm which combines Condensed Graphs with object orientation and this unified model leverages the advantages of both paradigms. This paper illustrates the basic features of Object Oriented Condensed Graphs as well as its support for large scale software development.

Key words: condensed graphs, object oriented systems, software engineering, distributed and parallel computing

1. Introduction. The support of large scale software systems has long been the focus of research in software engineering. Object Oriented Systems have attracted wide attention due to its many desirable properties which aid software development such as enhanced maintainability and reusability. With the support of characteristics such as inheritance, modularity, polymorphism and encapsulation, this paradigm can help the development of complex software programs [17]. However, the development of parallel and distributed applications is not significantly simplified by Object Orientation. The onus is still on the programmer to explicitly manage each parallel component and to ensure proper synchronisation. The interaction of parallel components in a large system can be complex and this complexity is compounded in sophisticated environments such as heterogeneous clusters and computational grids. Moreover, the encapsulation concept in OO is complex in a parallel environment. OO does not impose constraints upon invocation of an object's attributes or member functions. This complicates the relationship among objects when there are several method invocations. Similarly, memory management is a major concern. Most of the current Garbage Collection methodologies work sequentially; only a few OO languages can support automatic garbage collection in a parallel system. In addition, parallelism poses additional challenges for access control mechanisms and object state determination.

The Condensed Graph (CG) model of computing is based on Directed Acyclic Graphs (DAGs). This model is language independent and it unifies the imperative, lazy and eager computational models [2]. Due to its features including Implicit Synchronisation and Implicit Garbage Collection this computational model can be effectively deployed in a parallel environment. CGs have been employed in a spectrum of application domains, from FPGAs to the Grid [7, 4]. The most advanced reference implementation of the model is the WebCom abstract machine [5]. WebCom is being used as the basis of Grid-Ireland's middleware development and deployment [3].

This paper addresses the concept of **Object Oriented Condensed Graphs** (**OOCG**) as well as its development support in a large scale environment. Object Oriented Condensed Graphs is a unified model that combines the Object Oriented paradigm with the Condensed Graph methodology. This unified model helps to leverage the advantages of both paradigms. By integrating Condensed Graphs with Object Oriented principles, negative aspects of Object Orientation, when deployed in a large scale parallel environment, can be successfully addressed.

Some of the similar research in this area of modelling languages are Object Oriented Petri Nets (OOPN) [8] and Object Petri Nets (OPN) [9, 10, 11] in which the Object Orientation principles has combined with that of Petri Nets [13]. Other notable Petri Net modelling languages embodying OO concepts are CPN [12], HOOPN [14] and CO-OPN [15, 16].

This paper is organised as follows: Section 2 provides an overview of Condensed Graphs, and Section 3 presents the basics of Object Oriented Condensed Graphs. Section 4 presents Development support, particularly pattern based OOCG Development, as well as its performance analysis.

*The support of Science Foundation Ireland is gratefully acknowledged.
†Centre for Unified Computing, Dept. of Computer Science, National University of Ireland, University College Cork, Cork, Ireland (s.john,j.morrison@cs.ucc.ie). http://www.cuc.ucc.ie

2. Condensed Graphs. Like classical dataflow, the CG model is graph-based and uses the flow of entities on arcs to trigger execution. In contrast, CGs 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 CGs representing operands, operators and destinations. Condensed Graphs are so called because their nodes may be *condensations*, or abstractions, of other CGs. (Condensation is a concept used by graph theoreticians for exposing meta-level information from a graph by partitioning its vertex set, defining each subset of the partition to be a node in the condensation, and by connecting those nodes according to a well-defined rule.) Condensed Graphs can thus be represented by a single node (called a condensed node) in a graph at a higher level of abstraction. The number of possible abstraction levels derivable from a specific graph depends on the number of nodes in that graph and the partitions chosen for each condensation. Each graph in this sequence of condensations represents the same information but in a different level of abstraction. It is possible to navigate between these abstraction levels, moving from the specific to the abstract through condensation, and from the abstract to the specific through a complementary process called evaporation.

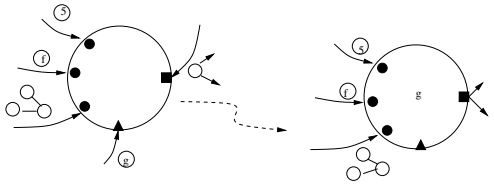

FIG. 2.1. *CGs congregating at a node to form an instruction.*

The basis of the CG firing rule is the presence of a CG in every port of a node. That is, a CG representing an operand is associated with every operand port, an operator CG with the operator port and a destination CG 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 CG 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 CG.

The present representation of a destination in the CG model is as a node whose own destination port is associated with one or more port identifications. Figure 2.1 illustrates the congregation of instruction elements at a node and the resultant rewriting that takes place.

Strict operands are consumed in an instruction execution but non-strict operands may be either consumed or propagated. The CG operators can be divided into two categories: those that are value-transforming and those that only move CGs from one node to another in a well-defined manner. Value-transforming operators are intimately connected with the underlying machine and can range from simple arithmetic operations to the invocation of software functions or components that form part of an application. In contrast, CG moving instructions are few in number and are architecture independent.

By statically constructing a CG to contain operators and destinations, the flow of operand CGs sequences the computation in a dataflow manner. Similarly, constructing a CG to statically contain operands and operators, the flow of destination CGs will drive the computation in a demand-driven manner. Finally, by constructing CGs 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 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.

3. Object Oriented Condensed Graphs. The Object Oriented Condensed Graphs paradigm, OOCG, has been developed to address the limitations of Object Oriented Systems mentioned in Section 1. As a unified model, Object Oriented Condensed Graphs preserves all the current functionalities of Condensed Graphs while featuring many major Object Oriented Concepts. The core features and functionalities are given below [1]:

3.1. Object Annotations. The *definition* of a Condensed Graph may be viewed as a class from which instances can be created. Such instances are analogous to objects. A class can contain nested graphs representing individual methods and attributes.

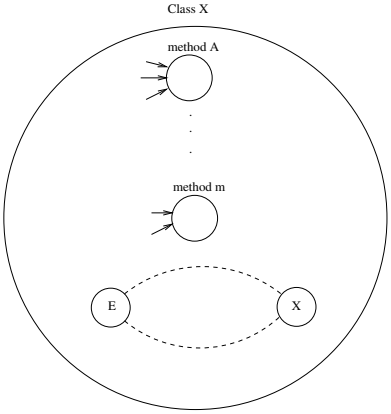

FIG. 3.1. *A Node to represent Class X. The member functions form child nodes within that node.*

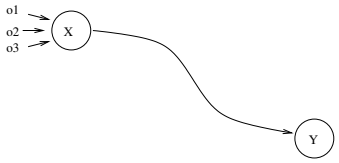

FIG. 3.2. *Invoking Object X with Operands. Sending the result to Y.*

In the CG model, Graph instance creation can occur implicitly when an appropriate CG node is invoked. Alternatively, explicit instance creation will give rise, not only to an appropriate graph instance but also, to a CG node which can be used to represent that instance. In effect, these nodes are coherent names of dynamically created objects.

OOCG specifies two kinds of visibility for its member functions and attributes. As is typical in an Object Oriented language, *private* members belonging to an Object can be accessed only within that Object. In contrast, a *public* member has a broad visibility. It can be accessed from inside as well as from outside of that Object. In other words, it has a *Global Scope*. Some of these scenarios are shown in Figs. 3.1, 3.2 and 3.3.

3.2. Inheritance in Condensed Graphs. Object Oriented Condensed Graphs incorporate *single inheritance*. A Class can inherit from other classes. In this way, a *sub class* can inherit the properties of its *super class*. The sub class can introduce new properties or can override the inherited properties of its super class. As shown in Figs. 3.4 and 3.5, subclass **Class B** extends superclass **Class A**. By default, public members of A are available in Class B and these can be used for operations within B.

The class definitions can be instantiated as Objects. The characteristics of *Polymorphism* also can be observed in the class instances. By overriding the parent class operation in the child class, the behaviour can be changed in the sub class. In the above example, we can redefine the member function *func1* in the sub class so that the whole operation behaviour can be redefined. This also gives rises *dynamic binding*. During compile time the contexts between the operations can be switched.

3.3. Concurrency and Synchronisation. OOCG adheres to the standard OO principle of Encapsulation. The public methods and attributes are visible and accessible from outside the Object. A method within an object can be subject to several invocations. These invocations can be intra Object or can be inter Object.

Condensed Graphs are inherently parallel by design. The definition of the graph dictates the parallel execution. The node dependencies within the graph explicitly specifies the order of execution. *Cross Cutting Condensed Graphs* [6] is a methodology to cater for inter graph synchronisation for concurrently executing graphs. This methodology helps to overcome the model's basic criteria that values exit a graph only through its X Node. In the cross-cutting version, values may exit via a special construct thus helping to coordinate independent graphs.

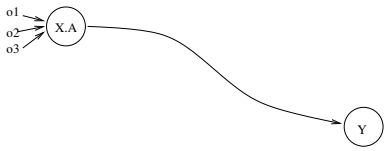

FIG. 3.3. *Invoking Method A of Object X with Operands. Sending the result to Y.*

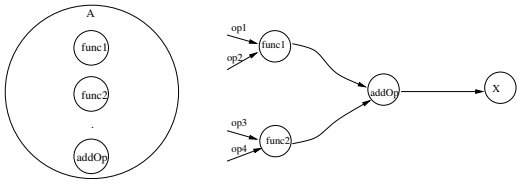

FIG. 3.4. *The parent super class A.*

A new graph model to support access to shared resources is proposed in Figure 3.6. Many operands may converge to the E node of this graph. A priority based queuing mechanism is employed in the E node to handle these accesses and a local semaphore is used to enforce sequential access to the underlying graph. Typically it sets a flag when it allows the invocation of the graph. When the X node of the graph fires this flag reverts back to its previous value.

3.4. Explicit Naming Scheme. OOCG Node names adapt from the code semantics. A Class Node bear the name of the Class in the program. A Class method name is normally preceded by a colon. As an example, the Nodes **C** and **C:getDetails** illustrate Class **C** and the method **getDetails** within class **C** respectively. Inheritance and the nested class scenarios also can be incorporated. The Node annotation **A.B.C:getDetails** illustrates the member function **getDetails** of the class **C** which is derived from base classes **B** and **A**. Some research regarding the naming scheme using the Condensed Graph model is addressed in [5].

3.5. Object Creation and Automatic Garbage Collection. Condensed Graphs support Object Creation and *Implicit Object Destruction* or Automatic Garbage Collection [1]. Object Creation can be performed by invoking the **Create** instruction. This instruction takes the Class Definition as an input Operand and creates an Object as output. Figs. 3.7 and 3.8 illustrates some scenarios of the Object creation. Typically when an Object's X Node fires the Object is collected automatically.

4. OOCG Development Support. OOCG development framework has been developed for the creation of OOCG applications for concurrent and distributed environments. **OOCGLib** is a programming package consisting of a set of Java classes and libraries to address integrated development. OOCGLib is integrated with $WebCom\text{-}G_{IDE}$ [3] which allows developers to develop OOCG applications in an interactive, distributed environment (Figure 4.1). This development environment is implemented as an application layer on top of the WebCom [5] middleware. As an integrated framework, it benefits from the features of WebCom such as load balancing, fault tolerance and security policies.

4.1. Multiphase Patterns. In modelling a system, patterns often used to efficiently express solution in a standard manner. Patterns are template solutions that have been developed to address common recurring problems in particular application domains.

Gamma et al. [18] suggested recurring solutions to common problems in design. They presented their ideas in the context of Object Oriented design. Le Guennec et al. [21] has incorporated some of this work on design patterns in UML using their customised tool UMLAUT. W.M.P. van der Aalst et al. suggested *Workflow Patterns* [22] as Requirements for workflow languages. Later research in this area has also addressed control flow patterns [23] and provided a formal description of each of them in the form of a Coloured Petri-Net (CPN) model.

Patterns currently available in software engineering only apply to particular engineering phases and do not extent into the multiple phases. A pattern paradigm which addresses problems in all phases of a software

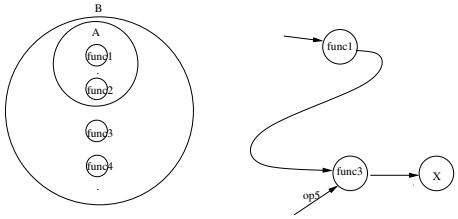

FIG. 3.5. *The child sub class B. The child class inherited the public contents of class A.*

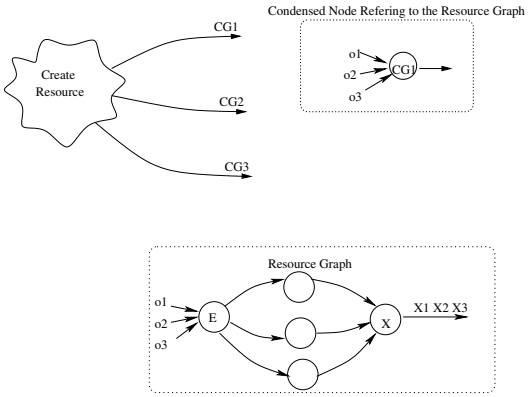

FIG. 3.6. *The Condensed Nodes refer to the Resource Graph. The E node of the Resource Graph contains a local semaphore which restricts the access to resource one at a time.*

lifecycle, from high level modelling to design, implementation and testing is currently missing from the software engineering landscape. The concept of a **Multiphase pattern** is presented here to address solutions to all phases of the software engineering lifecycle ranging from modelling, design to implementation.

Two Multiphase patterns, as part of the OOCG model, are presented in this paper to address common scenarios. These are design patterns which are being adapted and extended to cover multiple phases in the engineering lifecycle. Similar to the strategy adopted by Gamma et.al. [18], the OOCG Multiphase Patterns also name, motivate, and explain solutions for generic problems.

4.1.1. Predecessor-Successor Pattern. This pattern addresses the sequential execution of dependant tasks.

Name of the Pattern. Predecessor-Successor.

Related Patterns. Sequential Routing, Serial Routing [19]. In this Petri net based workflow pattern, a process wait for the other process for serial execution.

Motivation. In a concurrent system of dependent tasks, the order of execution of tasks is a matter of interest. As an example, if there are two dependent tasks and if any of the tasks requires a value that needs to be transferred from the other task, it is important to know their order of execution.

Description of the Problem. Consider two tasks G_1 and G_2. G_2 expects a value X_1 that needs to be transferred from task G_1. In this case, task G_1 has to be executed first to produce the value X_1.

Solution of the Problem. Induce the notion of *Sequencing* in the task modelling. If a task needs a value from another task, execute the first task prior to the second task. The value produced as a result of the execution of the first task is transferred to the second task.

Implementation. Arrange the order of execution of the tasks. The task which executes first will be denoted as *Predecessor* and the task which waits for the value from the *Predecessor* task will be termed as *Successor*. Once the value has been produced by the Predecessor task, the value will be consumed by the Successor. The value transfer will be based on the assumption that the Successor task waits for the value (Figure 4.2).

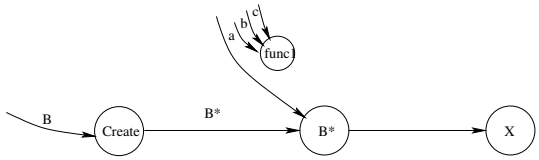

FIG. 3.7. *In this Condensed Graph realisation, object B* has been created and is used to invoke the function* func1 *with operands a,b and c.*

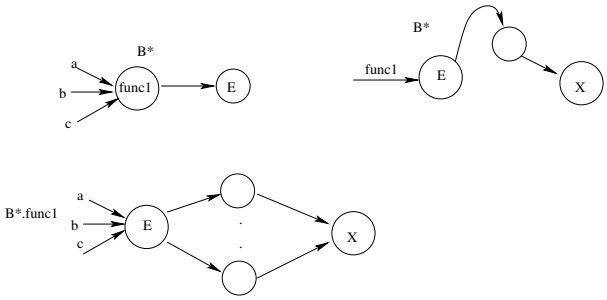

FIG. 3.8. *Different CG scenarios for the Object creation.*

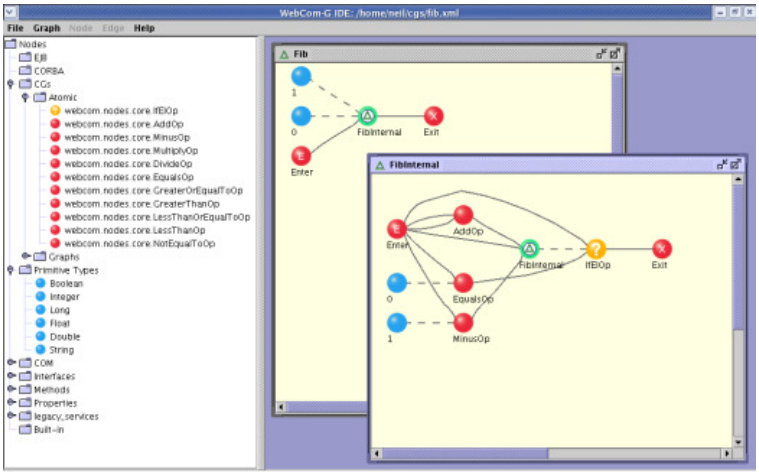

FIG. 4.1. *Sample Screen shot of WebCom-G$_{IDE}$, showing the Graphs used to generate a Fibonacci sequence.*

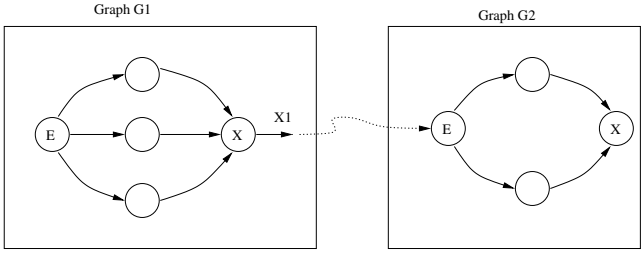

FIG. 4.2. *Predecessor-Successor pattern. Graph* G_2 *needs to wait for graph* G_1 *hence the graph* G_2 *is Successor.*

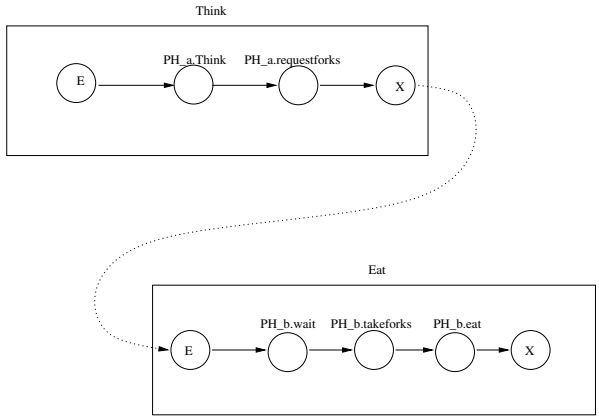

FIG. 4.3. *Graph* Eat *needs to wait for graph* Think.

FIG. 4.4. *(a) Connected dataflow graphs and (b) Non-connected Lazy graphs. In scenario (a), G_2 needs to wait for G_1. This scenario is applicable if the Operand port in G_2 is strict. In scenario (b), G_2 can start before G_1 executes. The Operand port in G_2 is non-strict in this scenario. In the first case, G_1 is the Predecessor graph and G_2 is the Successor graph. In the second case (b), G_1 becomes Successor and G_2 becomes Predecessor.*

Example. Consider the classical dining philosophers problem [20]. In this example, 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. In Figure 4.3, graph *Eat* has to wait for the value *fork* from the graph *Think*. As is evident from this scenario, *Think* acts as an *Predecessor* graph while *Eat* as the *Successor* graph.

Predecessor-Successor in OOCG. The sequencing of the execution of nodes in an OOCG may be *lazy* or *eager*. The actual order of evaluation is determined by the strictness of Node Operand Ports and the form of the Operand, i. e., whether the Operand is a normal order form or is in need of further reduction. An Eager evaluation sequence results when the operand ports of a node are *non-strict* regardless of the form of the operands. When the operand ports are *strict*, eager evaluation will result when the Operands is in normal order form and lazy Evaluation will be triggered when they are not in normal order form.

Figure 4.4 illustrates these two scenarios with tasks G_1 and G_2. Figure 4.4 (a) depicts a *strict* scenario in which the output of task G_1 is required for input of task G_2. Task G_2 will not execute without this input. This is *eager* mode of graph sequencing. In the *non-strict* sense, illustrated in Figure 4.4 (b), task G_1 represents the operand of task G_2. G_2 can be Predecessor in this *lazy* scenario. In certain cases, G_1 need not be evaluated at all.

In Figure 4.5, Multi-relationship between the graphs are depicted. In this case as well, *strictness* of the port determines Predecessor-Successor relationship.

In the Lazy way of graph sequencing in OOCG, Predecessor becomes Successor as the Successor becomes lazy. Thus, in lazy evaluation, Predecessor and Successor change their roles. The existence of Predecessor is used as the mechanism to drive the execution of Successor.

Transferring of data between the tasks is often performed using data *pipelining* in this pattern. Apart from *pipelining* of value, OOCG can also support data *streaming* between the tasks in which the values will be transferred in bulk as *streams*. In streaming, when the first value arrives as an operand to a task node, the task fires. In order to stop the task from executing twice, the Operand will be *deconstructed*. As soon as the first execution is finished, it paves way for the second set of value in the stream. The task fires again and makes way for the next stream of values.

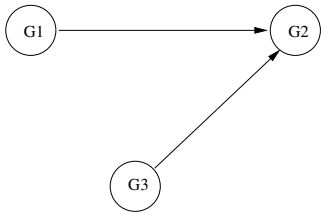

FIG. 4.5. *G_1 is Predecessor to G_3. If the port in G_2 is strict, G_3 and G_1 are Predecessors to G_2. Strictness of the port determines Predecessor-Successor relationship.*

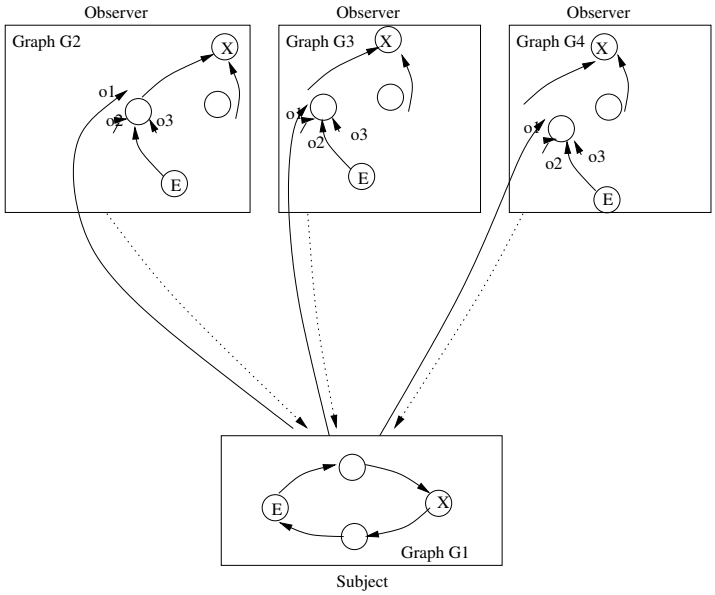

FIG. 4.6. *Observer pattern. Subject Graph* G_1 *notifies Observer Graphs* G_2, G_3 *and* G_4.

4.1.2. Multiphase Observer Pattern. This pattern is based on the Observer pattern [18] introduced by Gamma et al. **Multiphase Observer Pattern** is an extended version of Observer pattern with Multiphase concepts. This is categorised as a *Behavioural Pattern*.

Name of the Patter. Multiphase Observer

Also Known As. Publish-Subscribe

Motivation. If there are multiple dependent tasks and if a task's state change affects the state of other tasks it is important that there should be notification mechanism between the tasks so as to ensure a proper synchronisation.

Description of the Problem. Figure 4.6 illustrates the problem addressed by this pattern. Task G_1 is dependent on tasks G_2, G_3 and G_4. The execution of task G_1 affects the decision making within the tasks G_2, G_3 and G_4. A synchronisation mechanism is needed in this state change event.

Solution of the Problem. The task upon which other tasks are dependent is known as the **Subject** and the dependent tasks which observes the state change event from the Subject are known as **Observers**. A Subject can have any number of Observers. Whenever a Subject's state changes, it notifies its Observers. In response, each Observer synchronise its state with the Subject's state.

Implementation. Establish a hash table to maintain Subject-to-Observer task mapping. For each change the Subject notifies the Observer tasks. This notification mechanism can be modelled as **push** or **pull** [18]. In the **push model**, the subject sends detailed information to the Observers about the state change, whether they require it or not. In the **pull model**, however, the Subject sends minimal notification and Observers enquire explicitly thereafter.

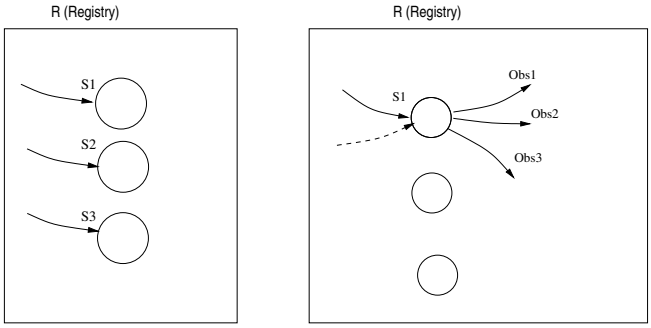

FIG. 4.7. *OOCG implementation of Observer pattern. The node in the registry are created by the Subject to reflect the state elements. Observers register as destination nodes of the Subject. By registering, they become part of the destination of the state node.*

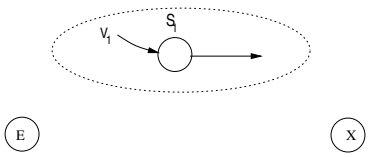

FIG. 4.8. *Subject graph. Value V_1 is being sent to the State Node S_1. V_1 is deconstructed after State Node S_1 is fired.*

Examples. In the dining philosophers problem mentioned before, the task which holds the fork acts as the *Subject* and the tasks which request forks act as *Observers*. When the Subject releases the fork, it notifies the Observers.

Observer Pattern in OOCG. In the OOCG adaptation of Observer pattern, the Subject-Observer relationship is maintained by means of registry mappings. The registry entry is created by the Subject. The nodes in the registry are created by the Subject to reflect the state elements.

Observers register destination nodes with the registry to be advised of change in state elements. By registering they become part of the destination of the appropriate *State Node* (Figure 4.7). In the Figure 4.7, Operand value V_1 must be passed onto the Observers. As Operator, Subject generates this value and Observer, as destination, consumes this value. The Operator copies the Operand value to the destination.

In the Subject graph, there is a link from the state element to the Registry (Figure 4.8). The notification operands are deconstructed when the state node is fired. When the E node of the Observer graph fires, it registers destination nodes with Registry. Similarly, when its X node fires, it removes the destination link from the Registry.

The Subject-Observer state changes can be by means of **push** or **pull** model. In the push model, Operand value converge to the Subject and passes to the Observer destination. In the pull model, Operator *pulls* the value from registry and send to the destination. After sending the value, the destination will be deconstructed. When a new Observer creates, it pulls the value from the registry. In this *Polling* mechanism, new State value arrives and overrides the old value.

4.1.3. Performance Analysis. A sample application based on OOCG is presented here to analyse the performances of the above mentioned patterns. This is an image processing application which applies compression algorithms on a series of images. The execution platform used for this experiment is WebCom [4] middleware. The main OOCG program consists of three sub-programs (*services*). When the main program is executed on the Client, the services are invoked on the hosts with service components interact each other. The initial graphs modelled from the system requirements are shown in Figure 4.9. It should be noted that these OOCG models are depicted as *Eager* graphs since the *Lazy* version is not suitable for this application context.

The Services A, B, and C are situated in different hosts. The Predecessor Successor pattern is applied between the Service CGs in order for proper execution of Service Components. Similarly, Observer pattern is implemented for interaction between the Client and the Service Hosts. In this implementation, the Ob-

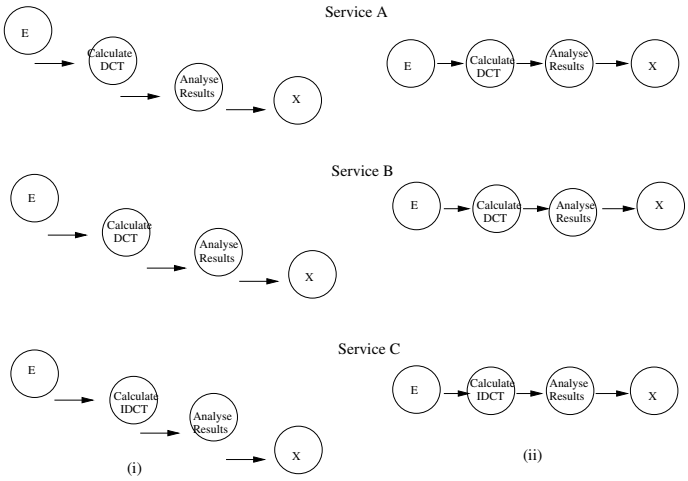

FIG. 4.9. *Initial OOCG models. (i) Lazy version, and (ii) Eager version. Since the tasks are dependent with each other, Lazy scenario is not applicable here.*

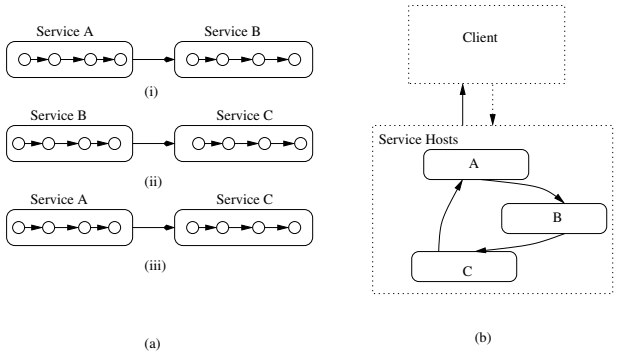

FIG. 4.10. *Some patterns implemented within the OOCG application. (a) Predecessor-Successor: (i) A-before-B, (ii) B-before-C, and (iii) A-before-C. (b) Observer Pattern. Client waits for the results from the Service Hosts.*

server (Client) waits for results from the Subject (Service Hosts). These pattern implementations are shown in Figure 4.10.

The performance of this setup has been observed by measuring the request-response interaction between Services. Performance is thus quantified as the average Request-Response time.

The experimental system is composed of three services and within each service, request to access other services are implemented as request threads. Similarly, responses also are implemented as response threads which will respond to the requests. The system can be fine-tuned by adjusting the number of concurrent requests. It can be observed from the experiments that the performance *decreases* with the increasing number of concurrent requests. Figure 4.11 illustrates this experimental results.

It has been observed from the initial experiments that a performance bottleneck occurs when the number of concurrent requests is high. The reason behind this performance bottleneck is due to the deadlock between the concurrent requests when their number is very large. The experiments have been re-run with a change of design; in which the Predecessor-Successor pattern connecting the Services is replaced by an Observer pattern. In this new design, a *Consumer* Service requests for data from the *Provider* and the Provider Service provides the data as soon as it is available. If the data is not readily available, the Consumer Service *waits* till that is available. With this enhanced design of the notification mechanisms, the deadlock between the Services is avoided. The updated result is shown in Figure 4.12.

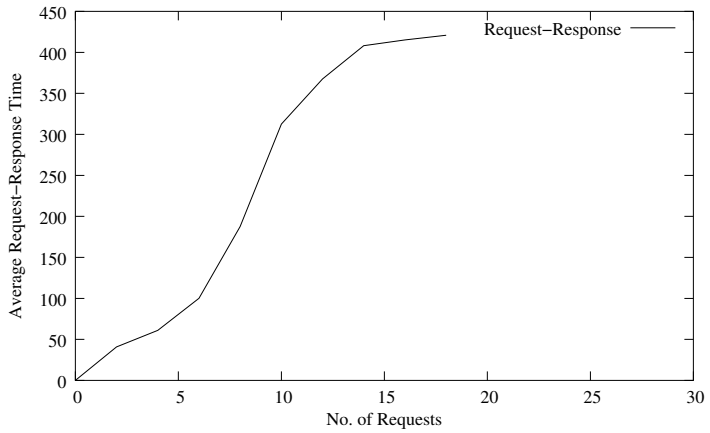

FIG. 4.11. *Average Request-Response time with respect to the concurrent Requests.*

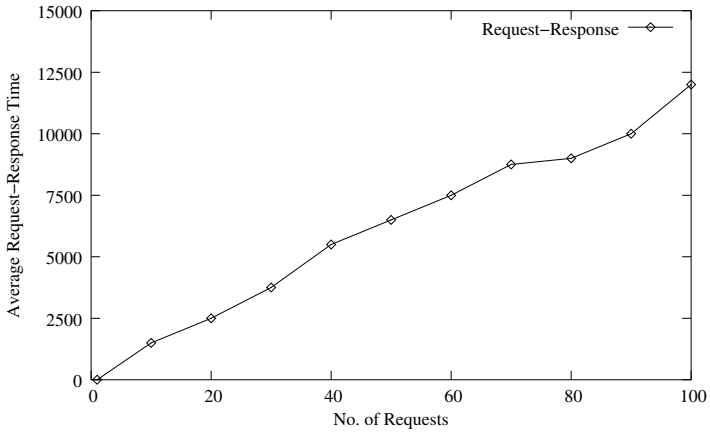

FIG. 4.12. *Average Request-Response time by removing performance bottleneck.*

5. Conclusions and Future Work. The contributions of this paper are the introduction of enhanced version of Condensed Graphs, Object Oriented Condensed Graphs, and its development support in a large scale environment. OOCG is a unified model that combines the Condensed Graphs methodology with object orientation and this leverages the advantages of both paradigms. This unified computational model is beneficial for the development of Large Scale Parallel Systems since the CG aspects allows the developer to think about parallelism and the Object Oriented aspects allow the developer to address the large scale concepts. OOCG provide implicit support of Synchronisation and Garbage Collection capabilities, which help the development of concurrent software. The Object concept and features such as Encapsulation and Inheritance enhance the Reusability and Maintainability of OOCG.

The existing implementation of Condensed Graphs has been extended to implement the OOCG features. Engineering practises have been proposed and implemented for its development in a large scale environment. OOCG model has been integrated into the Condensed Graph Integrated Development Environment, $WebCom\text{-}G_{IDE}$ [3] which is a development tool that enables visual application development and optimisation. Using the Integrated Development Environment, OOCG applications are created and deployed for concurrent and distributed environments.

Eventhough Object Oriented Condensed graphs have the potential for large scale software development, some limitations in the current model need to be taken into account. These limitations include the model's inability to create reusable datastructures, deficiency to interoperate with other paradigms and the lack of data management capabilities. These limitations would be addressed as future enhancements of the model.

REFERENCES

[1] Sunil John and John P. Morrison: Garbage Collection in Object Oriented Condensed Graphs, *Springer-Verlag LNCS, 3rd Workshop on Large Scale Computation on Grids,* LaSCoG 2007, held jointly with the 7th International Conference on Parallel Processing and Applied Mathematics, Gdańsk, Poland, September 9–12, 2007.

[2] John P. Morrison, Condensed Graphs: Unifying Availability-Driven, *Coercion-Driven and Control-Driven Computing,* PhD Thesis, Eindhoven: 1996.

[3] John P. Morrison, Sunil John, David A. Power, Neil Cafferkey and Adarsh Patil: A Grid Application Development Platform for WebCom-G, *IEEE Proceedings of the International Symposium of the Cluster and Grid Computing,* CCGrid 2005, Cardiff, United Kingdom.

[4] 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. Vol 2004(12), pp 419–438. Guest Editors: H. R. Arabnia, G. A. Gravvanis and M. P. Bekakos. September 2004.

[5] John P. Morrison, David A. Power and James J. Kennedy: An Evolution of the WebCom Metacomputer, *The Journal of Mathematical Modelling and Algorithms: Special issue on Computational Science and Applications,* 2003(2), pp. 263–276, Editor: G. A. Gravvanis.

[6] Barry P. Mulcahy, Simon. N. Foley and John P. Morrison: Cross Cutting Condensed Graphs, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA),* Vol. 3, pp. 965–973, Las Vegas, Nevada, USA, June 27–30, 2005.

[7] John P. Morrison, Philip D. Healy and Padraig J. O'Dowd: Architecture and Implementation of a Distributed Reconfigurable Metacomputer, *International Symposium on Parallel and Distributed Computing,* Ljubljana, Slovenia, October 13–17, 2003.

[8] J. Niu, J. Zou, and A. Ren: OOPN: Object-oriented Petri Nets and Its Integrated Development Environment, Proceedings of the Software Engineering and Applications, SEA 2003, Marina del Rey, USA.

[9] C. A. Lakos: From Coloured Petri Nets to Object Petri Nets, *Proceedings of the Application and Theory of Petri Nets,* volume 935, Springer-Verlag, Berlin, Germany, 1995.

[10] C. A. Lakos: Object Oriented Modelling with Object Petri Nets, *Advances in Petri Nets,* LNCS, Springer, Berlin 1997.

[11] C. D. Keen and C. A. Lakos" A Methodology for the Construction of Simulation Models Using Object Oriented Petri Nets, *Proc. of the European Simulation Multi-conference,* 1993, 267–271.

[12] Sarah L Englist: Colored Petri Nets for Object Oriented Modeling, Ph. D. Dissertation of University of Brighton, June 1993.

[13] T. Murata: Petri Nets: Properties, Analysis and Applications, *Proc. of the IEEE,* 77(4), 1989, 541–580.

[14] J. E. Hong and D. H. Bae: HOONets: Hierarchical Object-Oriented Petri Nets for System Modeling and Analysis, *KAIST Technical Report CS/TR-98-132,* November 1998.

[15] D. Buchs and N. Guelfi: CO-OPN: A Concurrent Object Oriented Petri Net approach, *12th Int. Conf. on Application and Theory of Petri Nets,* pp. 432–454, Aahrus, 1991.

[16] Gul A. Agha: Fiorella De Cindio and Grzegorz Rozenberg, *Concurrent object-oriented programming and petri nets: advances in petri nets,* Springer-Verlag New York, Inc., Secaucus, NJ, 2001.

[17] Bertrand Meyer: Object-Oriented Software Construction, second edition, Prentice Hall, ISBN 0-13-629155-4, 1997.

[18] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley Professional Computing Series.

[19] Li Xi-Zuo, Han Gui-Ying and Kim Sun-Ho: Applying Petri-net-based reduction approach for verifying the correctness of workflow models, *Wuhan University Journal of Natural Sciences,* Wuhan University Journals Press, Volume 11, Number 1, January, 2006.

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

[21] Alain Le Guennec, Gerson Sunyé and Jean-Marc Jézéquel: Precise Modeling of Design Patterns, *UML 2000—The Unified Modeling Language. Advancing the Standard,* Third International Conference, York, UK, October 2000, Proceedings.

[22] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski and A. P. Barros: Workflow Patterns, Springer-Verlag, *Distributed and Parallel Databases,* 14(3), pages 5–51, July 2003.

[23] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst and N. Mulyar: Workflow Control-Flow Patterns: A Revised View, *BPM Center Report BPM-06-22,* BPMcenter.org, 2006.

Edited by: Dana Petcu, Marcin Paprzycki
Received: April 30, 2008
Accepted: May 18, 2008