IMPLEMENTATION OF TYPE CHECKING IN WEBCOM

, ,

( )

Daithí Ó Crualaoich

September 2005

Abstract

This work approaches the problem of developing and implementing new functionality within the WebCom system, while simultaneously preserving the integrity of the software core of this system. This problem is addressed within the context of developing type checking support for Condensed Graph execution, although is not limited to this specific application. The key portion is the design and implementation of support functionality to assist third-party WebCom development.

Much of the discussion concerns introducing new programming paradigms and practices, including those of Aspect Oriented, Event-based, and Logic Programming models. Other areas covered in the methodologies and extension schemes suggested include the provision of WebCom metainformation, generic WebCom invocation, the minimization of code pollution, the implementation of WebCom modules, and a module framework including loading semantics.

The type checking example combines these various technologies, illustrating how they support third-party development of WebCom functionality. The results of this work include improved WebCom internals design, facility to employ aspect oriented and logic programming practices, type checking, metadata notations, as well as a wide range of actual applications from mobile phone WebCom invocation tools to desktop WebCom execution environments.

THE GRAYS HOPE TO WIN. STOP.
RAINBOW NEEDED URGENTLY. STOP.

— *Subcomandante Marcos*

Truth is a shining goddess, always veiled,
always distant, never wholly approachable,
but worthy of all the devotion of which
the human spirit is capable.

— *Bertrand Russell*

Apologies and Acknowledgments

I am delighted to acknowledge and graciously appreciate the fantastic assistance provided to me by my supervisor, Dr. J.P.Morrison, and by the Centre for Unified Computing. Their insightful advice has been of great use in all aspects of this work. If I have failed to convert this advice into a document equal to its measure, then I am solely at fault.

I also want to appreciate my internal and external readers for their efforts. And especially, I should like to apologise for the typographical errors and stilted prose to which I have subjected them. I have little to offer in my defense, save to express the disheartening frustration that every draft revision made to defeat these evil mistakes, only seemed to introduce new ones. My admiration for perfect prose has reached new levels.

This work has been funded by Science Foundation Ireland and by the National Development Plan.

Dedicated to the College. Thank you, most grateful thank you, Plato, for your Academy. May there always be ivory towers and dreamy spires. Although, we might be able to make do without Oxford.

Contents

1 Introduction

This dissertation concerns type checking in the WebCom implementation of the Condensed Graph model. It also variously deals with software module design, the use of aspects in event driven programming, the specification and execution of task descriptions, and other topics that may also arise.

It is best to begin here by describing the Condensed Graph model and the WebCom implementation in outline at least. Exhausting detail will be provided at the appropriate points in the following chapters, so herein it is preferable to outline the broadstrokes of the Condensed Graph mission.

The following chapters, on the supporting Information Framework, on the event system, and on module organization are somewhat new components[1] of WebCom designed or improved to meet the type checking implementation difficulties in a uniform and generalisable fashion. They represent interesting adventures in their own rights also and merit the careful study. This dissertation concludes with an examination and implementation of the core concern, the new type checking system. So toward that end, this dissertation begins by considering and outlining the Condensed Graph model.

1.1 What are Condensed Graphs?

The Condensed Graph (CG) model is a computation paradigm based on directed acyclic graphs (DAGs). While there may be superficial similarities with existing DAG based workflow packages, and although workflow is an application area in which Condensed Graphs excel, Condensed Graphs are *not* simply yet another

workflow solution. This view neglects particular features of the Condensed Graph model that conspire to produce a highly flexible and applicable programming environment.

§1 Granularity. *«A treatment of the wide applicability of the Condensed Graph model, and an outline of the various levels of operation granularity supported thereby. Examples of existing application platforms»*

As the Condensed Graph system was initially designed to support parallel computation, the original target architectures were larger multiprocessor systems. However, given the aggressive adoption of parallel techniques throughout the spectrum of computer design, the Condensed Graphs model can be comfortably applied to systems at all levels in the computer architecture food chain, from embedded systems to Grids.

The node object constitutes the fundamental parallelisable software artifact in the Condensed Graphs model. Nodes themselves are in fact just containers for the elements that define the desired computational task. One of these elements being the operation component makes control of the parallelisable task granularity possible. Key to the Condensed Graph model is easy and powerful accommodation of operations with variable granularities. Grain sizes ranging from the trivial to the vast and time consuming are compatible.

This flexible grain size is both practical and feasible also. Condensed Graphs can and have been applied in Field Programmable Gate Arrays (FPGAs), in mobile and embedded devices with J2ME, in desktop and serve systems, in networks of desktop systems, in Beowolf clusters and in Grid Computing.

Operations themselves need not be of overt numerical computation character either. Scheduling, workflow, security based, information organisation, database transactions, middleware invocation, expense account authorisations or literally any possible computation task or combination thereof is suitable.

§2 Parallelism. *«An examination of the role of parallelism in the development and operation of the Condensed Graph model»*

The flexible operation grain size is just one aspect of the parallelism support provided in the Condensed Graph framework. More interesting than nodes themselves, is how nodes can be combined.

The Condensed Graph model allows for nodes to be scheduled in imperative, lazy, or eager evaluation modes, or uniquely, in any combination of these modes. A computation designer has precise facility to specify how individual nodes are to be computationally executed. For example, a designer may indicate certain nodes that arrange a seldom used computation are to be executed lazily if required, while at the same time, the designer can indicate that the critical path of the graph based computation should be performed eagerly in order to minimise the total execution time of the graph.

Crucially, this blending of speculation with conservative execution is garbage free. The model semantics provide for the automatic and efficient removal of unrequired speculation and associated junk.[2]

§3 Paradigm. *«Discussion of the Condensed Graphs model from the perspective of programming paradigms and methodologies»*

In so far as the Condensed Graph model is language independent, and combines the imperative, lazy and eager computation models, it forms a separate computational paradigm. Its implementation as an augmentation of existing programming languages mirrors that of Object Oriented (OO) systems.[3]

The primacy of the intrinsic graph representation should be emphasised since it distinguishes the Condensed Graph paradigm from other augmentative paradigms such as the OO model. The Condensed Graph model is inherently wedded to the graph executable artifact whereas the OO model arranges computation by client-server message transactions. The structure of an OO execution is not an explicit entity in the paradigm in the same way as a Condensed Graph is central to the Condensed Graph model.

There is also large variation in how the Condensed Graph paradigm is applied to arrange computation. Condensed Graphs are often the main execution mechanism driving a parallel or serial computation. However, they are also often used as middleware "glue" to leverage distinct systems into collaborative execution.

§4 Word of Warning. *≪A cautionary tale about the Condensed Graph learning curve≫*
It ought to be noted that the flexibility of the Condensed Graph system comes with a price, not necessarily paid in computational overhead.[4] The initial exposition of the Condensed Graphs system and the mechanisms by which it acquires much flexibility are a typical stumbling block for novices. Much of this learning curve can be attributed to basic lack of familiarity with DAG based systems in general, but it remains true that learning the correct mode of thinking in Condensed Graph design does require some time.

This is not especially surprising considering that novice users face similar difficulties with other computational paradigms such as OO, Aspect Oriented Programming, Functional Programming, and Logic Programming. As with many of these technologies, the implementation of intuitive tools to support the development process is essential to alleviating initial obstacles to adoption. Indeed, tools such as Condensed Graph visual designers and debuggers are the focus of keen current development.

§5 Section Map. *≪A brief guide to the remaining portions of this chapter≫*
What follows constitute a effort to provide a sufficiently detailed operational description of the Condensed Graph model for the purposes required in this dissertation. Many important implementational and machine details are neglected. The reader is referred to other sources for more complete understanding of the machine specifications. [Mor96, MRb, MRa]

1.1.1 The Node Execution Triple

Computation is engineered by the aggregation of a computational triple consisting of an operator with which to act, operands upon which to act, and destinations to which the computation output is propagated. Nodes deal precisely with this conjunction and can be thought of as empty socket boxes onto which these separate computational elements are attached. Nodes exist within graphs, the arcs of which indicate the mechanism by which the computational elements are delivered to nodes.

Once the execution machine has conspired to present an operator, the correct number of operands and at least one destination to a node, the node is then capable of firing(or reducing.) The firing of a node, scheduled in some fashion by the execution machine, triggers the execution of the computation described by the node. The result of this computation is then forwarded to the indicated destinations.

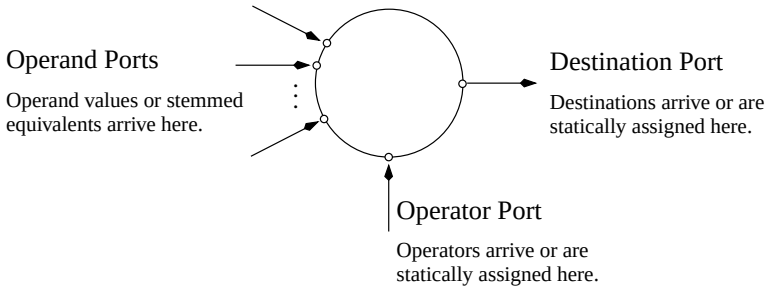

Figure 1: The Anatomy of a Node.

We examine the principle components of a node below in more detail. See Figure 1 for a graphical representation of a node and the ports at which computational elements are attached.

§6 Operator Ports. *≪Description of the operator port and its role in the Condensed Graph model≫*
The actual contents of a node's operator port depend on the Condensed Graph model implementation, but might for instance be a function pointer in C, a `Method` object in Java, or a lambda expression in a functional programming backed Condensed Graph implementation.

There is nothing particularly special about this port. Its contents are software artifacts just the same as the contents of the operand or destination ports that will be described shortly. This observation highlights that the output of node computations may be used as operators and not just as operands to subsequent nodes.

Although dynamic operators are not typically currently employed in Condensed Graph applications, they represent a powerful computational technique. Dynamic operator generation share many parallels with the expressive realisation of functions as first order objects in functional programming. There is current work on the use of dynamic operator generation to produce datastructures natively in terms of graphs. These datastructures inherit many benefits from the graph model.

§7 Operand Ports. *≪A discussion of the operand ports and their role in the Condensed Graph system≫*
Each node has one or more operand ports, according to the arity of the node's operator. Operand ports are the attachment points for the input data of the computational triple represented by the node.

It is not entirely clear, nor does it really matter how a node acquires the correct number of operand ports for its operator, especially in the dynamic setup. It can be imagined that the operand port configuration of a node is dynamically updated when an operator arrives at the operator port.

Operand ports are zero origin integer indexed for legacy implementation reasons.

§8 Destination Ports. *≪A discussion of destination ports and their role in the Condensed Graph system≫*
The final constituent part of a node is a single destination port. Destination ports contain references to the operand ports of other nodes and indicate where the node computation results are sent.

As before, the contents of destination ports are also regular software artifacts of the implementation system. Unlike the case of dynamic operators, dynamic destinations are likely of less utility to the end programmer and as such, destinations are usually assumed to be statically assigned.

Because of this assumed static assignation, it is tempting to view destination ports as providing an arc to an operand port over which node computation results pass. This view is unfortunately supported by the standard graphing notation that will be introduced below. Helpful as this picture appears, it is a deceptive view of destination port operation. There are specific semantic actions, such as stemming and grafting, that can be applied to destination ports but which simply do not have a consistent interpretation in terms of the result value arc path view. It is a common error to misconstrue destination ports as simply one end of an arc over which results travel.

A node possesses a single destination port. This is not to say that nodes can only forward their computation results to a single other node. Rather the software contents of a destination port are of a Composite [GHJV93, GHJV95] pattern type and can be used to refer to many distinct graph locations. Note, however that only a single destination location is required for a node to be fireable.

Thus far, single valued operations have been implicitly assumed. Whilst this is required by the current graph model, there is some ongoing work to extend the model to support multiple valued operators. To this end, multiple destination ports, one per output, are required together with more complicated node firing criteria. Regular multiple output nodes are not be of concern herein.

Before leaving the topic of multiple destination nodes, it must be noted that there are certain such nodes permitted in the basic model also. These nodes, called E nodes, are special purpose nodes that play a specific role in memory allocation and in the condensation process. These are the only multiple destination nodes permitted in the basic model.[5]

1.1.2 Graphs

Arcs in the Condensed Graph DAG computation representation are induced from the port contents of nodes in the graph. A destination port is connected to an operand port with an arc if the operand port is one of the destinations on the destination port. Operator and destination ports are also the endpoints for incident in arcs that convey the node operator and destinations respectively. Static operators and destinations are represented as primitive values on these arcs.

The graph topology thus constructed must be that of a DAG. Cyclicy is not permitted.

§9 E and X nodes. *≪Special purpose memory allocation and graph delimiting nodes≫*
There are two special node types, denoted Enter(E) and Exit(X). The E node represents the graph computation entry point and has as many operand ports as operands to the larger graph computation. For example, a graph to sum two integers will have two toplevel operands, and so the corresponding E node also has two operands.

An E node has as many destination ports as it has operand ports. The operation of the statically assigned operator is trivial, being a simple copy. On E node executed, each operand is copied from its operand port to

the destinations contained in the corresponding destination port of the E node. So the first operand is copied to the destinations in the first destination port, the second operand to those in the second destination port, etc.

E nodes also have a memory management function in the machine semantics. The triggering of an E nodes is a cue for the execution machine to step into a new execution or stack frame. Furthermore, the E node is also a marker node for the condensation process.

X nodes possess a single operand and a single destination port, together with a static operator. On execution, the X node copies the input operand to the output destinations. The idea is that this output constitutes the output of the whole graph execution. So whereas the E node represents the graph entry point, the X node represents its exit point.

In terms of machine semantics, the X node cues the machine to deallocate and clear the graph execution or stack frame. It is also the concluding marker node for the condensation process.

§10 Example Condensed Graphs. *«Illustrative example Condensed Graphs and explanations»*
Consider the graph in Figure 2, which describes a computation with two inputs and a single output. This graph functions to sum the two, presumably numeric,[6] input parameters and pushes the summation value to the output.

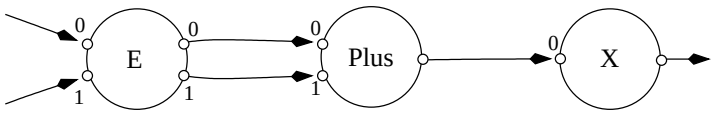

Figure 2: Example Condensed Graph.

Note that Figure 2 incorporates the convenient shorthand practice of writing static operator names inside of nodes, rather than indicating passage via the operator port.

As before, caution is required when considering the arcs that connect destination ports to the operand ports they contain. There is a strong misconception that output results pass over the destination port(commonly incorrectly referred to as an “output port”) and along the arc to the endpoint operand port. The viewer must always be aware that this interpretation obsfucates the real mechanisms of the destination port.

It will be seen shortly that destinations may be stripped from nodes by the stemming process. Although there is a graphical notation for this process, it needs to be emphasised that the plain arc notation obscures the node stemming facility. Misunderstanding the destination port semantics is a common error made by newcomers to the Condensed Graph system. Unfortunately the standard graph notation fails to highlight the special status of the destination port.

It is instructive to consider the reduction of nodes in the example graph and the illustration of operand dataflow in Figure 3. Suppose the E node operand ports are populated with the numeric values x and y at the commencement of a graph execution. It is not of terrible concern for now how the toplevel actually populates these inputs.

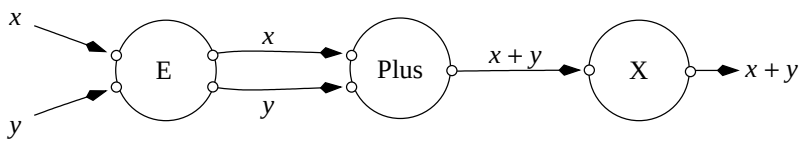

Figure 3: Example Graph Datapath

Initially, only the E node contains a full complement of computational triple elements and is the only node capable of firing. Note that the Plus node is incomplete because operands are missing. This is despite what the diagram may suggest on casual inspection. Similarly, the X node is incomplete.

When the E node fires, its destinations are populated with the input operands. In particular, this means that the Plus operand ports are populated, making the Plus node itself fireable. When the Plus node itself fires, its destinations are populated with $x + y$, making the X node complete. The firing of the X node pushes $x + y$ onto the toplevel output.

This example illustrates a computation that is imperative in nature. This is a consequences of the linear graph structure. The real value of Condensed Graphs comes about when there are multiple execution branches possible at a given moment as in the case of Figure 4.

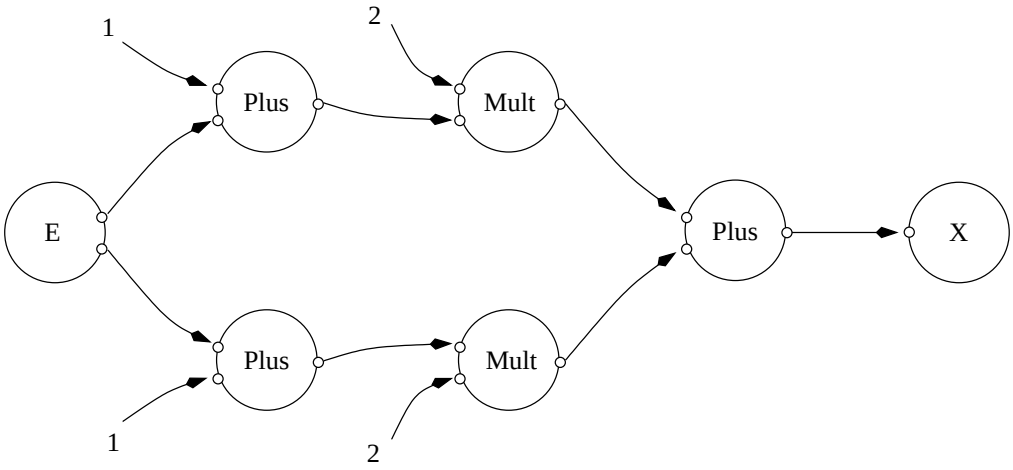

Figure 4: Example Condensed Graph II.

Following the triggering of the E node in Figure 4, both of the Plus nodes are complete and ready to be executed. Any order or form of parallel or serial execution scheduling can be employed to evaluate nodes that are not strictly ordered in the computation DAG. Nodes represent independent computational triples, whereas the graph determines the data dependencies present.

1.1.3 Stemming and Port Strictness

§11 Stemming. *≪Introduction of the Condensed Graph mechanism for influencing the character of a computation≫*
Stemming is the design time process of temporarily removing a destination port's contents so as to prevent the node containing that port from firing. The destinations are not removed completely, the mechanism simply prevents the node from firing until some later computation explicitly requires it to fire. The reverse operation, that of replacing destinations, is called grafting.

Where before stemming, a destination port is occupied, after stemming it helps to think that the operand ports to which the destination port points have been occupied by a composite operand value. In this way, although one node is stripped of its destination port element, other later nodes are populated with operand port values.

Stemming facilitates lazy evaluation by automatic grafting. The grafting process can be triggered on a stemmed node by the firing of nodes that depend on the result of that stemmed node. Stemming is also most primarily used in conjunction with recursion and IfEl nodes as will be seen shortly.

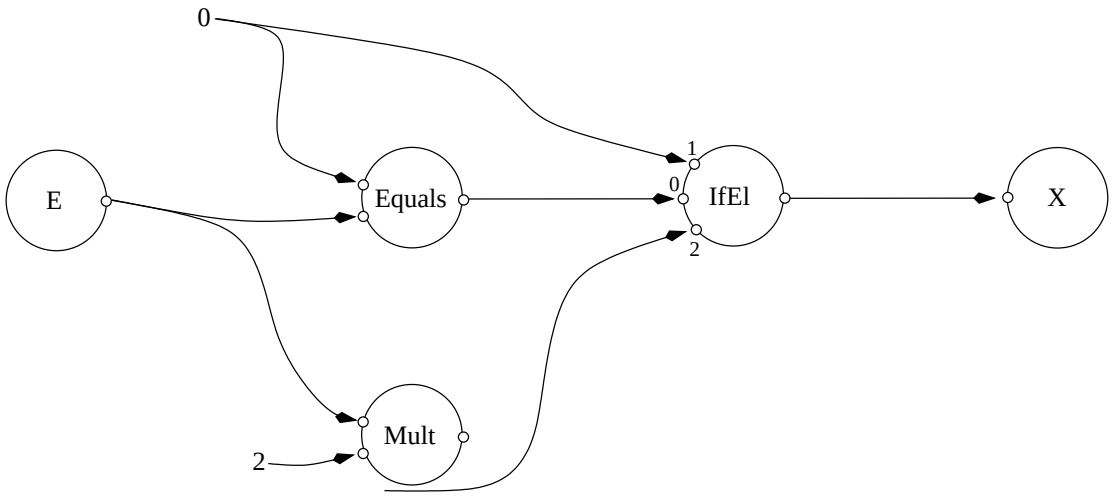

Figure 5: Example Stemmed Node.

Consider the example in Figure 5, the purpose of which is to double the input parameter. The graph first performs a comparison and will then only compute the multiplication if the input parameter is nonzero. This example is deliberately contrived in order to illustrate the stemming process.

The previously unseen operators function as follows. Equals compares two operands for equality, returning an indicative Boolean result. The Mult operation multiplies the two operands. The IfEl operation[7] achieves the basic branching in the Condensed Graph system. Operand zero is a Boolean indicating which operand to use as output, operand one if true, operand two if false.

Aside from the new nodes, the main point of interest is the stemming of the Mult node to the IfEl node. It is very common to stem IfEl branches in order to require their lazy evaluation. To all purposes here, even when the E node is fired, the Mult node will remain incomplete. Thus, it is assured that the multiplication is not done eagerly ahead of the equality test. During execution, the equality test is executed first, then the IfEl triggered. Note that the IfEl node is completed once the Equals node has fired but that the Mult node is still incomplete without its destination.

The IfEl node operation depends on whether the contents of the input selected operand port are primitive or not. In the case that the contents are primitive, the value is simply copied to the output destinations. If the selected operand port contains a stemmed node, this stemmed node is "passed through" the IfEl node. That is, the destinations of the stemmed operand node are rewritten to remove the IfEl node and replace it with the destinations of the IfEl node. Furthermore these destinations must be arranged to have the operand node as a stemmed operand. This ensures the stemmed node is properly redirected to the destinations of the IfEl node.

In practice, the implementation is not so involved. Since the stemming mechanism is implementation dependent, it can be designed in such a way as to easily facilitate this scenario. For instance, the WebCom system simply moves the referenced node to the operand ports of the destinations, in effect treating the stemmed node as a primitive value[8].

The style of processing used by the IfEl node tends to be atypical. In general, a node with a stemmed operator usually coerces the triggering of the stemmed node to produce a primitive value. This coercion is done automatically via the use of port strictness and coerced grafting.

The behavior of the IfEl operands on the non-Boolean operand ports can be compared with the behaviour of currying in functional programming. Although, the comparison is entirely superficial, the IfEl does perform a basic higher order function that does not require knowledge of the actual operand contents, and as such does not require the explicit computation of these values.

§12 Port Strictness. *≪Port strictness tags are a method to automatically coerce lazy computations≫*
Operand port strictness and the stemming process are closely linked. Operand ports may be either strict or nonstrict, indicating whether primitive values are required on the port for the execution of the node operation. Although strictness is associated with operand ports of the node entity, the artifact really depends on the operation not the node.

A strict port requires a primitive value. A nonstrict port can take a primitive value or a stemmed node. In the IfEl node example, the Boolean test is a required primitive operand in order to decide which alternative branch to use. Therefore this operand port of an IfEl node must be strict. The contents of the non-Boolean operand ports need not be primitive. Since the IfEl operation specifically processes stemmed nodes, these non-Boolean operand ports of an IfEl node should be nonstrict.

§13 Coerced Grafting. *≪Explanation of the use of port strictness to coerce grafting≫*
Stemmed nodes on strict operand ports of otherwise fireable nodes must be grafted in order to permit firing. This grafting returns the destination of the stemmed node and may cause it to become complete, or cause it

to coerce completion from its stemmed operands. The firing of this node then populates the original operand port and facilitates the triggering thereof.

This coerced grafting is performed automatically by the execution machine as required. Essential grafting involves grafts that are required in order to move the computation forward. There is also the possibility of using discretionary grafting in lazy graphs to convert lazy computations into eager computations and to throttle computation. For example, if during an execution, the machine determines it can avail of additional computation resources but that there are not enough fireable nodes to maintain useful throughput, then the execution machine can discretionary graft stemmed nodes in order to raise the amount of fireable nodes in the graph. Since fireable nodes correspond directly to exploitable parallelism, discretionary grafting is a parallelism throttle.

Stemming and grafting can be viewed as inverse operations that move a single entity between destination ports and operand ports. At times, it is more beneficial to have this element on the destination port location in order to push the computation forward aggressively. At other times, it is more useful for this entity to reside on operand ports and to encourage little additional computation.

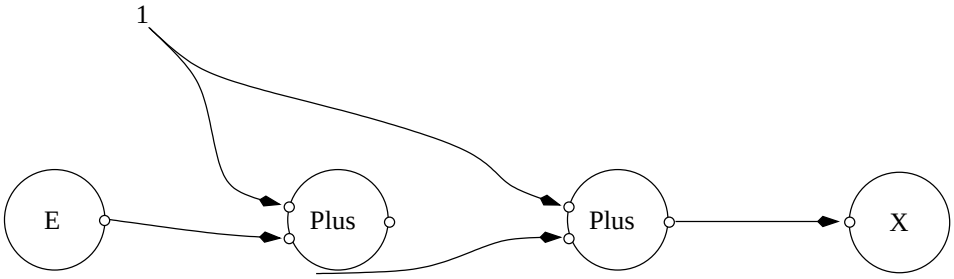

Figure 6: Example Stemmed Node II.

Consider the example graph in Figure 6 that adds two to the input by way of a stemmed Plus node chain. The execution proceeds by firstly firing the E node and populating the input of the first Plus node. [9] Due to stemming, the first Plus node is *not* complete. The only complete node remaining is the second Plus node. But since both node operand ports of the Plus node are strict, the second Plus node will have to graft the first Plus node in order to fire.

See Figure 7 for the graph arrangement at this point. The coerced graft will make the second Plus node incomplete, but will complete the first Plus node and thus coerce its execution in a lazy fashion. The first Plus node then fires and populates the second Plus node operand port. The computation is completed in a straightforward fashion from this point.

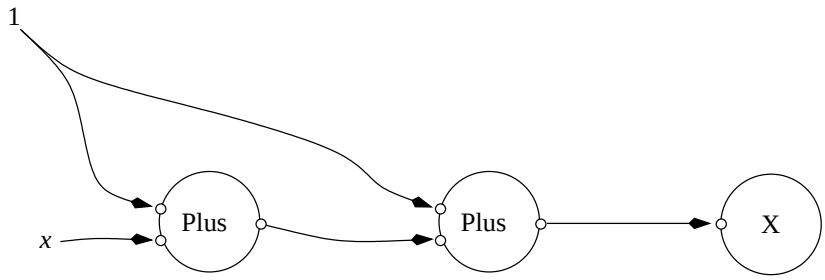

Figure 7: Partial Execution of Example Stemmed Node II.

1.1.4 Condensation

Condensation is the recursive technique of embedding graphs as nodes within other graphs. A graph is converted to an operator and placed on the operator port of a new node. On triggering, this operation evaporates into the condensed graph description as a graph and connects the operands and destinations of the node to the new graph. This operator view of condensation and evaporation is strictly operational and does neglect some semantic details that are not of concern here.[10]

§14 Condensation Example. *«An example of the condensation and evaporation processes»*
Consider the example recursive graph in Figure 8 which implements recursive base two exponentiation. For toplevel operand x, the graph returns 2^x. The recursive computation is maintained in the condensed Pof2 node. On execution, this creates a new instance of the Figure 8 graph in place. See Figure 9 for an illustration of the unrolled computation.

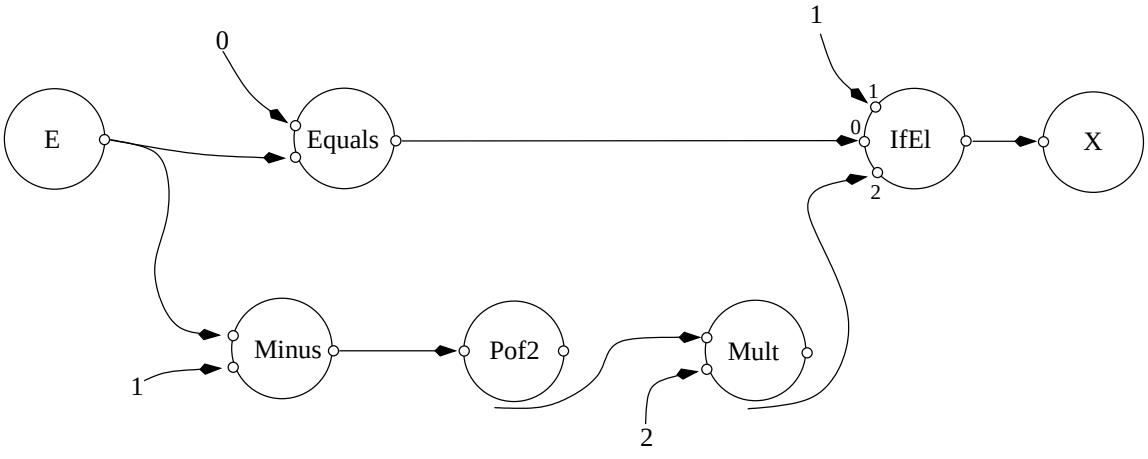

Figure 8: Example Recursive Graph.

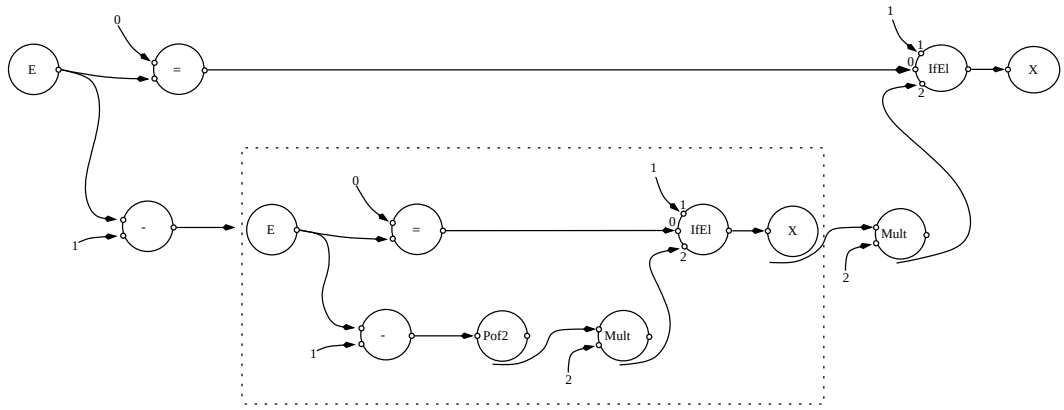

Figure 9: Unrolled Example Recursive Graph.

§15 Memory Problems. *≪A note on the memory issues involved in condensation≫*
Examination of Figure 9 raises the question of efficient memory allocation. Each time an E node is passed a full new activation frame or execution frame is required. In deep condensation sequences there will be an Condensed Graph version of recursion overhead. There is active work on deploying efficient iteration schemes in the Condensed Graph implementations to alleviate the current difficulty in using Condensed Graphs to do large scale iterative computation.

1.2 What is WebCom?

WebCom is the primary triple manager (TM) or execution machine for the Condensed Graph system. It is currently in its second major development incarnation and is being prepared for open public release. As much of this dissertation details various WebCom internals, only a brief introduction is outlined below. More in depth discussions are saved for the appropriate later occasions.

§16 Condensed Graph Triple Managers. *≪The specification for execution machines in the Condensed Graph system≫*
A triple manager is a machine for the processing of Condensed Graphs into instructions and their subsequent execution. This machine must maintain graph memory datastructures and identify executable nodes in managed graphs. These executable nodes must be marshelled for computation, scheduled, executed and the results thereof propagated.

The Condensed Graph system makes no demands of the scheduling or execution order save executions must respect the graph topology. At any point a number of nodes may be fireable, but the triple manager has complete discretion as to the node firing order.

In addition to basic node scheduling, triple managers may optionally implement features like the throttling support as described by speculative grafting, the exchange of messages and work with other triple managers, additional security features, or any other features that a developer may find interesting or useful. There are a number of existing triple manager implementations including development simulations, sequential programming triple managers, PVM parallel triple managers and WebCom. There is also developed research on hardware triple manager implementations.

§17 WebCom Module Core Architecture. *≪The main Condensed Graph execution machine≫*
The module architecture of WebCom will be the focus of detailed consideration later, but a cursory examination of the basic modules here will help outline the major features of WebCom. The seven main core modules in WebCom are the Backplane Module, the Connection Manager Module, the Engine Module, the Fault Tolerance Module, the Scheduler Module, the Load Balancing Module and the Security Module. Each of these is covered briefly in turn in the following paragraphs.

All WebCom modules are planted in a Backplane Module. This functions as the main bootstrap module and is almost entirely managerial in purpose, its only other main role being the internal routing of module message communications. The Backplane Module is considered as a module for implementation convenience rather than because it has inherent module status. Indeed, historically the Backplane was not a module.

WebCom is triple manager software that coordinates the parallel execution of Condensed Graphs. Collaborating WebCom instances communicate over regular network socket interfaces via their Connection Manager Modules. WebCom only loosely enforces network topology with a parent-child relationship and can be used in a wide range of P2P and tree configurations.

The basic triple manager functions of WebCom are contained in the Engine Module. There have been numerous Engine modules written to support WebCom and allowing the fundamental triple manager commands and instruction sets to be extended. For example, there are Engine Modules to interface WebCom with middleware solutions such as COM, DCOM, EJB and Corba. WebCom itself also operates a middleware solution leveraging the named current middleware solutions.

WebCom maintains recoverability and tolerance in the face of network faults via the use of the Fault Tolerance Module. This module journals work exchanges as part of WebCom computations in order to repair lost communications.

The Scheduler and Load Balancing modules conspire with the Engine module to schedule nodes into instructions and order them for execution. The Load Balancer has a role in arranging an even distribution of work between collaborating WebCom machines.

The final essential module for a WebCom configuration is a Security Module. Security has been a built in component of the WebCom system from the earliest stages. The authority of a WebCom machine to execute a particular instruction or action is always vetted by the Security module. There is an extensive authentication system. [FMQ04, QF04, FQM02, FQ02, FQO$^+$04, QCF04, FQM$^+$00]

1.2.1 Past and Future Trends

§18 Where from and where to. *«The origins and future directions for the Condensed Graph/WebCom project»*

The Condensed Graph was originally introduced in the doctoral dissertation by J.P. Morrison[Mor96]. Development of Condensed Graph and WebCom tools has been the source of numerous funded projects by Science Foundation Ireland, Enterprise Ireland, and the National Development Plan of the Irish Government. Over the course of these projects a large amount of research has been developed. [JPMP04, MKPa, MKPb, MKPc, MKPd, MPK, MC, MP, MPC, Ken04, MOH] The main WebCom system is supported by other research such as the Cyclone cycle harvesting system, the Anyware technologies and the security framework for WebCom.

In the short term, the next probable developments in the Condensed Graph model include loop unrolling, iteration optimisations, multiple outputs, datastructures and nondeterministic path merge operations. Regarding the WebCom technology, it is under active development in preparation for public release. Current development works includes node targeting, debugging support, and Grid information management.

Chapter Notes

[1]The Information Framework and the aspect based internal event systems are entirely new. The Module API develops some previous work in the WebCom software but contains new elements, notably in the use philosophy of modules and in the extended examples.

[2]The automatic removal of unwanted speculative computations represents a marked difference between the Condensed Graph system and some other workflow type systems. Clearing bad speculation is a problem that has plagued other workflow systems in the past.

[3]Comparing Condensed Graphs to the Object Oriented paradigm results in a weak analogy because the Condensed Graph model has an intrinsic graphical representation of computations that can be viewed as a program. The parallel notation of executable UML in the OO paradigm is a strictly weaker concept.

[4]The Condensed Graph model must of course incur some inevitable additional computational overhead versus traditional computation methods. Although the extent of this overhead is not clearly understood at this point, based on profiling and benchmarking there are some reasonable grounds for optimism in this regard.

[5]One advantage in incorporating multiple valued operators in the basic model is that this would partly uniformise the treatment of E nodes. In the multiple output operators context, E nodes do not require specific operation semantics as they do in the current model. E nodes would still require special treatment as part of their memory and condensation process functions, though.

[6]The input operands in the example graph of Figure 2 need not necessarily be numeric. The input operands need just be compatible with whatever operation is denoted by the Plus operation. Depending on the implementation of this operation, any input types may be permitted. For instance, if Plus is implemented using the "+" operator in Java, then the inputs might possibly be String objects.

[7]It should be noted that IfEl nodes have a special status in the execution machine mechanics. They are triple manager operations and must always be executed by the local machine.

[8]The use of stemmed nodes in operations does involve some API considerations for the operation programmers however. To handle the circumstances of stemming, the operation writers need to be provided with an API to process stemmed nodes. As it stands, treating the stemmed node as a primitive value unties a lot of difficulties at the IfEl node, but there are cases when a node writer will want to treat a stemmed node operand in a nonprimitive way. In reality, the practice of writing node operations is does not require reference to stemming in nearly all cases. Consequently, it has been possible to develop the rare cases involving stemming on a per case basis.

[9]Note that initially in Figure 6 the second Plus node is also fireable. Although it is semantically undesirable for a node to fire before the E node of its graph, this may be a practical machine optimisation.

[10]Specifically, the H and V graph definitions of condensation are not of concern herein save to mention that the E and X nodes form delimiting markers for the condensation tree.

2

Information Framework

The Information Framework is a new component of the WebCom system, designed to implement a step in the production of a supported type system within the WebCom realisation of Condensed Graphs. The information requirements in this typing application illustrate the value of general purpose metainformation mechanisms within WebCom.

The prototype Information Framework is outlined below and comprises of a component augmentation of the present system that may be extended to support other non-typing applications. Also discussed are examples of metainformation objects, together with potential and realised applications of the information system. Note that this framework is an enabling component, and its primary utility is in the realisation of a strict separation between typing data and graph data representations within WebCom.

2.1 Meta Information Motivation

At present, there is a lack of both available runtime and before time documenting data for Condensed Graph elements and WebCom modules. The Condensed Graph elements consist of the many software objects used to represent Condensed Graphs within WebCom, namely nodes, ports, operands, operators, graphs, and so forth. The WebCom system operates on these elements as mandated by Triple Manager specifications.

These objects have a solely execution orientation in that they are designed and optimised to enable WebCom to efficiently perform Triple Manager actions. Type checking is intended as a strictly optional operation

and as such should introduce only a limited execution overhead. The implementational transparency of the meta object notations to the existing WebCom system is also a key requirement.

Efficiency and transparency concerns notwithstanding, it is furthermore essential that these software artifacts be able to educate other objects as to their attributes and capabilities. It is precisely this facility that provides for the unification of distinct application areas within the WebCom system of software. For instance, the exportation of typing information from the core elements, upon which Triple Managers operate, can have additional uses outside of the type checking system. In fact, the typing information is currently used to provide a rudimentary documentation system in the style of JavaDoc. This documentation system could be incorporated, in automated form, into other tools in the WebCom suite, a point that is further discussed with reference to enduser development support later in this chapter.

The existence of a metadata resource within the WebCom runtime environment is also of use to future developers. The intention being that the Information Framework will provide a basis for a more expanded metadata system. The exact constituent data elements of this future system would depend on developing requirements within WebCom research but the Information Framework will implement a structure allowing for the straightforward adoption of new metadata streams and categories.

At present, new metadata requirements necessitate overhauling core WebCom software. The type checking application will serve a wider design purpose if it provides a basis for future metadata augmentation and minimises refactoring tasks. From a design perspective, the Information Framework facilitates a cleaner core architecture, in that metadata notations may be used to achieve requirements which previously might have been implemented in the core software, incurring unwelcome software coupling.

In spirit, the Information Framework compares closely to JavaDoc annotations, although in practice it is much closer aligned with the Java BeanInfo system. The framework is intended to function in a manner akin to the JavaDoc support in the Java 1.5 release. This Java 1.5 release includes an updated JavaDoc tool incorporating many improvements trailed in the XDoclet project. For instance, developers can implement their own JavaDoc tags or annotations and extract metadata from these custom annotations at runtime via the use of a query API. Analogously, it is proposed here that graphs, nodes and other elements export available metadata to other portions of the runtime environment via a simple query interface.

It is also conceivable that metadata items may be dynamic. For the purposes of type checking, static notations suffice, but in more general purpose system it might be advantageously to facilitate metadata mutation. Such mutation might consist of attribute value mutation or of modifications to the metadata schema itself. Attribute value mutation would be straightforward to implement, the main considerations being the mutation mechanism API. Metadata schema mutation is more involved and does not feature in this prototype.[1]

From a pure object oriented perspective though, attribute value mutable metadata compromises data value encapsulation. The mutable value metadata design encourages a distinction between graph objects used for graph execution purposes and their descriptive peer counterparts. In the present WebCom implementation, graphs and nodes are monolithic entities containing both active and passive portions. Picture this as core graph execution objects, containing the active data items,[2] around which are wrapped peer objects containing

passive and descriptive values. Mutable metadata would mean having active data items in the wrapper peer.

The proposed Information Framework will help in separating these passive and active concerns, but will also have a wider design benefit in minimising class pollution. Despite a sometimes task oriented development, the WebCom system design possesses a strong identifiable core set of classes. However, these core elements are very vulnerable to feature creep, something which the Information Framework as outlined here can help discourage. In addition to providing an alternative to core class extraneous functionality pollution, refactoring efforts can leverage this metadata system to cleanse current core pollution. The use of notational based constructs will also support superior design in future system augmentation.

As a final motivating remark, note that the actual prototype implementation incurs only a small cost. From this perspective, even if the system is underutilised there is still a neglible penalty in its incorporation.

2.2 Parallels with BeanInfo

While the design intentions of the Information Framework mirror those of the newer JavaDoc annotations, the actual implementation differs somewhat. The Information Framework follows the implementation model of the Java BeanInfo API, one of the APIs making up the JavaBean component object middleware system.

A JavaBean is a Java object written according to a certain predefined format, so enabling the automatic discovery of object attributes and events sourced within that object. The BeanInfo framework provides support for this data to be otherwise specified and populated, through the use of a `BeanInfo` object. Given that Java reflection capabilities can provide default marshelled `BeanInfo`s, JavaBean developers themselves do not typically use BeanInfos directly. Nevertheless, this option remains available. Whilst the automatically generated `BeanInfo`s are usually sufficient for the demands of JavaBean applications, it is the handwritten `BeanInfo`s that provide the implementation model for the prototype WebCom Information Framework.

JavaBeans are associated with `BeanInfo`s on a per class basis, a design mirrored in the Information Framework by associating WebCom graph execution objects with an `Info` object on a per class basis. This minimises overhead in that instantiations of a particular graph element class can share a common `Info` object.

Further mirroring the JavaBean model, there is provision for automatic population of WebCom `Info` object attributes via reflection. In fact, JavaBean marshelling cues and reflection schemes suited WebCom `Info` object reflection so satisfactorily that they were borrowed with minor modification. This lending allows users of JavaBeans to immediately recognise corresponding concepts in the WebCom `Info` system and to leverage an existing view of the JavaBean system into an initial view of the WebCom Info system.

The two main reflection cues employed are a no-argument constructor for initial blank `Info` population, and attribute element reflection in the base class to provide description data for the `Info` class. Since graphs elements do not currently source events in their operation models, there is no analogue for JavaBean events in the WebCom `Info` system. Though, event dispatch capabilities may be desirable at some point. So, in the absence of generated events, the WebCom `Info` system does not implement reflection cue constructs from the JavaBean system that are solely event system related.

An important caveat on disregarding events within the Information Framework concerns module events. An extension of the current WebCom module configuration system raises a use for module metadata[3] and consequently for module `Info` objects. Modules are logically the source of interesting system events and module `Info` objects are thus candidates for event dispatch documentation within the Information Framework structures. It should be noted that module event generation and dispatch is handled within an entirely separate design philosophy and so may be omitted from consideration at this point. There is, though, valid design rationale for using the WebCom Information system to manage module event metadata.

In a further JavaBean analogy, the WebCom Information Framework might be used in conjunction with BeanBox style containers. A BeanBox describes software that provides a visual environment in which to manipulate JavaBeans. This visual Bean manipulation is used in conjunction with object serialisation to preconstruct and deploy arrangements of JavaBeans which achieve particular software tasks. A similar scenario exists within the context of WebCom entities and essentially outlines a possible Condensed Graph IDE implementation.[4] In this scenario, the enduser can create and mutate the data contents of node objects, set parameters, connect nodes, and perform other graph design activities. Upon completion of graph arrangement, the graph designer can serialised the designed graph to a useful representation. For instance, the designer might generate a representation in the common XML based Condensed Graphs description format and deploy this representation for execution or for further use by other tools in the WebCom software suite.

2.3 UML Outline

With this motivation and general implementation sketch in mind, this section will detail a draft Information Framework UML specification. It is anticipated that this specification will evolve to meet future criteria as mentioned in the above section.

§19 Misleading Nominature. *≪A warning and lesson regarding the illchosen* `NodeInfo` *terminology≫*
Before proceeding with a description of the software elements of the framework, there is an issue of confusing terminology to address. The early versions of this framework were developed with a static node operator viewpoint. When the fluidity of operator port contents is ignored, Condensed Graphs appear as a very node-centric model. In reality, however, the key component of the system is actually the operator. This is very much the case in regards to documentation notations, and is intuitive when it is considered that the only significantly varying elements of the model are the operators. Node-centricity fails in any interpretation that stresses the variable elements of the Condensed Graph model.[5]

Unfortunately, the framework was initially developed in the context of static operators, and so the node object has undue priority in nominature over the operator object. This is most apparent in the name `NodeInfo`, referring to a principal meta information class. In fact, this object really describes the metadata of the operator on the operator port of the node in question. In the static operator case, this coincides with the node itself. But although this works fine within the static operator case, it fails seriously in the dynamic operator case.

It has proven difficult to change this unfortunate terminology with the benefit of hindsight. So, although the existing terminology will be used herein, more appropriate nominature will also be highlighted.

§20 Core Information Framework UML. *≪UML outline of the core classes in the Information Framework≫*
The central spine of the Information Framework is the `Info`, `NodeInfo` and `CondensedGraphInfo` inheritance hierarchy, diagrammed in Figure 10. A top-down description of these classes is the most illuminating.

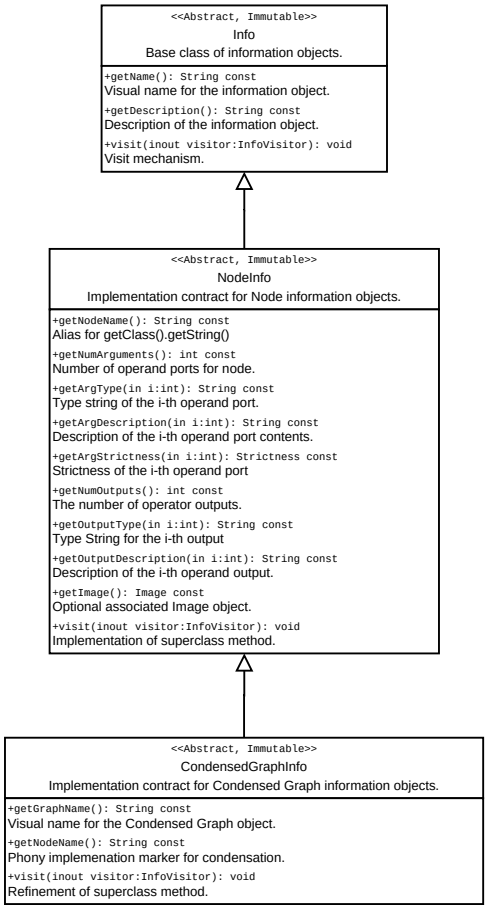

Figure 10: UML Diagram of Core Information Framework Classes.

The `Info` class is the abstract base class of the structure, its purpose to act as the hierarchy root and to provide the basic common interface expected of all `Info` classes. Contentwise, it is a trivial data class comprising two immutable metadata fields, namely a name and description of the object under consideration. These fields are intended to contain display friendly cues for endusers, the name field being a short reference tag for the entity, the description field a more elaborate tooltip type text.

At present, when endusers encounter graph elements in the current WebCom toolset, the tags used to report contents and properties are not immediately helpful. The name and description fields in `Info` classes can be used to provide more quickly recognisable cues. Although, these entries are of use in error reporting, this is not presumed to be the typical or only application.

The existence of a hierarchy base means functionality can also be required of all `Info` objects, not just attributes. In particular, support for the Visitor pattern is mandated by the `visit` method. This is the hook method for return dispatch in the Visitor pattern implementation and is described in more detail below.

There are only two `Info` class extensions of importance. The first is the `NodeInfo` class, used to describe metadata particular for an inplace operator. Note that `OperatorInfo` would be more proper terminology for this class. The second class is `CondensedGraphInfo`, used to describe metadata regarding a Condensed Graph object. It is more accurate to say, however, that this `Info` is used to contain metadata for the condensed operator that manages the condensation and evaporation process of the actual graph. That it is a peer for condensed operator metadata, clarifies most precisely why it is a subclass of the `NodeInfo` class which models operator metadata.[6]

Considering `NodeInfo` in more detail first, it is an abstract, immutable and primarily data oriented class, similar to the `Info` class in this regard. It is essentially not much more than a number of additional fields augmenting the `Info` class together with a relabelling. The additional metadata attributes are:

- A node name, the internal name for the operator, defaulting to a classname string.
- A field containing the number of operands. This is both the number of operand ports on a node containing the operator and the arity of that operator.[7]
- An indexed attribute of operand typing data. Each operand has an associated type string describing permitted datatypes. The interpretation of this string will be discussed in the chapter on type checking.
- An indexed attribute of operand user friendly descriptions.
- The number of operator outputs[8].
- An indexed attribute of output type strings, one per output.
- An indexed attribute of output descriptions.
- An optional graphical image for potential user visualisation. For the Condensed Graph Infos this might be an actual diagram of the expanded graph, for instance. This is blank by default.

Finally, the `CondensedGraphInfo`, which really refers to condensed operators, is a small further extension of `NodeInfo` with the addition of an attribute string graph name field for friendly graph references. It also includes some node name field masking.

Both `NodeInfo` and `CondensedGraphInfo` extend the `visit` method to correctly dispatch callback.

§21 Example Info. *≪Basic application of the Info classes and a trivial `NodeInfo` example.≫*

The use of Information Framework classes by third party applications is intended to be by the implementation of concrete `Info` objects, describing operators and graphs. The main current implementations involve static descriptions of operator data. An extremely typical such `NodeInfo` example is included in Appendix A. The programing is simple, the only interesting portions being the string constants. If anything, this example illustrates how these classes are trivially amenable to automatic generation schemes.

The included example is a `NodeInfo` object for a basic addition operator. The attribute assignments are what distinguishes this `NodeInfo` from any other current example. The name assigned is "Addition Node", the node name is "`webcom.nodes.core.AdditionNode`", the contents of the operand type strings, both identical, are "OR(`java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`)". The interpretation of these strings is covered in more detail later, but this example indicates the conjunction of the primitive numeric Java types. This particular operator or static node has a single output, the result of the summation of the two input operands permitted.

Implementing these `Info` interfaces need not involve static descriptions of operators and graphs, however. It is possible to produce implementations backed by nonstatic data. For example, all available operators and metadata items may be stored in a database and a concrete realization of the `NodeInfo` interface programmed to acquire data dynamically from this database on request.

§22 Visitor and Factory UML. *≪Description of Visitor and Factory patterns in the Information Framework.≫*

The Visitor and Factory patterns in the Information Framework complement the above described base data classes from the perspectives of flexible application and construction respectively. Figure 11 illustrates the UML outlines for the relevant classes.

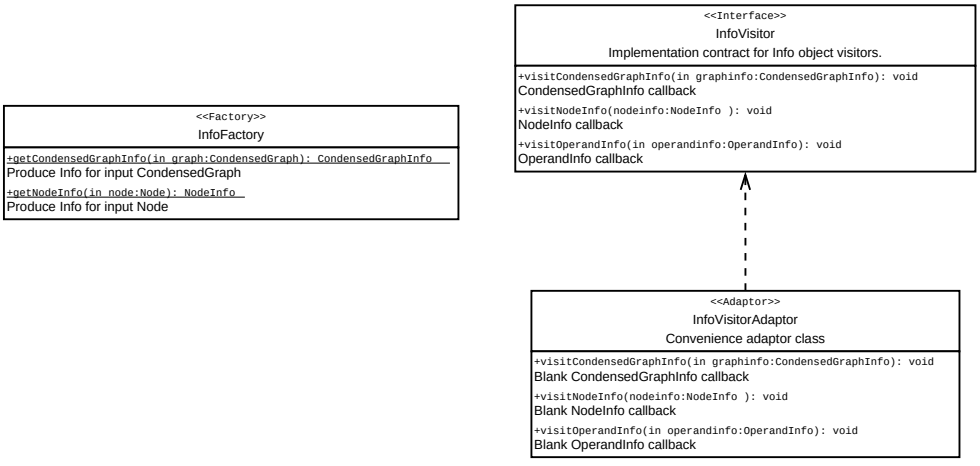

Figure 11: UML Diagram of Visitor and Factory Information Framework Classes.

The Information Framework Visitor pattern support, via the mechanism of double dispatch hooks, has already been encountered. The `InfoVisitor` interface represents the other half of the dispatch pair and forms the contract for visiting objects to implement in order to automatically traverse the visit tree via builtin mechanisms. So, an application programmer wishing to process one or multiple `Info` items, may specialise the `InfoVisitor` interface or its convenience adaptor subclass to the desired end, and then invoke the `visit` methods of the `Info` items to process, handing them the customised `InfoVisitor` instance.

Centralising metadata processing code in a single visitor class, or in a tightly collaborating facet or Mediator, promotes good design in the applications that exploit this Visitor pattern. This can encourage sound design in the internal WebCom programming base.

The next section includes example applications of the Information Framework. Of special note is the documentation generation example, as it illustrates the canonical use of the Visitor pattern. The idea there is to dump the available metadata in a typesettable form, very much like a dumb JavaDoc Doclet. The point of interest is in the simple design of this application given the Visitor support.

The Factory pattern is likewise elementary, but similarly valuable. At present, there is a basic Factory implementation in the `InfoFactory` class to support flexible construction. This interface can be presented with a Condensed Graph or node at runtime and will return a rudimentary `Info` object for that element. The present version makes a best guess at metaattribute contents using some internal WebCom reflection capabilities. It is of particular design value to present a simpler interface to these internal functions, as they are currently too low level for easy third party application.

In future, it may be useful to extend the `InfoFactory` class to support both persistent metadata and dynamically backed metadata. So, the results of each query would be persistently stored and augmented with other available information as it becomes available. One way of actually implementing this scheme is to use a database maintenance module, and have it listen for WebCom events that may provide additional attribute information. The discussion in the next chapter on event systems will clarify how this might operate. Also, informing methods can be provided for third parties to educate the system about particular `Info` objects if this is convenient. And finally, this module can also interact with other information management modules and exchange mutually interesting data.

2.4 (Potential) WebCom Applications of the Information Framework

The remainder of the chapter concerns some further motivation for the Information Framework, in the form of potential and real applications thereof. Described below are a number of applications that highlight the flexible nature of the system. Despite its lightweight design, the system plays a keystone role in higher level WebCom plumbing software.

2.4.1 Documentation Provision

The first application involves the delivery of documentation aides and cues to developers of Condensed Graph programs. Although, the Information Framework by its very nature can be exploited in many documentary opportunities, the focus here will be on two particular instances. Since the framework disseminates valuable information, these two examples fall short of exhausting the possibilities.

§23 IDE Runtime documentation provision. *«Applications of the Framework within existing WebCom tools»*
The current WebCom software development suite of tools is centred on the Condensed Graph IDE. This tool facilities the production of diagrammatic and XML Condensed Graphs representations. It also forms a testbed harness for the execution of these Condensed Graphs. Note as an aside, that there are a range of execution harnesses and the IDE forms just one option. The topic of an unified execution API is covered in the Submission Framework application below.

Operationally, the IDE presents the user with a graph construction canvas upon which nodes with static operators may be arranged and interconnected by means of pointer manipulations. Available operator/node elements are presented in a palette for drag and drop selection. The IDE is an essential tool for experienced graph developers, but is also a valuable means of introducing new users to the graph paradigm which can otherwise be difficult to grasp. Many beginner difficulties are often resolved in the first IDE session.s are often resolved in the first IDE session.

A deficiency in the IDE, and a current focus of development activity, is a lack of relevant documentation cues in the software. Rather than help manual support, the Condensed Graph development philosophy depends on the provision of pertinent details on the construction panel. Searching a help system is an awkward solution and would impede graph design activity. The graph designer should be presented with sufficiently detailed operator data immediately to hand, in order to facilitate design selections in constructing new graphs.

The Information Framework helps with this difficulty in two ways. The provision of easily consulted printed documentation is discussed presently, but first runtime program help cues are examined.

Essentially, the Information Framework can be used to provide content in a tooltip style approach within the IDE software application. Potentially useful and already used `Info` objects may be cached in the IDE[9] and recalled to provide documentation details to designers. New `Info`s can be added to the cache as their underlying peer entities are referenced in IDE graph constructions. This way, the IDE can present a book of `Info` objects that is comprehensive in regard to the graph under construction.

Exactly how these Info objects are used to prompt graph designers is a careful user interface problem. This usability is critical but does not practically affect the mechanism of delivering documentation via the Information Framework. Cues may be presented as automatic, hidden, selectable, or otherwise.

The Information Framework is ideally suited to use as a documentation content system within the IDE. In addition to UI improvements, the adoption of the Information Framework for content management also has a the wider impact of expanding the body of available `Info` classes. Mandating the documentation of

operators via `Info` objects prior to acceptance in the IDE would mean that in addition to IDE documentation, these `Info` objects would be available to other tools in the WebCom application suite.

The documentation provided by the Info system consists of more than just description attribute fields. If anything, experience has shown that the type string attributes could possibly be the more valuable documentation. Often an operator's name is sufficient to describe its function, but will not provide enough information regarding operand types. The Information Framework provides a mechanism for the description of such available documentation and also for the future additional of relevant data attributes.

§24 Printed documentation generation. *«An example exploitation of the Visitor pattern.»*
Printed documentation notes on available operators serve as a useful desk reference during the graph designing process. Furthermore, the automatic generation of such documentation from available `Info` objects also demonstrates the application of the Visitor pattern in the Information Framework.

Name	Addition Node
Node Name	webcom.nodes.core.AdditionNode
Description	Add the operands.
Num Operands	2
Num Outputs	1
Operand 0	A summand
Type	OR(java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double)
Strictness	STRICT
Operand 1	A summand
Type	OR(java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double)
Strictness	STRICT
Output 0	Result of Addition
Type	OR(java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double)

Figure 12: Addition Node Documentation

The basic operation of the documentation generator is to tabulate the attribute entries of the available operators. An example of produced results can be see in Figure 12. The generator is implemented as an `InfoVisitor` extended to generate a LaTeX listing of each `NodeInfo` or `CondensedGraphInfo` it is applied to. There is similarity between the documentation generator walking over available Info objects and the operation of a Java Doclet descending available Java packages for source code. The production of LaTeX source is simply a typesetting convenience. The tool could easily be modified to also produce hypertext or an XML formatted list of the Info documentation.

2.4.2 Type Checking

The original purpose of the Information Framework was to provide data useful in the graph type checking process. That is, provide data content to establish the compatibility of operators occupying nodes. Although the particulars of this type checking will be outlined later, they serve an illustration value in demonstrating the Information Framework and as such, it is of use to trail the type checking description here.

The type checking data is carried in the type string fields of `NodeInfo` and `CondensedGraphInfo` objects and the compatibility of operators is determined by matching outputs to input type strings either before or during a graph execution. The output type string of an operator needs to be contained within the input type strings of the operators to which the initial operator result is propagated. Type strings contain a set based notation for permitted type strings, so output types are contained within input types in set containment terms.

The process of type checking Condensed Graph DAGs involves some amount of processing and symbol table type activity. The goal of the chapters leading up to a full description of the type checking system is to outline software and design to facilitate the implementation of type checking within a general WebCom mechanism. In particular, the module and event API supports need to be described before the type checking implementation harness can be properly considered.

There are two aspects to the type checking operation. It can be run statically during the endphase of the design or it can run dynamically at runtime. Although, the latter approach involves a computational overhead, it may be appropriate for looser typed graphs. Another interesting consideration of dynamic type checking in Condensed Graphs is in the case of dynamic operators. Specifically, the generation of operators via Condensed Graphs and their latter application within the graph that constructed them, rather than dynamic operators as in the condensation and evapouration process. Either way, the proposed system is designed to be used either statically as a design aid or dynamical for flexibility.

The more indepth examination of the type checking problem is postponed until Chapter 6.

2.4.3 Submission Framework

The final motivating example here is the WebCom Job Submission Framework API. This API uniformises programmatic WebCom task submission mechanisms and presents a single API for WebCom invocations.

Presently, there are numerous methods for submitting work to WebCom, including direct WebCom commandline invocations, constructing WebCom instances from within Java, Web Services invocations, and via the IDE interface. In particular, the Web Services and commandline invocation methods have programmatic Java counterparts. The current commandline tools simply invoke Java applications to perform the submissions. These and future invocation schemes are handled uniformly by the Submission Framework.

A uniform API approach to WebCom job submission allows different WebCom backend scenarios to be varied independently of the invoking technology. This is of particular value to application developers who wish to use WebCom as an implementation technology. These developers can program job submissions to this new API and leave the choice of actual backing setup to the local site.

From this point of view, the Submission Framework supports the use of WebCom from within other applications. Specifically, these applications themselves need not necessarily be programmed as Condensed Graphs. So, a traditional programmer might implement the majority of an application in Java, for instance, but leverage WebCom to handle portions of the program that highly suited to Condensed Graph solution.

§25 J2ME Origins. *«Submission Framework origins in WebCom mobile phone software»*
The Submission Framework was developed as a result of demands arising from J2ME WebCom applications. J2ME is the embedded systems Java edition supported on most recent mobile phones, handheld computers and increasingly on other low end consumer electronics.

The J2ME WebCom application was designed to enable users to submit jobs to an Internet connected WebCom server. These jobs would be submitted by mobile devices, over WAP or GPRS networks, to machines which would perform the specified WebCom tasks and return results to the original clients.

This connectivity makes possible a variety of novel WebCom applications. For instance, a scientist might preprogram Grid submissions,[10] so that she might run large computational trials remotely while onsite. Similarly, mobile inventory could be conducted by mobile devices connecting to a server database. Business tasks could be scheduled and managed via WebCom, etc.

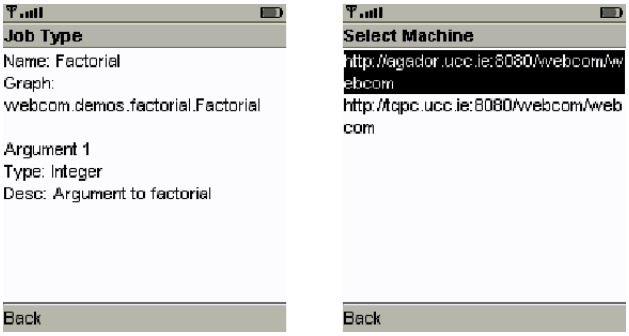

Figure 13: J2ME Application.

Figures 13 and 14 contain screenshots from the prototype J2ME WebCom application. The left side of Figure 13 illustrates a preprogrammed Factorial computation task. Users may either select preprogrammed tasks or construct customised job specifications. The right side of Figure 13 shows the machine selection screen where users target operations to server machines. Users may also augment this target machine list.

The left of Figure 14 shows execution parameters for a Factorial task being entered. To submit a task, users select a target machine and enter required parameters as per the job specification description for the desired task. This job is then submitted via, in this case, a WAP connection[11] to the target machine. This machine performs the required WebCom computation locally and returns result data to the original client. The right of Figure 14 shows this information being displayed.

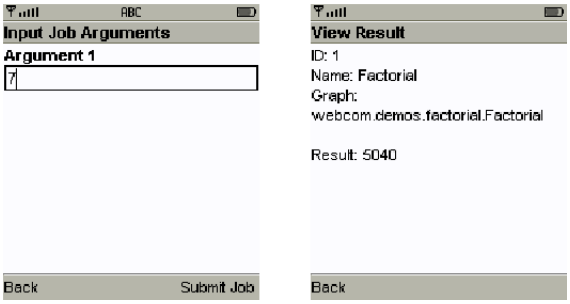

Figure 14: J2ME Application Continued.

Being a simple prototype the initial J2ME application does not address key security concerns. In particular, a security mechanism supported by a J2ME security toolkit[12] on the client side and by a full secure HTTP server architecture is essential. This security would also need to respect the internal WebCom security manager semantics. Some intermediate security measures might include restricting permitted machine to job pairings on both ends of the transaction, and requiring strong authentication by client machines.

The current application is limited to request type actions. In future, it is hoped that a fully connected WebCom instance will be developed for the J2ME architecture. This would permit J2ME devices to be leveraged into the networked WebCom topology. In particular, J2ME devices could be scheduled specific work that requires the attention of the device owner. In this way, workflow and scheduling aspects of WebCom can be exploited to benefit the device owner, rather than in exploiting the limited machine cycles available on a J2ME device for group computation.

The Submission Framework arise from the need to describe task descriptions both in the J2ME client and in the WebCom Java servlet on the server end. This WebCom servlet can receive WebCom requests from non-J2ME sources also, and might be used, for instance, as a web interface to Grid WebCom.

This servlet also raises the issue of uniformising programmatic access to WebCom invocation methods. Application programmers might benefit from a single WebCom invocation API, and a variety of backend WebCom invocation implementation schemes. So, for example, a program that embeds a direct WebCom instance could be easily configured to depend on a servlet WebCom instead, without recompilation.

§26 Framework Organisation. *«Sketch UML for Submission Framework and outline»*
The organisation of the Submission Framework inherits the Information Framework classes, and adds submission software for complete job tasks. A basic submission job representation is a `CondensedGraphInfo` together with representations for graph operands and some management details.

Figure 15 depicts the representation for passing graph operand job parameters. Here, the `OperandValue` class maintains a string representation of the desired operand and a mapping to the `OperandInfo` containing the operand typing metadata. This class inherits from `Info` and participates in the Visitor pattern. A more

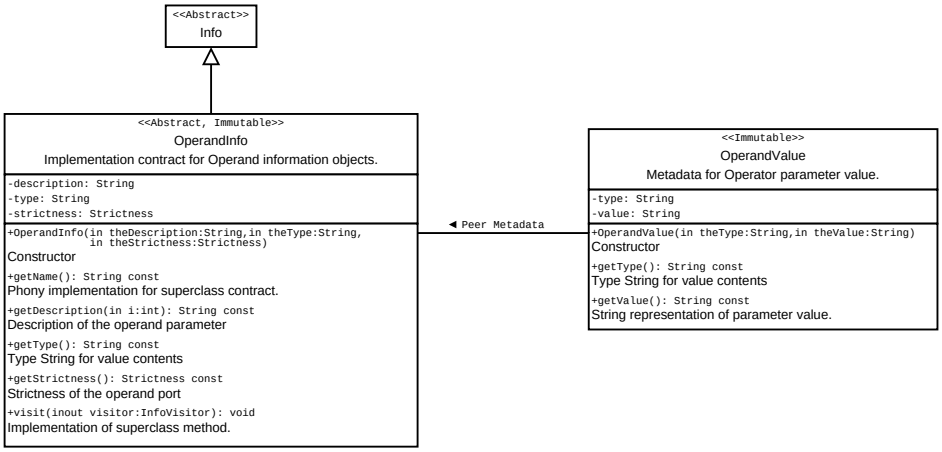

Figure 15: UML Diagram of Submission Info Classes.

correct model would manage the operand data in `CondensedGraphInfo` using these `OperandInfo` objects. This would allow the coupling of `OperandValue` objects to their peer `OperandInfo`s and to the containing `CondensedGraphInfo`. This would represent a more accurate view of the software relationships and tidy some of the multiattributes of the `CondensedGraphInfo` class.

The role of `OperandValue` objects is to contain input graph operand values for the job graph to be executed. Figure 16 illustrates the remaining job management classes. Here, `Job` is the final element of the job description triple, containing transmission and result storage details. In particular, it stores the request time, the result, and a universally unique[13] job identifier, in additional to references to the `CondensedGraphInfo` and associated `OperandValue` objects.

The submission objects form a matching hierarchy for the submitter classes, in Figure fig:umlsubmitter below. The parent `JobSubmission` contains the job submission data elements, namely its liveness and a list of interested objects. A submission is different from a job in that job descriptions may be reused, particularly in the scenario where a job is submitted to multiple WebCom backends which compete to turnaround the result. For instance, the same job may be submitted to a local WebCom cluster and perhaps to a larger national Grid for simultaneous execution. In this case, the idea is to try for quick turnaround on the cheap local cluster resource whilst the job is queueing on the more expensive remote Grid.

The present framework includes two concrete `JobSubmission` class implementations, each requiring custom submission code. These are `BasicWebComJobSubmission` for local direct WebCom invocations, and `ServletWebComJobSubmission`, containing a protocol servlet address field, for servlet WebCom operation. Both realisations have peers in the submitter hierarchy in Figure 18.

Submission objects arrange the transmission of data to and from WebCom backends, and so warrant asynchronous thread implementation. These threads are managed by submitter elements, introduced presently.

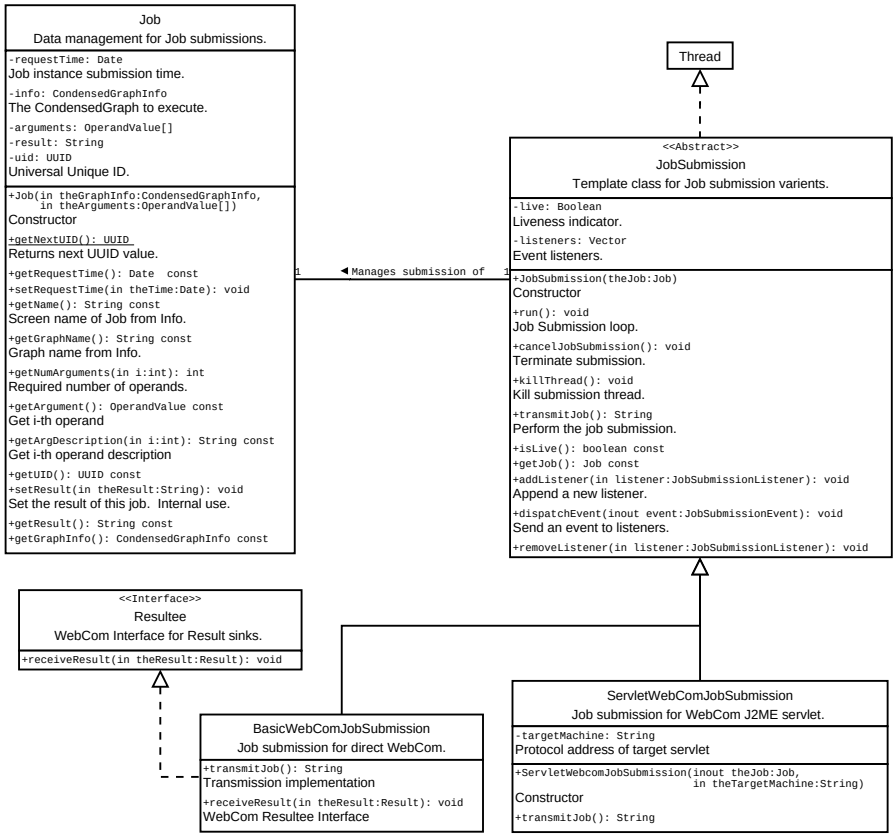

Figure 16: UML Diagram of Job Submission Classes.

The `JobSubmission` threads raise events at the start, completion and cancellation of submissions under event model described by Figure 17. This is a traditional event model where `JobSubmission` forms the event source and implementations of `JobSubmissionListener` interface sink the `JobSubmissionEvent` events. Listener registration is done by the `JobSubmission` class.

The main useful application of the event structure, other than by third party code interested in job submission progression, is by the submitter classes shown in Figure 18. The `JobSubmitter` class provides a hook for third parties to invoke submissions and to process the results obtained. Once a `JobSubmission` object has been acquired, it can be actively submitted via the `submitJob` method of the relevant `JobSubmitter` object. This `JobSubmitter` starts the job and provides callback adaptor hooks for the start, finish and cancel events. In this sense, `JobSubmitter` is simply an adaptor for the `JobSubmissionListener` interface, albeit one with useful methods supporting third party participation in the submission process.

`BasicJobSubmitter` is a trivial concrete `JobSubmitter` implementation. This class forwards basic text representations of the submitted job, output and cancellation details, to the `JobSubmitter`'s `Console` object.

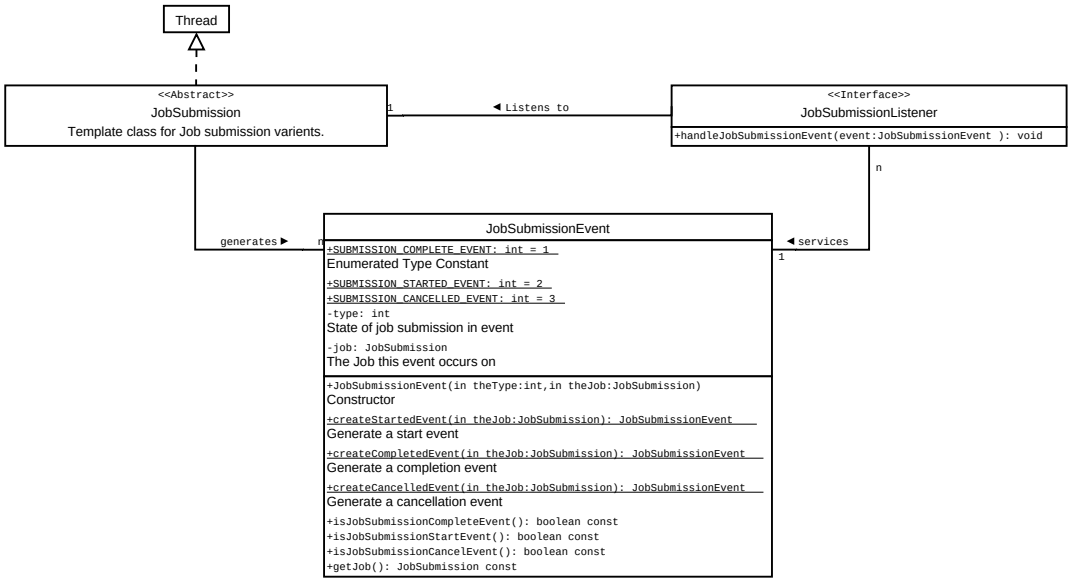

Figure 17: UML Diagram of Job Submission Events Model.

A trivial commandline job submission tool would simply then dump the contents of this `Console`.

It helps to reconsider at this point, how a WebCom backend might be added to an existing application. Suppose an application designer has isolated program elements which have an elegant graph representation. The designer constructs suitable graphs and prepares job specifications for them in the form of `CondensedGraphInfo` objects. Thereafter, a `JobSubmission` object is created by constructing the required `Job` object and coupling the `CondensedGraphInfo` object to the instance operands. The designer finally selects the means of execution by instantiating a `JobSubmission`. In practice, the selection of concrete implementation is done with a factory class, leaving the designer's application WebCom backend independent.

For the designer's purpose, it may be necessary to implement a custom `JobSubmitter` to format the result strings into exactly the desired form for reimportation into the original application. To invoke WebCom the designer simply constructs the desired instance of `JobSubmitter` and uses this to fire the execution of a concrete `JobSubmission` via the particular means described therein. This will cause the desired graph to be executed and the designer can recover the results from the custom `JobSubmitter` extension.

§27 LaunchGUI. *«Graphical tool for the submission of Condensed Graphs»*
LaunchGUI is a graphical application for the selection and submission of Condensed Graphs to WebCom backends, implemented with the approached outlined just above. It applies the Submission Framework as proposed and provides basic reporting to populate the display of a graphical console.

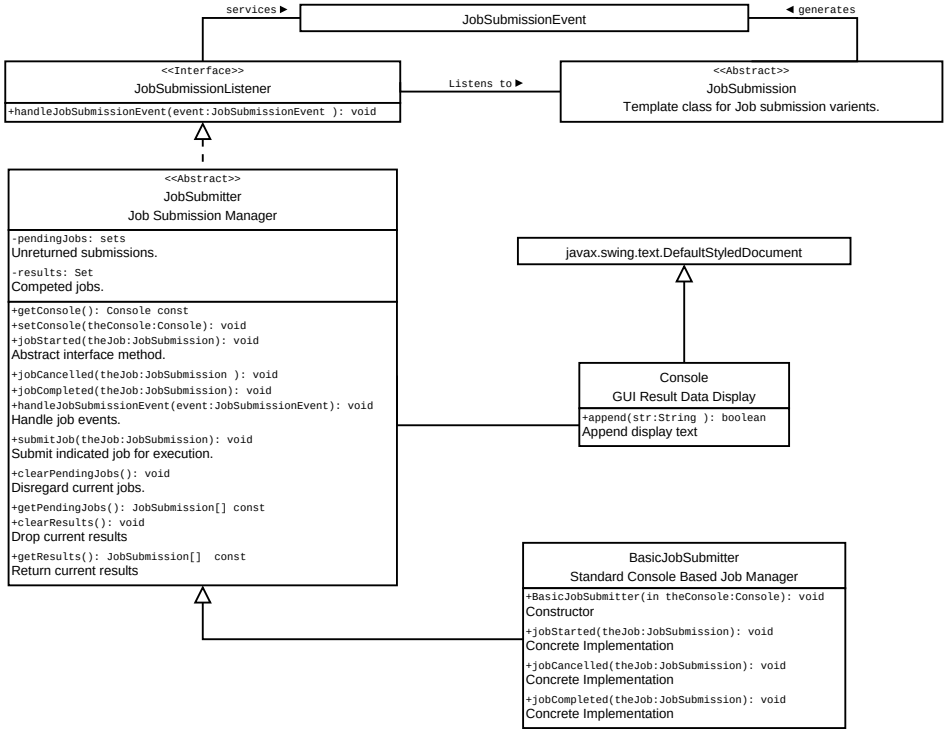

Figure 18: UML Diagram of Job Submitter.

The LaunchGUI tool provides convenient graphical WebCom invocation and is particularly suited to new WebCom users. Jar libraries may be loaded and unloaded to dynamically change the available task descriptions and backing classes. Once desired libraries are loaded, the user are shown the display in Figure 19. The top half of the window diagrams the available task descriptions, automatically extracted from the available libraries by identifying and instantiating `CondensedGraphInfo` classes. The bottom half of the window displays the information provided in the selected `CondensedGraphInfo`. In this case, the `Factorial` condensed graph object is selected and relevant details are displayed from the `FactorialGraphInfo` class.

Once a desired job description is selected, the user may advance the submission process by pressing the "Set Arguments and Execute" button. Doing so presents the display in Figure 20. Here, required operands can be configured. The displayed descriptions and typing data are taken from the relevant `CondensedGraphInfo`. The user may customise the operands by modifying the "Value" fields. When the user has finished setting parameters, the actual task submission may be accessed via the "Execute" tab.

Selecting the "Execute" tab yields the view in Figure 21 where available submission procedures may be selected via the "Submission Type" widget. Like job descriptions, these submission methods are also automatically read from the available libraries by identifying concrete instances of `JobSubmitter`.

Upon submission method selection, a configuration panel is displayed in the available space. In Figure 21,

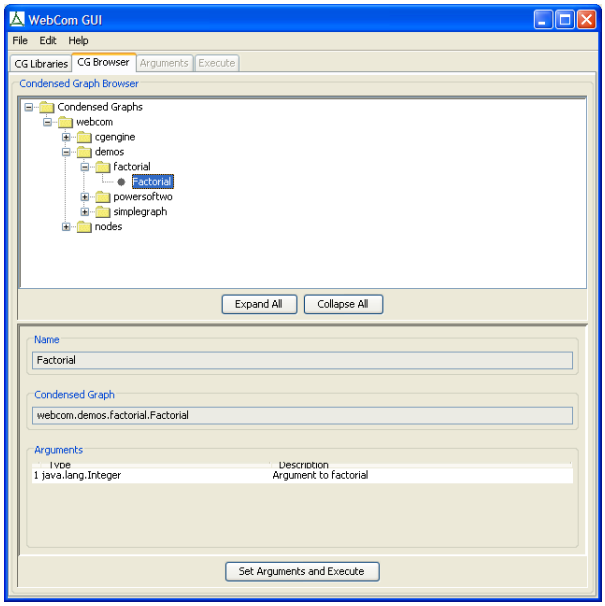

Figure 19: LaunchGUI `CondensedGraphInfo` Selection.

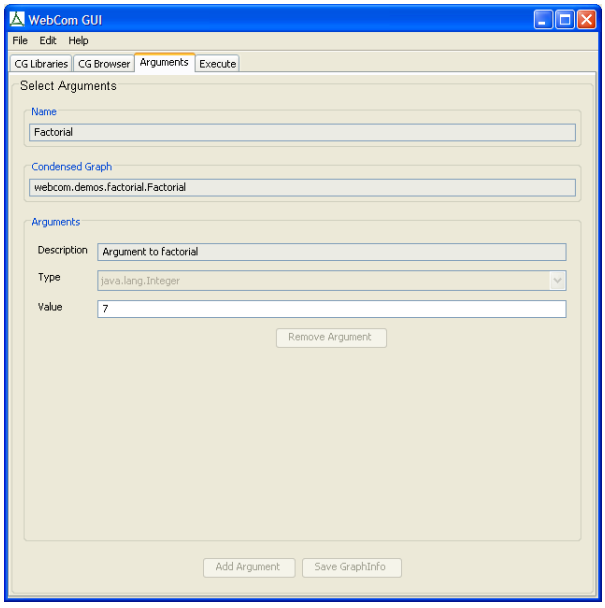

Figure 20: LaunchGUI Operand Configuration.

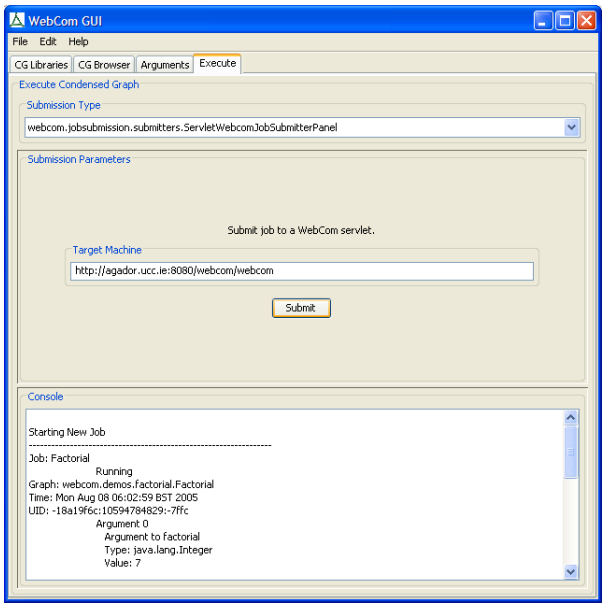

Figure 21: LaunchGUI Submission Dialogue.

a servlet submission is selected and the user is being queried for the servlet protocol address. When the submission process has been configured, the user may perform the submission by selecting "Submit.".

Upon submission, the requested `JobSubmitter` is constructed and the associated configuration panel displayed. The `JobSubmitter` knows which version of `JobSubmission` is required, meaning that concrete `JobSubmission` versions can be constructed blind to the LaunchGUI tool by the pluggable `JobSubmitter`. With the `JobSubmitter` and `JobSubmission` in place, there is remaining work. The submitter is informed of the correct `Console`, namely the text box at the bottom of Figure 21, and then invoked. In the case of the LaunchGUI there is nothing further to do and the output will appear in the `Console` text box display. More sophisticated applications might augment the `Console` or `JobSubmitter` to recover specific result data.

Aside from illustrating the user application of the Submission Framework, this tool has a role in the WebCom application suite as the present WebCom tools lack a friendly graphical tool solely for the submission of WebCom tasks. Previously, either a commandline tool was used or the task was first loaded into the IDE application. The former is not suitable for new or nontechnical users, and the latter is too detailed for simple WebCom trials. In any case, the IDE is part of the software development kit that is not required in a runtime environment. The LaunchGUI tool is more than sufficient for enduser invocations.

The LaunchGUI and IDE may be combine in future to uniformise submission mechanisms. While the IDE does include a separate task invoker, this employs neither the Submission nor Information Frameworks.

Also, the LaunchGUI tool may be written in the notation of the Module API, that will be discussed in Chapter 4, and incorporated into the system tray toolset described there. This is an always on version of

WebCom with potential to be leveraged as a system level application, and is the focus of more in-depth discussion later. The tray tool makes WebCom available to all applications and would be nicely complemented by a LaunchGUI feature. Such an addon is easily implemented with the Module API.

But before considering the Module API, it is first necessary to examine the aspect based eventing system that has been added to WebCom. This is the topic of the next chapter.

Chapter Notes

[1]Metadata schema mutation requires some way of informing clients of the current schema. This support is not required for type checking and is perhaps not required for the vast majority of other applications also. As such, the implementation of metadata schema mutation is a low priority item. In terms of module data schemata, there are some concerns, relating to informing clients of module functionality, that are considered briefly in the chapter on modules and form some overlap with metadata schemata mutation.

[2]And likely containing certain passive data items neither required nor desired in the metadata schema.

[3]This WebCom module configuration and other module oriented developments will be considered in a later chapter. But briefly, the new module configuration supports the automatic loading of third party modules and is intended to provide opportunities for module writers to advertise functionality. Both these design goals benefit from the use of metadata.

[4]Note that a Condensed Graph IDE is already in use, following a different design. Nevertheless, a BeanBox approach to graph design might be of benefit, either in future versions of the existing IDE tool, or in a separate lightweight graph designer.

[5]This is not to say that nodes are not the fundamental structure in the Condensed Graph model. They are, but they are not central in the operational implementation to the same degree as they are in the specification. From the enduser view, it is operators that are central.

[6]The alternative view would be to say that Condensed Graphs are instances of Nodes. This is true but doesn't help explain why the metadata requirements of a Condensed Graph are similar to those of an operator.

[7]This is a small but precise distinction. The number of operand ports on a node will change if a different operator is placed on the operator port. The number of node operands can change dynamically to match the graph construction. Operator arities are fixed.

[8]Presently, this must be one as multiple output operators are not currently permitted.

[9]The actual management of `Info` objects is ideally suited to a module formalisation as noted earlier.

[10]The WebCom technology supports integration with Grid computing.

[11]Any network connection technology could be used for this connection, e.g., WAP, GPRS, Bluetooth, WiFi, etc.

[12]Such as that provided by the Bouncy Castle project.

[13]Via the UUID Internet specification.

3

Aspects and the Event API

A major development in the WebCom design over the course of this dissertation has been the adoption of Aspect Oriented Programming (AOP). Whilst WebCom remains predominantly OO in nature, significant elements of AOP design have been introduced, especially in the guise of the Event API. The direction of this chapter is to explain aspects and what aspect design brings to WebCom.

Particular effort is spent on the Event API, as this provides a great example of incorporating the benefits of aspects whilst retaining an illusion of OO implementation. Moreover, the Event API is a cornerstone of the revised module structure and a key component in the implementation of type checking support.

3.1 Introduction to Aspects

Aspect oriented programming is a paradigm which has recently acquired much publicity. Although, aspects have emerged as a recent phenomenon, they have roots in older Xerox PARC research projects. With the development of better support tools for AOP, and with more and more developers being introduced to aspect techniques, it is only a matter of time before aspects will be seen in most OO projects.

§28 Aspects and Object Oriented Design. *«A comparison and consideration of aspects and OO design philosophies»*
Aspect oriented programming does not seek to replace OO programming. In fact, aspects complement OO design, just as OO design complements imperative programming. Aspects address a difficult OO design

problem and implement a pattern to limit the undesirable consequences of this problem. What differentiates OO programming from imperative programming is design philosophy more than any actual implementation programming language. This holds true for aspect oriented programming also. Although aspect design occurs intrinsically coupled with OO design, it requires a radical change in view.

OO design is not limited to imperative base languages and OO principles can be applied in a range of different settings, logic programming, functional programming, etc. And although this argument may be made of AOP also, it is somewhat true that aspect oriented design is counterproductive outside of imperative and OO contexts. The problem with using aspects within declarative programming is that, in some senses, aspects are a disguised goto statement. Aspects do provide an ability to abuse program control flow and, as such, are at odds with a declarative approach. However, aspects involve a highly structured approach and so can be forgiven most of the goto statement comparison. In truth, aspects are really less questionable than OO stables like exception handling support.

Aspects provide the facility to augment existing programming code at particular definable points. Developers can use this to indicate generic code to perform at each instance of a particular circumstance within a program, and the mechanism whereby this is achieved is the single main advance provided by AOP. With disciplined programming and a framework for generating events, an AOP style of programming could be implemented without the use of aspect constructs. By implementing this "little eventing policy" pattern, aspects contribute enormously to the successful negotiation of crosscutting concerns in program design.

The aspect approach is a valuable development in OO design. In fact, AOP and OOP are very critically interdependent. OOP depends on aspects to solve a difficult recurring design problem, whereas aspects depend on OOP for their proper context.

§29 Aspects within WebCom. *≪The adoption of AOP possibilities within the WebCom software suite.≫*
Support for the AOP paradigm has been introduced to the WebCom suite as part of this dissertation. Introduced AOP even makes possible an event based programming methodology within WebCom. Later, it will be seen that logic programming methodologies are also available, though not AOP dependent. These programming techniques are transparent to developers who do not explicitly wish to apply them.

The introduction of aspects to WebCom was done using the AspectJ system, a Java AOP implementation. AspectJ adds additional notation to the Java specification and compiles aspect source code into binary compatible Java bytecode.[1] The AspectJ compiler also compiles Java code that does not exploit aspects, so a change of compiler is the only change that developers ignorant of aspects notice. In the context of automated WebCom builds, this change is difficult to spot. This just illustrates how easy it is to adopt aspect support within an existing project, and how this adoption can be done without any consequences in the existing code.

Aspect documentation herein will be presented with AspectJ notation and terminology. The basic concepts are common to AOP implementations, and casting them in AspectJ terms should not cause difficulty.

It is important to point at the results of changing to the AspectJ compiler. Essentially, without noticeable cost, the WebCom system can provide the Event API abstraction described below. This is a mechanism for

providing core WebCom event data without requiring modification to these internals. The Event API also provides a model for implementing event systems within the WebCom core or in module development.

The basic facility to program using aspects in the WebCom suite is also important. The possibility to apply aspect design solutions within WebCom tools is a valuable aide. With careful application during refactoring and extension, AOP provides an opportunity to further tidy the WebCom design.

3.1.1 Crosscutting Concerns

In short, aspects handle crosscutting concerns, providing a formalism, namely that of the aspect, in which crosscutting concepts may be modularised and reused. The introduction of this aspect device promotes good design in addressing crosscutting concerns, but can also be used as an effective program implementation device. Most importantly, aspects introduce additional OO design possibilities and augment basic OO in a manner akin to that of patterns. Aspects may be considered as a very important pattern, but ignoring the software developed around aspects and describing AOP as the application of a single pattern is inaccurate.

The problem AOP addresses is crosscutting concerns, elements of an OO design which are scattered over a number of design units. A crosscutting concern is a design element that affects or is implemented by a number of classes. From a design perspective, this is highly undesirable in that it promotes high coupling and low reusability. Well designed OO applications try to minimise such coupling.

Crosscutting is a very serious problem and inhibits reusability aspirations of OO design more than perhaps any other design problem. Often, the particular crosscutting problem is inherently multiclass and does not present a noncrosscutting implementation. Such unavoidable crosscutting is difficult to implement, requiring consistent ad hoc programming in a range of different classes. Moreover, the implementation of schemes to manage crosscutting tend to be oneshot, if even attempted. Crosscutting is often just tolerated. But in either case, the reusability of the affected components is torpedoed. Worst still, crosscutting concerns are encountered time and again by OO designers, and occupy disproportionate amounts of programming effort.

Some motivating examples may help clarify this discussion and, unfortunately, there are more than enough classic crosscutting examples to choose from. For instance, consider the problem of adding security checks to an existing system.[2] Credentials need to be verified at points where elevated privileges are required. However, these points are often scattered throughout the software implementation, since it is improbable that privilege elevation points might have been centralised during the original implementation. And even if such foresight was employed, it is unlikely that all privilege elevation points were successfully and consistently captured. Essentially, the adoption of a security mechanism using traditional design would require a demanding refactoring effort touching on all elements affected in the security crosscutting concern. Such time intensive reimplementation is a defining crosscutting concern characteristic.

Highlighting the crosscutting problem is not sufficient, AOP must also provide an effective solution. This solution is to aggregate definitions of the points where crosscutting intersect base code, and to provide programming code, or advice, to be performed at these points. This aggregation is the aspect element.

In the case of the security crosscutting concern, AOP provides a design solution by allowing all the privilege elevation points to be conveniently referenced within a security aspect. Each time a privilege elevation points is encountered, the aspect system provides an opportunity for the security aspect to execute some code. In this case, code to verify the user is authorised to elevate privilege.

This is a solution because it centralises the authorisation code, dislocating it from the privilege elevation points in the code base. Introducing this code at every privilege elevation point is programmer intensive, whereas isolating the points and describing the authorisation check once is far more productive. Furthermore, the original code containing the privilege elevation points is ignorant of the security aspect and thus freed of the crosscutting concern problem as manifested in the security requirement. The original code may be reused with or without reference to security, which if required can be reapplied via the aspect.

Security is not the only domain that lends itself to good aspect expression. Other recognised application areas for AOP include caching, logging, profiling, and debugging. A caching design is described below.

AOP facilitates the avoidance of crosscutting by modularised design which would otherwise pollute the overall OO design. In this way, AOP improves reusability giving rise to better OO design. But AOP is not effective in solving the crosscutting problem entirely. A lot of crosscutting concerns present easy aspect solutions, but this is not the case for all such concerns. Some may require convoluted and counterproductive AOP solutions. In a sense, it is a question of design application. AOP offers a design tool, effective in a large number of cases and consequently it is certainly valuable. But AOP is not a magic bullet for OO design.

There are other disadvantages in the application of AOP concepts and methodologies. One of which, is a reluctance on the part of developers to adopt the new technology. This problem is faced by any new paradigm and is only overcome with time and good development tools. This mirrors the transition from imperative programming practices to OO practices, although the change is not so revolutionary in the case of AOP. Much of the problem stems from lack of experience with AOP, which manifests itself in sometimes poor initial AOP design efforts. This is a discouragement for developers new to aspects.

Perhaps the biggest problem with AOP is the nonlinearity of aspect programs and the consequent debugging difficulties. This is a serious problem and the subject of much development activity. The emergence of mature visualisation tools for AOP is anticipated to help alleviate this problem. However, in the meantime, the initial obstacles in pursuing AOP are sometimes difficult to overcome.

3.1.2 Join points, Pointcuts and Advice

Describing the applications and mechanisms of AOP within WebCom will require some technical discussion of the aspect model presented in AspectJ. The detail outlined in this section will be used to present two classic aspect application implementations before moving on to consider specific WebCom aspect usage. These two case studies present aspect software elements that have already been trialled in the WebCom system.

The first piece of aspect jargon here is “join point”. A join point is an identifiable point in a control flow execution, such as a method call, assignment, conditional statement, for-loop initialisation, thrown exception,

or so forth. Perhaps the best characterisation of an AOP system are the join points permitted, describing how the system tradeoffs detailed join point reference against the encouragement of efficient, practical design.

AspectJ supports join points for method(or constructor) invocation and execution,[3] instance variable access and mutation, exception catch blocks, together with various class and object initialisation points that are of lesser interest here. It helps to think of AspectJ providing join point granularity at about the level of method access. That is, submethod level detail is difficult or impossible to capture in AspectJ, and counterproductive to attempt. Instance variable access and mutation are an exception to this view, but such operations deserved dedicated methods anyway. The other discrepancy, that of exception catch blocks, is an application of sufficiently utility to warrant relaxing the method level view.

A pointcut is a set of join points. AOP systems typically provide programmers with wildcards, predicates and other means by which to flexibly select sets of join points to form pointcuts. AOP depends on convenient aggregation of join points into pointcuts, specifying of code to execute when control occupies these pointcuts.

AspectJ provides a range of support for pointcut specification, including the specification of join point elements in Java dotted notation with unrestricted, letter, subclass and interface wildcards. AspectJ allows Boolean operators to be used on pointcuts to conjoin, disjoin, or negate sets of join points. Join points may be referenced by sort, e.g., instance variable access and mutation, method and constructor invocation and execution, and the various initialisation sorts. Pointcuts can restrict join points based on method arguments or target objects, based on position in a call tree or on syntactic class scopes, or based on user defined Boolean criteria referencing any available Java methods or variables.

Precision in pointcut specification is the black art of AOP. It is essential to describe pointcuts with sufficient detail to discriminate the join points of interest, yet robustness is equally desirable and extremely detailed pointcuts risk being made redundant on small base code changes. For instance, a pointcut explicitly referring to a specific class method may fail if that method is promoted into a superclass.

Advice is the term used for code executed at pointcuts. Particular advice is associated with a given pointcut, and is executed based on the nature of this association whenever the conditions of the pointcut are satisfied. Advice can be arranged at pointcuts in a number of ways. It may be executed before the pointcut is realised, after it is finished, or if it finishes with a particular result or error. Advice can also be specified around a pointcut, including code to execute before and after the pointcut. Advice may even refuse a pointcut execution, provided that any necessary return types are supplemented.

Additionally, advice may be used to inject new class and instance members into an object, so an object's interface can change dynamically to fit circumstances during an execution. For instance, if a certain pointcut demands a particular interface from an object of interest, the advice may augment that object with such an interface implementation if one is not already possessed. In this sense, advice can "decorate" existing code and permits a sneaky, encapsulation violating implementation of the Decorator pattern itself.

Note that pointcuts and their constituent join points are the only place where advice can be used to insert or modify behaviour. This explains why the exposed pointcuts determine the character of an AOP system.

Provision of advice at unsupported join points require abusing supported join points to coincide with the desired join point, and is deeply counterproductive.

An advice-pointcut pairing will typically address a single facet of a wider problem. Groupings of advice and pointcuts forming a cohesive construct may be combined in an aspect. So, for instance, in the earlier security application example, the pointcut specifying privilege elevation points and the authorisation advice code may be grouped into a security aspect.

Aspects look very much like classes within AspectJ, but there are some aspect specific constructs, like advice and pointcuts, which are not available in regular Java classes. Otherwise, classes and aspects share many elements. Aspects may include variables and methods, both instance and static.[4] Like classes, aspects may form part of polymorphic and dynamically bound hierarchies, where both the abstract and extends keyword behave much as they do for classes. Aspects may implement interfaces and stand in for these interfaces type. And finally, for now, aspects may also be nested in the manner of inner classes and interfaces.

The runtime operation of AspectJ is best explained by considering how control flow operates in the neighbourhood of an active pointcut. Assume that control in the region preceding a pointcut, is as it would in the nonaspect scenario. The type of the pointcut advice determines how it is triggered. So, for instance, `before` advice is triggered immediately before the pointcut execution, as is `around` advice, whereas `after` advice is triggered immediately after the pointcut execution. In the case of where multiple applicable advices are possible, the AspectJ runtime determines precedence and applies the alternatives in order.

On encountering a pointcut trigger, metaobjects are prepared to provide the advice with access to information regarding the trigger join point, such as its location in the control flow, its application target and so forth. The control flow is passed with this metadata to the relevant advice. Once the advice has concluded, control is returned to just after the trigger join point. An exception applies in the case of around advice, where control is returned just after the original method call irregardless of whether the advice invoked the method.

Before considering a pair of examples, it may help to reexamine how aspects help solve the crosscutting dilemma. The isolation of points where crosscutting concerns intersect the base implementation is done by the use of carefully formulated pointcuts. The behaviour of the crosscutting concern at these intersection points is then managed by the use of advice, although the expression of crosscutting behaviours may also be implemented by augmenting object interfaces and other techniques. The crosscutting concern is managed by aggregation into aspects, promoting the modularity and reusability of the concern.

3.1.3 Example Aspects

§30 Caching. *«Presentation of a simple but extremely effective aspect to transparently cache.»*

Caching is an example of a crosscutting problem introducing a high amount of localised class pollution. Caching results of class or instance operations is typically implemented locally via the storage of precomputations or previous results. Furthermore, all methods producing cachable results, need to be augmented with code to service the implemented caching arrangement. This arrangement need not be uniform and different classes may implement local caching in different ways. Whilst this may be desirable for domain specific

reasons, it is nevertheless scattered code and difficult to maintain and a developer must completely appreciate the particular caching requirements of a class before making changes or updates.

Such localised caching implementation is difficult to centralise without coupling classes participating in caching with a dedicated caching class. This coupling limits class reuse in that classes must bring this dedicated caching support with them. But there are sound reasons to centralise caching, this being the only way to implement consistent cache policies[5] robust to changes in backing storage.

Taken over an entire application, the caching concern has damaging local effects. Every class requiring caching functionality is polluted with fields and code solely serving the caching objective and not that of the underlying class. In this way, class cohesion is damaged, for consider an implementation which requires only optional caching. Implementing separate cacheless classes in the hierarchy is undesirable since it introduces update dependencies. On the other hand, it creates other difficulties to allow cache equipped classes to disable caching. Aside from adding more incohesive code to an already polluted class, there is little point having caching code and structures if they may never be used..

Fortunately, aspects can help and indeed caching is a clear example of the benefits which aspects can introduce to a system design. It provides a viable alternative to the above problematic designs, solving the caching implementation difficulties. The use of aspects in caching is both straightforward and extremely effective. Caching, together with logging, is one of the killer applications for aspects.

The caching aspect implementation involves centralising the caching backing datastructures in an aspect, and updating these structures blind to the classes on which the caching is applied. Each method that is cached or precomputed is captured by a pointcut in the caching aspect. This pointcut is then advised as to how to implement caching for that method. The caching aspect then consists of pointcuts for all the methods to be cached, the relevant advice, and the backing datastructures.

Say, for instance, that an application wishes to implement caching, or precomputation tables, for various often used trigonometric functions. Suppose further that a cacheless version of the application has already been implemented that employs the library trigonometric functions from the Java API. With aspects, it is not even necessary to change the references to the library functions in the base application. Instead, a caching aspect is written that advices the operation of these library functions to serve the caching goal.

Take the case of the sine function, the other functions being treated similarly. The caching structures for the sine results are implemented within the cache aspect, be it in a lookup table, a hashtable of previous computations or whatever. The cache aspect will also contain a pointcut capturing the join point representing invocations of the sine function in the Java Math libraries. This pointcut is given around advice, meaning that it can mask the actual library sine invocation by providing alternative results. The actual advice first considers the sine function parameter, available in the reflective metadata passed to the advice. If the parameter can be served from the cache data, i.e., is present in the lookup table or prior computations table, then this answer may be provided directly to the caller without requiring an invocation of the Java library method. On the other hand, if a result is not available in the cache, the library method can be invoked and the result appended to the caching aspect data store before being returned to the caller.

The use of trigonometric examples is somewhat trivial in that the Java library implementations are already efficient and caching aspect overhead would be prohibitive. This particular domain is used for illustration only. The aspect application pattern remains the same even for more complicated and intensive operations.

This proposed cache aspect solves the design problems described earlier. In particular, it deals with the scattered caching code. All caching code is contained directly within the caching aspect, and base objects are ignorant of whether received results are served from cache structures or from direct method invocations.

Excising caching code from the base objects removes all caching induced coupling and code pollution, increasing cohesiveness. Centralising the caching code also means that implementations and policies may vary independently of the base classes without requiring systemwide refactoring efforts. Note that implementing different caching policies for different elements is possible, since methods may be categorised into pointcuts based on the caching policy to apply and this policy may be implemented in custom pointcut advice.

Also note that optional caching is easily managed within the aspect approach. Globally, caching may be disabled by either using a Boolean flag in the cache aspect, or simply not loading the aspect. This latter solution involves absolutely no overhead whatsoever, whilst the former introduces some unnecessary stack frames and operations. However, the former approach can be refined to offer caching on a optional per method, or pointcut, basis. In this case, whatever overhead is incurred would have been incurred by optional caching support in a nonaspect based implementation of caching anyway. Only in this case, the code would also have been scattered. With an aspect based implementation, the advantages of a separate hierarchy style design for optional caching are leveraged without any of the associated update maintenance problems.

Caching remains one of the most convincing aspect designs. The implementation is successful in solving many of the problems that blight traditional OO caching design. Notice the manner in which the developer is freed from considering the crosscutting concern in every class of the application, and can focus on implementing the base class modulo difficult crosscutting dilemmas. Only once an efficient and well designed base application is implemented, need the developer begin to focus on crosscutting concerns.

The ability to concentrate on a proper implementation, only to later layer the many crosscutting concerns on top of this, is the goal and promise of AOP. The need to mix crosscutting concerns with base implementation is a deficiency in the OO design process.

§31 Logging. *«Logging aspect design and use»*

Logging and tracing support is the second example aspect application, and bears some initial similarity to caching. Many crosscutting concerns are similarly effectively tackled with this basic aspect pattern.

The logging aspect tackles the crosscutting concern of reporting occurrences of particular conditions and software events. At such an interesting event, the logging aspect is to augment a log file, or console output, with a brief text message describing the condition. The logging aspect design is to log in the manner of the Log4J package, without requiring the scattering of logging calls throughout the logged application.

Preventing scattering yields the same benefits as in the case of caching. But a particular issue to logging is internationalisation. By centralising logging operations and associated messages, internationalisation and

translation is simplified. This is not an alternative to gettext or other well-designed internationalisation tools, but serves to motivate a potential benefit in smaller applications.

Although logging may be implemented using existing architectures and tools, there are distinct advantages in using aspects. Apart from minimising log call scattering, there is also the reduction of string construction operations associated with logging calls. Typically, a log call made from within an application might involve string concatenation to form the display message. This concatenation can be a serious overhead when done at the point of the logging call, especially if logging may be disabled in the application. For although disabled, an application may still be stuck with the unnecessary string concatenations. Using aspects can avoid this.

A logging aspect, firstly, possesses a backing log mechanism, either handcrafted or leveraged from an existing logging framework. In addition, the aspect can maintain the list of log messages, by the use of an external system like gettext, by listing the messages statically, or by other means.

To implement logging calls without polluting the class sourcing the log event, a pointcut is designed to capture the desired event. This event may be a method invocation, a field access, or usefully, any Boolean-testable Java occurrence. So, timing and condition events depending on scattered state can be easily checked also. On tripping a specific pointcut, advice sends the relevant message to the logging backend.

Log message scattering is significantly reduced in this scheme, and string concatenation penalties may be avoided when logging is disabled. If string concatenations are done within the aspect advice, and if logging is disable by not loading the logging aspect, then the string concatenations are bypassed.

Before leaving the issue of aspect logging, it should also be noted that logging is just one of many useful aspect applications in debugging. The ability to inject documentating actions into a class without modifying that class is a great debugging convenience. For instance, aspects can be written to print stack traces during execution, to display formal parameter values on method invocations, dump instance field contents at selected events such as exceptions and in not-possible control flow, etc. These, and other creative aspect applications in the debugging process, are easy to implement and of practical value.

3.1.4 Aspect Use in WebCom

This project has either trialled or included a number of aspect applications within WebCom, including using aspects for internals assertion checking and for logging existing log messages.

The logging application highlighted the necessity to design applications with aspects in mind. In the case of WebCom, a mature logging implementation already existed before an aspect implementation thereof was considered. With the restrictions of traditional OO design, this existing logging support was highly scattered, and given the extensive use of logging it proved too time-intensive to refactor into an aspect. This is a case of the damage already being done, and merely highlights reasons to encourage aspects use.

There is also potential for the use of caching aspects within WebCom. In particular, caching may be used on referentially transparent operators to great effect. These operators are already identified and optimised within WebCom anyway, and the use of caching aspects may streamline these operators even to the extent that

they need not be explicitly segregated within WebCom in order to apply optimisations. A uniform treatment of operators, with cache optimisation for referentially transparent operators, would improve efficiency.

Debugging support aspects are commonly used to help isolate problems within WebCom, and illustrate further aspect utility within WebCom, even if they are absent from the deployed WebCom version.

However, perhaps the most interesting aspect application within WebCom is the Event API, the focus of the next section. This API facilitates informing interested parties of events within the WebCom core without requiring this core architecture to propagate or even be aware of these events. This application of aspects is also invisible to the end programmer, and forms part of the Module API, the subject of Chapter 4. Aspects play a critical role in the overall design of the Module API, in that the API design is guided by a requirement not to modify or pollute the core WebCom architecture.

The introduction of the aspect compiler and the proliferation of aspects behind the scenes in WebCom, is also interesting from a paradigm point of view. The section on the Event API illustrates the use of aspects in facilitating an event driven programming component within WebCom. This Event API architecture may be used, or abused, by endusers to program WebCom applications and modules in an event driven fashion.

Presenting new programming paradigms for enduser module programming is a major theme of this dissertation. The Event API, not to mention the AOP support, adds to the capabilities already available to module programmers. Note, that module programmers are free to use and construct aspects within the implementation of custom modules, hence the enduser availability of AOP.

Furthermore,logic programming facilities will also be made available later in the course of supporting type checking. The type checker implementation involves adding a general purpose resolver to the WebCom architecture, and this resolver also makes logic programming methodologies available to module designers. In fact, the type checker module is interesting in that it blends traditional OO, logic programming, and even event driven techniques under the guise of the aspects in the Event API.

3.2 Event API

§32 Event API and Origins. *≪High-level description of the Event API. Reasons prompting development of this API≫*
The Event API, part of the wider WebCom Module API documented in Chapter 4, is an interface providing module writers with information regarding events occurring within the WebCom core, without requiring modification to this core WebCom. This event information availability is crucial in effective module writing, yet ringfencing the vulnerable core WebCom from code pollution is likewise essential. Modifying core code for the purposes of implementing optional third party modules should justifiably be prohibited.

The Event API is implemented via aspects, but hides this fact from Event API users. Aspects provide the mechanism to make Event API users aware of events without modifying the core WebCom classes.

The WebCom core consists of the backplane, providing module loading and message passing, together with some core basic modules. These modules include the connection manager, the load balancer, the scheduler, the fault tolerance module, engine module and security module. However, these modules themselves are

not core. It is rather the roles and interactions of these modules within WebCom that is core. For instance, the particular implementation of the scheduler module is not á prior core and may be replaced as desired by equivalent scheduler modules. However, the scheduler is required to behave according to certain rules and this is core to the WebCom machine. Historically, this interaction was encoded in a particular base scheduler implementation and has evolved with changes to this implementation. This is undesirable, since core object specification moves based on reference implementation changes, whereas the opposite should be the case.

This is true of all the core components, although the actual impact varies. The engine and security modules are relatively straightforward in their operation and a semantics could be easily outlined. The connection manager operates in a manner that often divorced from the other modules, and may be tackled in isolation, something which simplifies specification. The remaining modules, the scheduler, load balancer, and fault tolerance modules have necessary interdependencies that bind their specifications into a single component.

The unfortunate upshot is that WebCom internals specification depends on the implementation of a set of reference modules. It is essential therefore to protect these implementations and their stability. And it is to this end that the Event API most serves a purpose.

The implementation of interesting additional functionality in WebCom often involves the augmentation of these core modules, and a consequent alteration of core module specifications. Typically, this specification update need not really be required for the additional functionality, e.g., the implementation of statistics collection should be orthogonal to core module purposes. In practice, it is necessary to change core modules to implement statistics generation. This is undesirable.

In the case of statistics module and some other cases, providing event data is the sole function of core module changes required. And if more detailed core module modifications are necessary for the particular extension demands, the extension will still also require event data anyway. So, even if an application still necessitates core modifications, these can at least be mitigated by limiting the event generation modifications.[6]

Reducing the need to modify core WebCom classes for event generation purposes is a first step in removing code pollution in core modules. The Module API, see next chapter, is designed to further reduce core code pollution by addressing other typical requirements of extension writers, including module presentation.

It is critical to appreciate that code pollution proliferation in the reference modules also pollute WebCom core specification. This has made it virtually impossible to implement alternative core modules, since the specification is informal and likely to move from under any alternative implementations. This situation damages confidence in the kind of modular architecture that WebCom professes.

3.2.1 Aspect based event system

§33 Event System Implementation Difficulties. *«A summary of the difficulties»*

Essentially, a mechanism is needed to inform interested parties of the occurrence of a specific events within the core set of WebCom modules. This must be implemented with the least amount of modification in existing code, and furthermore, be robust to changes in the base core modules. It should be possible to augment the

event set with new events, as determined to be useful in future development, without requiring significant modification. Neither should the addition of new events be vulnerable to changes in the core module code.

These requirements are articulated to highlight aspect strengths and indeed the aspect based solution does satisfy these requirements. In particular there is almost no modification in the existing WebCom, the only modification being a trivial code rearrangement, entirely for convenience.

Deck stacking aside, the aspect based implementation outlined below is superior to a tradition implementation, which could not avoid introducing event dispatch code at relevant places in the core code. This falls foul of all the inherent problems in such crosscutting and scattering, so despite any clever engineering, the implementation will be crippled by this scattering.

Maintenance would be excessively timeconsuming. For example, removing an event would mean locating and excising all dispatch instances for that code within the core classes. Similarly, adding an event would require locating, by hand, all positions in the code base where this event occurs and inserting dispatch code. Accidental error introduction aside, there remains a problem inheriting these events into subclasses. A tradition solution is just not feasible. It is impractical to implement the reference classes, and module hierarchy bases, in a template method fashion in order to permit subclasses to acquire event hooks by very selective extension of superclasses. It does not work either since it is not robust to the introduction of new events. A new event would potentially mean changing the base class and its template method structures, necessitating further changes in all implemented subclasses.

§34 Lightweight aspect events. *«The aspect based solution.»*
Aspects permit a robust event system implementation with minimal modification to the existing code. The implementation is straightforward also. Essentially, the source code positions of required events are captured by pointcuts. It is worth noting that these events often have properties suitable for concise pointcut capture. For instance, operator triggering is an event of interest, and the operator interface specifies the exact name of the operation implementation method. So, a pointcut expression can be made using wildcards to capture all these operation methods, rather than do so by tracking internal WebCom management and invocation structures that may be subject to future change.

Once pointcuts have been designed to capture event locations, one event group per pointcut, advice can be used to dispatch events indications. That is, each event type is captured by a specific pointcut, which is then advised how to generate and dispatch corresponding event objects. An aspect aggregates these pointcuts and relevant advice, and also maintains the event dispatch code and listener lists. Event dispatch is done in an essentially identical manner for each event type and benefits from central management.

The reflection metadata at the pointcut advice can be used to extract parameters and details pertaining to the generated event. For instance, join point metadata can be used to populate an event object with operator and operand references in the case of an operator trigger event.

Using aspects in a lightweight event system has many advantages, beyond those described earlier. Disabling events is then trivial, and may make sense in the WebCom context. The Event API system is only

intended to provide information to interested parties, the provision of such data is not demanded or guaranteed. If a site administrator decides to optimise WebCom purely for basic job execution, then the event system is overhead. With aspects, this overhead may then be eliminated, without requiring a recompilation.

§35 Adaptor Interfaces. *≪Obsfucation of aspect implementation. Presentation of event system as traditional OO API≫*
Aspect based implementation presents some difficulties, especially in an environment that is new to AOP. As such, it is desirable to minimise overlap between aspects and core interfaces, reducing the threshold for group developer contribution. Interfacing with aspect based code requires some AOP familiarity and for many developers in large systems like WebCom, there is little immediate requirement to learn aspect techniques. Incorporating an aspect based subsystem needs to done with minimal necessary paradigm shifting.

The use of adaptor interfaces is important within the Event API for exactly these reasons.[7] The adaptor pattern is used to present a traditional event dispatch-response interaction to module developers. Specifically, client application register for events and are called back with event description objects on occurrence of these events. Nowhere in client code is the aspect based implementation of the Event API visible.

Consider the example scenario of an event propagation. Client code expresses interest in a particular event by registering with an Event API management object, and is added to a listeners list. The aspects capture events and dispatch them to the client code by way of this Event API management object.

The Event API consists essentially of static elements. Listener lists are maintained on a class basis, as are the event propagation callback methods used by the aspect. And whilst there is an indirection penalty, it can be aggressively optimised. This penalty really consists of the extra static class and of an additional method invocation per event. The list management functionality of the static management class is required, and eliminating this class would mean moving this functionality to the aspect anyway.

The extra method call is to a small static method which may be inlined easily. This inlining is simplified since the method only requires data read access to the list objects within the static management class. Because this data is also static, references may be used within the event aspect.

The static middleman class might be incorporated into the event dispatch aspect at the cost of requiring client code to register with the aspect, instead of with a static class. The registering would be done via a static method in any case, so end users might still pretend that the invocation were being done on a class.[8]

The advantages of nonaspect client code is significant and worth the overhead, optimised out anyways. There is benefit in facilitating third parties to ignore the aspect nature of the code and yet still apply the API.

3.2.2 Event API Implementation

§36 API and UML. *≪An examination of the UML outline for the Event API organisation.≫*
Figures 22 and 23 detail the implementation of the Event API, with Figure 22 illustrating the aspect implementation and OO masking. Note, class notation for aspects in this figure is incorrect but unavoidable.

The `EventAPIAspect` aspect maintains pointcuts capturing a range of events within the WebCom internals. Sample events include instruction creation and execution, message transmission, graph memory

3.2 Event API

Figure 22: UML Diagram Event API Part I.

3.2 Event API

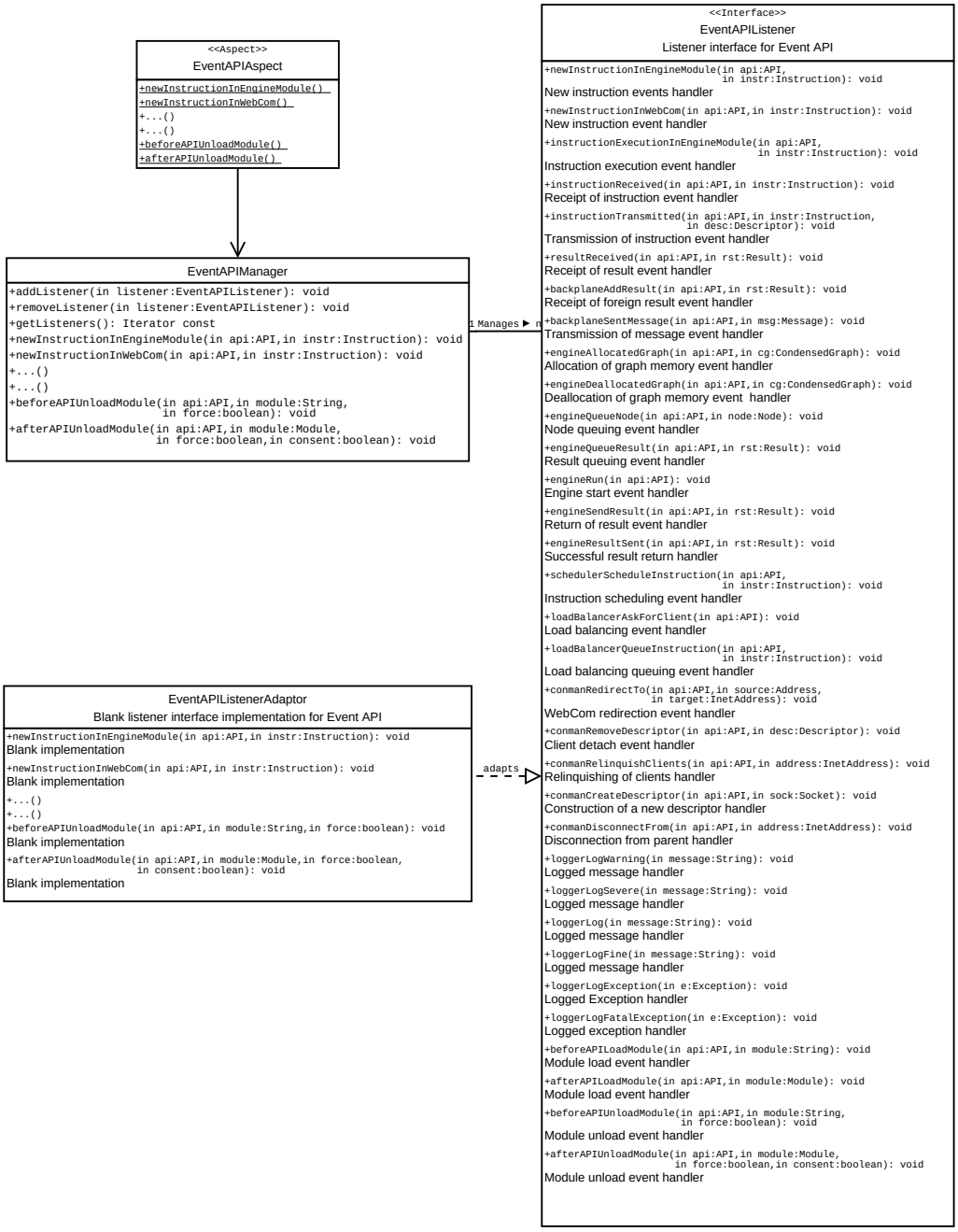

Figure 23: UML Diagram Event API Part II.

allocation, etc. At event capture, pointcut are advised how to extract pertinent data from the reflective data structures. For instance, at new instruction events, the WebCom instance is stored in an `API` reference, and the actual new instruction in an `Instruction` reference. This event data is then passed to interested clients.

The actual propagation is done via the `EventAPIManager` class, which maintains and manages interested listeners. When the aspect raises an event, it forwards the event and relevant data to a matching method in the `EventAPIManager` class. This method then informs registered listeners of the event.

Interested parties implement the `EventAPIListener` interface, or alternatively implement the blank adaptor `EventAPIListenerAdaptor`, from Figure 23. This interface lists the matching methods that the `EventAPIManager` class invokes to forward event notifications it receives from the aspect.

Since these event interfaces are registered via the `registerListener` method of the manager class, the Event API appears as a traditional Observer pattern to third party programmers. Note, only the registration methods in `EventAPIManager` and the `EventAPIListener` interface need be visible to third parties, making the pattern appear traditional even though it is implemented via the `EventAPIAspect` aspect.

A developer wishing to use of the Event API need only implement and register a `EventAPIListener` interface to respond to available WebCom events. A basic example of this relatively straightforward development cycle will be demonstrated in the next section. However, this is still not a fully workable solution to end programmer requirements, since it misses mechanisms to directly query the WebCom core and to request particular WebCom actions. These are the subject of the Module API, the focus of Chapter 4, which together with the Event API constitutes a fully featured API while maintaining the integrity of WebCom internals.

3.2.3 Applications

The production of an event trace tool forms a simple illustration of the Event API. This tool reports on all exported event detail from the WebCom core in text format. Such a tool has applications both in debugging WebCom interactions and in providing information on internal WebCom state transitions. An events listing may also partly substitute for a core dump within a specialised WebCom debugger.[9]

The development of the tracing module demonstrating the ease of Event API application. The main programming is contained within the `DebugEventAPIListener` extension of the `EventAPIListenerAdaptor` class. This listener maintains a `PrintWriter` object, or output stream, to forward details of generated events. This output is constructed with the listener, together with shutdown hooks to close and tidy resources.

The `DebugEventAPIListener` responses to all the generated event endpoints available and dumps a text based description of each to the maintained output stream. This is the full extent of the programming required, and consists of little more than I/O output statements in the listener interface methods.

There is some additional framework code outside of the `DebugEventAPIListener` class that has to do with a supporting skeleton module and with listener loading and unloading. This code is minimal, comprising a module to just perform listener registration, thereby starting the event tracing operation. The mechanism by which this simple module is loaded into WebCom and set in operation is the focus of the Module API and covered in the next chapter.

Figure 24: Odd parity testing graph.

By way of brief example, consider the graph in Figure 24. This linear graph returns a Boolean indicating whether the input integer has odd parity. An execution event trace of this graph is included in full in Appendix B, together with an outline description of the event sequence. This is useful for a detailed understanding of WebCom processing, but is unnecessary for the purposes here. Instead, some illustrative sample events will give the necessary flavour.

The following event illustrates construction of a new instruction to handle the toplevel graph as a dynamic operator. The new instruction refers to the condensed operator, although the reference is not so helpful.

```
newInstructionInWebCom :-
webcom.core.Instruction@1e6e305
operation main
```

The next example event demonstrates the capture of a message transaction within WebCom. At this point, the WebCom core is passing a message containing the toplevel instruction created above.[10]

```
backplaneSentMessage :-
EngineMessage: source = agador/143.239.211.35 module value: top level,
destination = agador/143.239.211.35 module value: engine,type = INSTRUCTION,
data = webcom.core.Instruction@1e6e305
```

The final example event illustrates event callback on the execution of an instruction within the engine module.

```
instructionExecutionInEngineModule :-
webcom.core.Instruction@9abc69
operation webcom.nodes.core.EvenOp
```

These examples demonstrate some of the events available for capture. There is a wide range of events that interested parties may listen for, encouraging the design of versatile applications. Although the event trace application is somewhat trivial, it is nevertheless quite useful. However, more potentially interesting example applications have been deferred until the Module API harness structure has been outlined, and there will be further demonstration of the Event API within useful modules and applications in the next chapter.

3.2.4 Final remarks

This general pattern in using aspects to engineer an event system exposing conditions and occurrences, whilst protecting the integrity of a core software artifact, may be deployed in other circumstances. Although in

WebCom, there is no other immediate cohesive core block that might benefit from a similar design, this general means of ringfencing fragile code can still be reused to benefit in other applications.

The Event API system provides yet another example of how aspects may be used to cleanly manage difficult crosscutting concerns. In this case, the aspect design simplifies the implementation of what would otherwise be a severely scattered event dispatch system. Moreover, it would not have been practical to retrofit such a scattered event dispatch mechanism in the core code base. In this sense, AOP has enable functionality which otherwise would have been excessively timeconsuming to implement and maintain.

The next stage of development is to fit this Event API into a wider module support framework, including load, query and operation mechanisms. This will form the implementation framework for the eventual type checking application, and is the subject of the next chapter.

Chapter Notes

[1]The use of a small Jar library is required at runtime to enable the aspect constructs, though.

[2]Although phrased in terms of adding a new component to a system, this problem should have been addressed in isolation even if security was an initial element of the software. Presentation in terms of refactoring simply highlights the particular crosscutting concern.

[3]The difference being that on method invocation control flow lies outside the method being called, whereas upon method execution, control flow lies within the called method.

[4]The distinction between instance and static members is more subtle in aspects than in classes. The typical arrangement is that there is a single instance of a particular aspect per Java Virtual Machine. In this circumstance, the difference between static and instance members is simply one of initialisation. It is possible, however, to arrange aspects on a per object or control flow basis, in which case the difference between instance members and static members is the same as in classes.

[5]Even if policies vary between classes, they may still be managed centrally to greater effect than in local caching implementation.

[6]The actual goal of the Module API is to remove all reasons to modify core modules to achieve particular extensions. But it remains the case, that events are a separate concern and explains why the Event API is really a component of the Module API.

[7]The original ideas for adaptor interfaces were suggested by Keith Power.

[8]Which in fact it is since the aspect is likely to be realised by AspectJ as a class together with additional code to trap the aspect pointcuts and forward to the aspect advice. The advice is naturally likely to be implemented as methods in the implementation class.

[9]A WebCom debugger or replayer is under current development and although, eventually may not be based on the Event API architecture, still represents a potential application of the Event API.

[10]It is the same instruction because the references match.

4

Module API

The Event API can be complemented by providing support for two further areas within the sphere of module programming. These areas include the mechanisms and semantics for loading, unloading and configuring modules into a running WebCom system, together with the means of querying and requesting the core WebCom to undertake particular actions on behalf of a module.

The Event API together with these additional components are collectively referred to as the Module API. However, this chapter will focus almost exclusively on the two new components, only mentioning the Event API in passing. There is Module API term is often used to refer to the querying and action operations exclusively, note. This usage excludes both the loading mechanism and the Event API.

The design philosophy behind the Module API will be described, together with sketch implementation details in the first half of this chapter. The remainder covers examples of the application to information gathering and actioning operations within the module loading context. These examples illustrate the versatility of the new organisations and how they are used to engineer more pervasive WebCom operation modes.

4.1 Module API

The Module API implementation makes two primary modifications to existing WebCom structures. The first adds an `API` class to represent a single WebCom instance, and to facilitate query and operation interactions with that WebCom. Although, existing classes may be used to refer to a WebCom instance[1], it is better to

abstract the concept of a running WebCom from its implementation. The `API` class, detailed later, is also the handle returned by Event API callbacks referencing the operation context sourcing a particular event.

The second main programming change involves modifying the WebCom internals to arrange for uniform loading and unloading. The adoption of a uniform load process, and retrofitting it to core modules, greatly improves the scope for third party modules. Such modules may then be programmed and incorporated seamlessly into WebCom instances in a simplified manner. These loading triggers and callbacks together with a refined addressing scheme[2] form the dynamic load mechanisms and will also be covered in more detail later.

The most interesting facet of the Module API, though, is in the reworking of the module role within WebCom. Changes within the Module API introduce a more powerful and flexible view of modules. So, even while the changes are fundamentally simple, they present a subtle and important change. These changes are incremental, extending the existing module concept and applying it in circumstances previously either not possible nor envisioned. The next section considers this new position and module philosophy.

4.1.1 Philosophy

Reworking the module code raises new potentials for the module system, include the promotion of modules to first order elements within WebCom. While currently, modules are the main component of WebCom, the new structures make it practical to specify that modules are the *only* permitted WebCom components. This is, modulo some base plumbing, reworking the WebCom design into a completely plugin architecture.

With a plugin[3] architecture, there is increased facility to isolate and protect core WebCom components. These core plugins, or modules, may be detached and strongly ringfenced, in this way reducing code pollution and maintenance trauma. This extends the guiding philosophy employed in the use of aspects earlier.

Another main change in WebCom practice is the introduction of dynamic module loading and unloading, as a byproduct of the uniform loading mechanism. This is support and scope for changing the running WebCom environment, for changing the capabilities and the operation support available. There is even potential to load core WebCom facilities and even instances, in some senses, as modules. This is particularly useful in conjunction with the IDE and other graphical WebCom tools, as will be seen in the examples later.

Although these constitute the primary changes in outline, they neglect some of the lesser new features. These are explored somewhat, but not exhaustively, in the example modules and include concepts such as GUI modules, bridge loader modules, the construction of WebCom applications from collaborating basic modules, and the aforementioned use of modules to instantiate WebComs and other WebCom artifacts.

§37 Completely Plugin Architecture. *«WebCom as a confederation of interacting modules»*
Nearly everything within WebCom may be considered as a potential module, and implemented via the module construct. That is, all functionality outside of the basic graph representation and handling, and the otherwise bare minimum in WebCom could be implemented in module form.

Take the Eclipse IDE as a model, being as it has illustrated the purely plugin design philosophy to a wide audience. In Eclipse, everything barring the bootstrap is implemented in a plugin. Now WebCom itself

already has a strong module philosophy, and with a subtle shift in viewpoint, could easily be similarly viewed as module-everywhere or purely plugin. The ramifications of this for future design are significant.

Promoting a purely plugin WebCom architecture does not mean that modules are all equal. It should be emphasised that there are two very distinct classes of modules. The core modules, like the engine and connection manager, are fundamental to WebCom operation and an instance of each must always be present. Noncore modules fall into a category of entirely optional modules. A site administrator may load or unload these modules to form WebCom configurations.

Because there are firmly two classes of modules, loading uniformity is not perfect in that the bootstrap must always orchestrate a core set of modules. But aside from this, the interface offered to modules can be entirely uniform. Once a core WebCom can been acquired, all modules operate according to the same rules. It is the case, of course, that the more important core modules already have defined interactions, which bind these modules into a cooperating tool to drive graph reductions.

Although core modules are not independent, there are no requirements for optional modules to be dependent on other modules. There will, presumably, be dependencies on core WebCom modules and for instance, on the message passing structures available via a backplane module, but there is no reason why optional modules need cooperate with other modules to perform tasks. Optional modules may either stand alone modules or work in a larger grouping of modules directed at some goal.

The basic WebCom module is defined as a cohesive block of code, or collaborative element, that may be added or removed from WebCom to provide or support additional functionality. The module concept is a design construct for the aggregation of code, but one convenient for developers of WebCom features.

The interface to optional modules is reasonable straightforward to state, contrasting with the difficult interactions in core modules. The optional modules possess barrier interfaces, facilitating a exploitation of plugability. The goal, although difficult, should be to facilitate similar levels of plugability in core modules.

The module everywhere notion that this suggests is a very flexible circumstance. For example, the ability to treat modules uniformly helps in the development of user tools such as the SysTray application, a background GUI widget providing access to WebCom. By virtue of module proliferation, this application is little more than a GUI module loader tool, itself a module, with WebCom features being made available also via the module loader. The implementation of this application, discussed later, as a module itself leads to a variety of convenient launching scenarios.

§38 Ringfence Core Machine. *≪Discussion of the use of the Module API to promote a leaner core WebCom implementation≫*
A lightweight view of optional modules is a helpful software organisational tool, but not one limited just to the implementation of additional functionality. There are many core features which could be managed by this approach. These include for instance, logging, class loading, state server and client, database operations, etc. These functions can be implemented conveniently via the use of optional noncore modules and such implementation has a desirable trimming effect on the core code base.

In addition to not strictly fundamental functions within the WebCom core, there has also been a tendency to implement new WebCom features by augmenting this code. This is a destabilizing behaviour which

should be discouraged within the WebCom design. Applying new features within the WebCom core degrades integrity and cohesion, confuses core documentation and is oftentimes unnecessary.

In practice, however, there hasn't always been a workable alternative available. The Module API aims to plug this gap with optional noncore modules.[4] The Module API and purely plugin view of WebCom offers a safer, cleaner and more robust means to implement WebCom extensions. Additional, nonbasic functionality can be provided by means of new modules.

Internal modules need to be pared to the bone and set in stone, for design, maintain

The core modules should only include the barest minimum functionality, sufficient to arrange graph reductions only. This includes the backplane support, the connection manager, scheduler, load balancer and fault tolerance modules, together with an engine module implementing the basic WebCom operations, with each of these elements striped of unnecessary functionality. The security manager, is typically implemented via a blank interface, and so is not a concern in core bloat minimisation.

Typically, what is imagined to be core is not really. For instance, the transmission of class descriptions to peers is not properly a core operation. If peers do not have the required classes to implement an operation, that operation will be returned to the originator via other WebCom channels. But the current core implementation of a class server and client structure presents drawbacks. For instance, it is not straightforward to remove the class serving facility. Although it may be configured not to run, it still represents a separate initialisation and configuration step that would be better served within the framework of a module. Then if the site administrator does not require the class server, the whole module need not be loaded.

The configuration of somewhat primitive functionality may be managed by site administrators using the plugable nature of WebCom configuration in a simple and even dynamic manner. This dynamism is currently not available with the present configuration schemes and models a longer execution cycle view of WebCom. The present configuration mechanism is oriented toward one-shot WebCom executions mostly, whereas a more dynamic loader scheme supports a pervasive, always available WebCom operation.

Most support considered core currently may not be so, and may be suitable for optional module implementation. There is great benefit in forbidding unnecessary core module changes and the promotion of a module-everywhere approach is a step in this direction. For, a solid alternative modular design is clearly a better place to implement functionality. That said, it cannot also be denied that there are occasions when the core modules are the correct place to make changes.

Core modules are not special. They should ideally be programmed in exactly the same manner as optional modules, modulo the load discrepancy. So, while much core module interaction is current done by direct method invocations on supporting modules, they might be better served using methods available in the Module API, to be described presently. These facilities are already available to the core modules anyway, and the modules may as well leverage these generic access hooks rather than use direct references.

Historically, core modules have depended greatly on the backplane interface, which in many ways, has been used to refer to the particular running WebCom instance. The Module API includes a new `API` class to play this role, and avoids problems in using the backplane as both machine identifier and implementation.

This is not a major concern, until the presence of two backplanes within a single WebCom machine is considered. Although this is not a currently permitted configuration, there is little technical reason why this should be the case. There are interesting WebCom topologies implementable via the use of backplanes, since backplanes themselves are also modules and so may be plugged into other backplanes. This could conceivably be used to implement module chainings and hierarchies in structural rather than policy based approaches.

Using the Module API within the core modules might also help with internals documentation. It would, at least, help determine border protocols within core module interfaces, since such border interactions would be via the Module API. It may be that a core version of the API is needed. Some of the core module interactions are coupled tightly, something which should not be apparent to public users of the Module API, and which may be hidden within a protected core Module API.

The core modules will always be core, though, regardless of whether they use the API. Their interfaces denote required functionality which the Module API helps decouple from module implementation.

§39 Module Loader. *«Notes regarding the load and configuration processes for modules»*

The use of the Module API in WebCom configuration has already been mentioned to some extent. This and the notion of swappable or dynamic modules are the points to note with regard to module loading procedures.

The Module API sets out a definite mechanism for loading and unloading modules. Previously, it had been acceptable to manage this in ad hoc fashion, since the view of modules was more limited. Module load and unload semantics is an essential key in promoting WebCom as a module-everywhere architecture.

The load mechanism is straightforward. Modules are loaded automatically at the WebCom bootstrap stage if specified in a particular configuration file. Additional modules may be loaded later from within WebCom by any element with access to an `API` reference, i.e., via proper procedure within the Module API. Modules are constructed using a no-argument constructor for historical reasons, but are offered an initialisation callback once they have been attached to their target backplane and before any further WebCom configuration.

The unloading process is similar. Modules are unloaded at WebCom destruction, or as specified by Module API actions. In order to preserve internal state, there are currently restrictions on unloading core modules, but optional modules are always detachable. Modules are offered a chance to perform specific unload operations, or to softly refuse unloading. However, the Module API man be instructed to force an unload if necessary, the module being offered notification of this also, but no right of refusal. The mechanism will then ignore any further module resistance.

The loading and unloading is more significant than the actual mechanism, and may be used to replace currently loaded modules, or to add modules required for specific operations. There is potential to manage module dependencies, indeed there is an initial `Info` object structure to support future work on this.[5]

This initial approach may be improved, particularly in regard to unload semantics. For instance, in-transit messages referencing an unloaded module lose an endpoint. Such messages may be dropped, rather than left outstanding, so as waiting modules may be notified of the termination in the peer message endpoint.

As noted earlier, swappable modules mean that basic WebCom configurations may be created and left operational. User desired functionality can be dynamically incorporated as required by the WebCom operator.

In particular, there is good means to run WebCom as a background task in an interactive desktop. This background WebCom may service WebCom operations from a range of sources, the user, other applications, network peers, the operating system. Such an operation mode is a step toward a WebCom operating system implementation. Later examples in this chapter demonstrate that this operation mode is practical, and at illustrate how WebCom might evolve toward an operating system implementation.

A wider range of optional modules means a wider range of functionality and configuration options within WebCom. The loader mechanism provides for the seamless incorporation of both core behaviour and lightweight functionality. To the enduser, this can manifest as rich versatility in the WebCom application.

4.1.2 Design

The section takes described the various classes in the Module API, aimed at illustrating the module writing process. It begins by considering the `Module` superclass which modules must implement, and describes communication mechanisms available to module writers. The `API` class and components are then outlined, illustrating the action and query facilities available to module writers. There is a final section describing the module load and unload procedures.

§40 Module. *«Description of the facilities present via the* `Module` *interface and available to third party module writers»*
Module writers have access to a number of Module API resources to perform communications and actions. In regards communications, the previously discussed Event API describes means by which internal elements of WebCom may communicate with user written modules and share event information.

While third party modules cannot augment the set of events, there is existing support to provide an Observer pattern within user written modules. Each subclass of `Module`, the basic abstract interface for all modules, is prefitted with event dispatch and listener list maintenance features. Implementing an additional heavyweight event system from a module is a matter of subscribing to particular methods and interfaces.

The other main communication mechanism available to module writers, and the only one which facilitates intermachine communication, involves WebCom message passing. Subclass of `Module` are required to implement a callback to handle messages destined for that module. These messages are sent via the `ActionAPI`, discussed presently, and are formatted as `Message` subclasses. Actual data content and interpretation is left to the discretion of the module writer. So, module writers wishing to pass messages to separate machines, may implement a new `Message` subclass to use with the builtin message passing routines.

`Message`s are addressed by means of `Address` classes. This class provides a specification for destination machines by either internal socket reference or by Internet addresses. The target module is currently specified by module classname, although this is under review and may change in the future. The important core modules may be directly references using certain constants, so it is not necessary to know the specific core module implementation in order to address it.

The `ModuleInfo` classes provide module metadata, and are named conforming to a naming structure so the system can automatically find metainformation for particular modules. The `ModuleInfo` for a particular

4.1 Module API

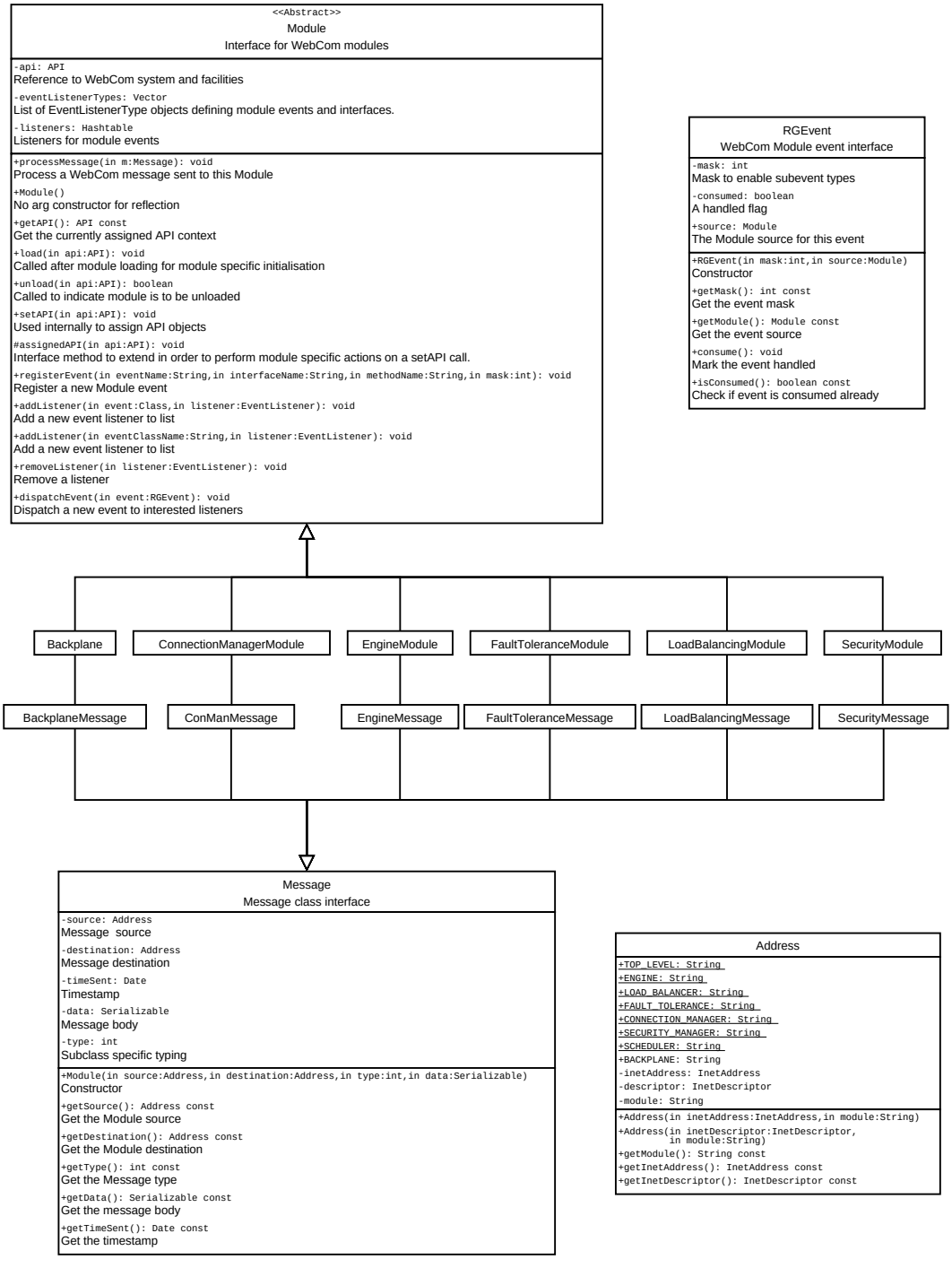

Figure 25: UML Diagram Module API `Module`s and `Message`s.

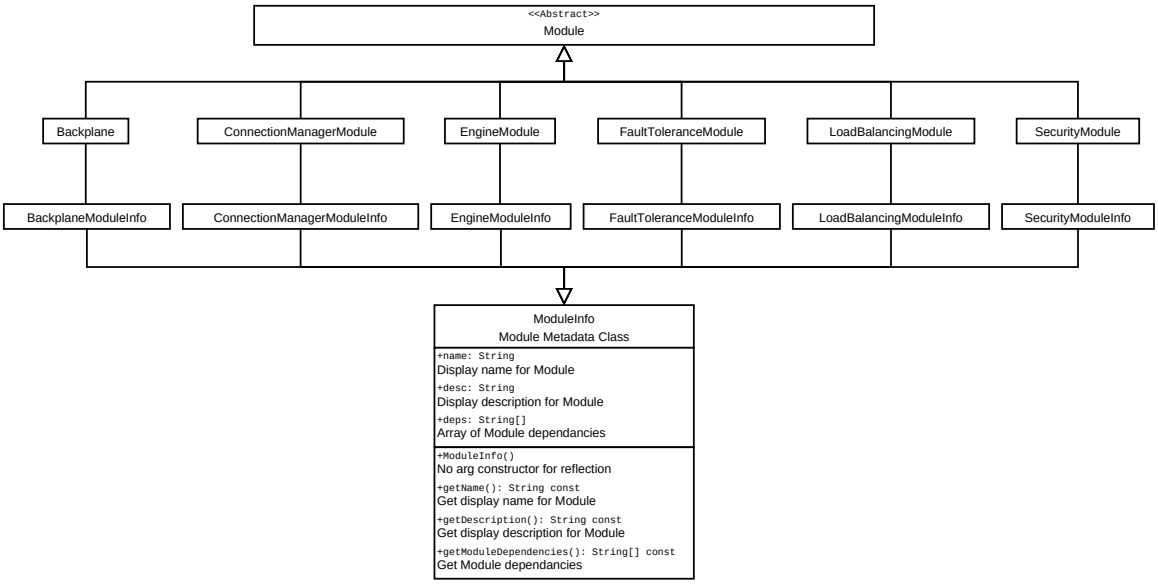

Figure 26: UML Diagram Module API `ModuleInfo`.

module should be contained in the class with the original module name suffixed with `ModuleInfo`. So, if metadata is required for module `Blah`, the system expects to find it in the `BlahModuleInfo` class.[6] The `ModuleInfo` class provides metadata hooks for `Module` display names and descriptions, together with an currently unsupported list of module dependencies.

Module writers must subclass `Module` and implement their basic module elements. This class is loaded automatically, or on demand, and subject to the specific load actions described below. The load call provide hooks for module initialisation, event registration and other preparation work. If the module is to use message based communications, the write should prepare a custom subclass of `Message`, and implement the module `processMessage` method to process these messages. The message routing is handled automatically. If interested in internal WebCom events, the module must register with the `EventAPIManager` as in the previous chapter. If it wishes to provide interested parties with access to heavyweight events generated within the module, there must also be some initialisation and registration code, together with dispatch code at the relevant points. Module writers should also implement a `ModuleInfo` class to provide module metainformation.

Figure 25 diagrams the basic `Module` and `Message` interfaces. Figure 26 illustrates `ModuleInfo`.

§41 API Class. *«Queries and operations available to third party module writers»*

All modules have access to an `API` object, via their `getAPI` methods. This `API` class represents the WebCom machine instance, modules getting the `API` referring to the WebCom machine onto which they are loaded. In particular, this means a module may only be loaded on one WebCom instance within a JVM.[7]

4.1 Module API

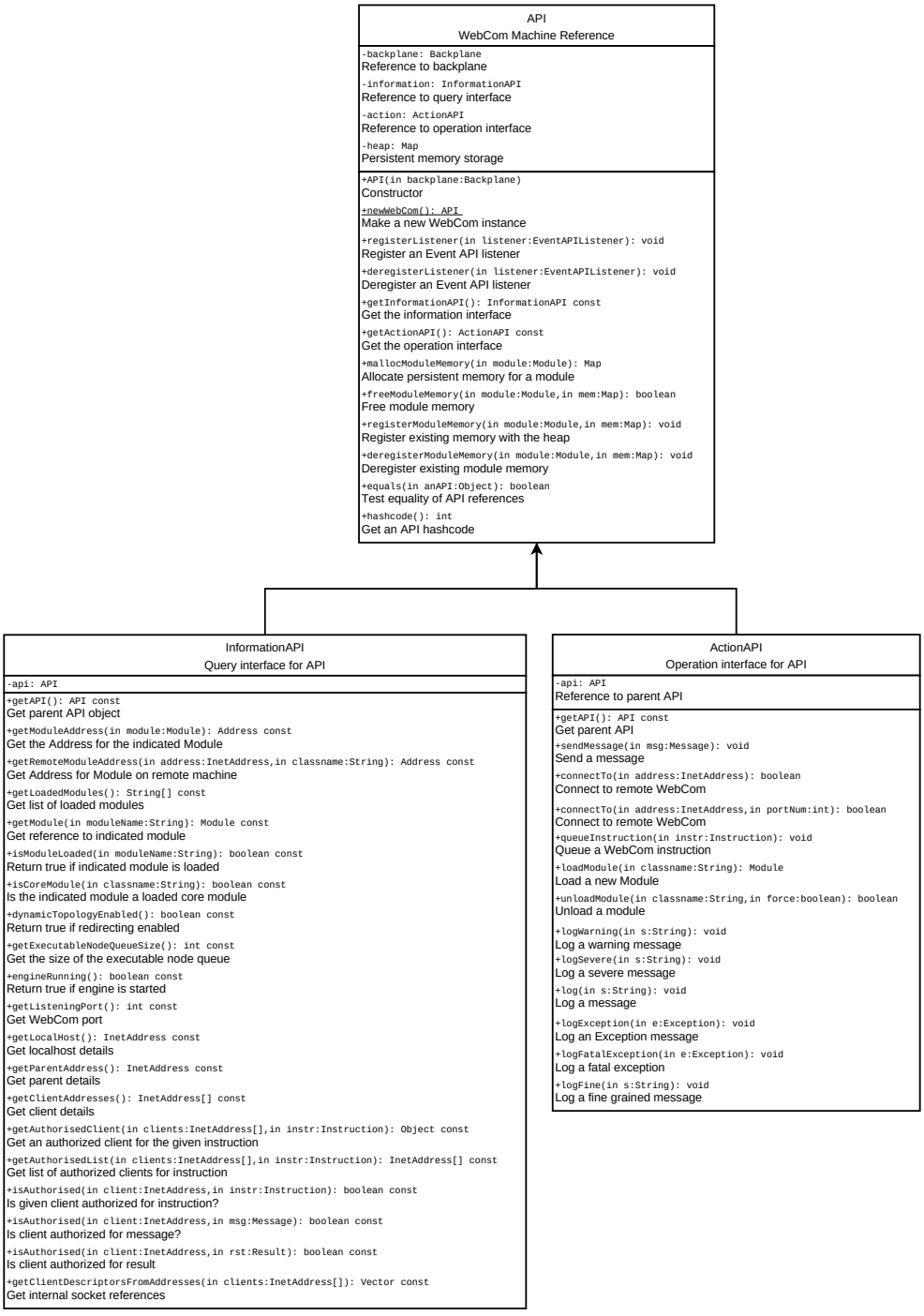

Figure 27: UML Diagram Module API.

The API class, illustrated in Figure 27, serves as a placeholder reference for the WebCom machine, and provides access to resources and operations on that machine. The API class may be used, instead of the EventAPIManager, to register and deregister for Event API events, and is probably more convenient.

Modules may avail of a persistent memory store, which exists so long as the API does, persisting across module loads and unloads, but not currently across WebCom instances. Modules may request Map storage objects, or register Maps of currently referenced objects. The API maintains a handle on Map objects independently of the module, and they may be reclaimed by modules following a reload, if desired.

The API also includes a static method for constructing new WebCom instances. Third party applications wishing to use WebCom as an implementation architecture, can use this method to start and maintain a reference on a WebCom instance. Messages can be sent to this WebCom to perform application work.

The API class contains references to an InformationAPI and an ActionAPI object, respectively, the query and operation interfaces for the represented WebCom machine. The query interface provides access to machine data, whereas the operation interface provides methods to effect changes in machine state.

The InformationAPI provides a number of query methods, providing security authorisation, module addressing, module referencing, and some minor configuration details. It is intended to expand the InformationAPI as necessary to support desired query requests from third party modules. As such, this is an early specification, likely to evolve to meet developer requirements. The InformationAPI methods are implemented via direct calls to existing core module operations and module writers are advised to use the InformationAPI rather than internal methods, for robustness. Internal interfaces are subject to change and the indirection level provided by the InformationAPI is useful.

The ActionAPI provides operation methods, dealing with WebCom connections, module loading, and logging messages. Send messages is also done via the ActionAPI, as previously noted. As with the InformationAPI, the ActionAPI is intended to evolve to meet module writer requirements.

Together, the InformationAPI and the ActionAPI classes should provide all the facilities and information required by module writers. Their ability to meet this goal will be improved based on user feedback and feature requests. At present, many of the core modules do not employ the InformationAPI and ActionAPI classes and it is hoped to improve these these legacy implementations during their next refactoring.[8]

§42 Module Loading Mechanism. *≪Details on the module loading process≫*

The Module API includes a new system for loading and unloading modules, extending the older system to support callbacks during loading and to support module unloading. There are also extensions to the module configuration files to support automatic loading of third party or user modules.

The module lifecycle begins at loading. Modules may be loaded automatically when the WebCom is created, or loaded explicitly via use of the ActionAPI.loadModule call. To be loaded automatically, a module must be listed in the WebCom modules.properties file. This file contains lines listing the required core modules, those which must be present for the WebCom to run, and an optional listing of third party modules to load at runtime. Either load mechanism has the same result in the case of user modules.

Core modules are loaded in a slightly differently from user modules, mostly for legacy compatibility reasons. The only noticeable difference is that a set of core modules must be acquired and loaded *before* any third party modules are given an opportunity to load. The load procedure is pretty much the same for core and user modules, with some additional configuration and addressing matters in the case of core modules. Only third party module loading will be considered below.

To load a module, the named class is instantiated via a reflective `Class.forName` call. Any failures are logged and successfully returned `Object` references are cast to `Module`. The `API` reference field of this `Module` is then set to the current WebCom `API` and the module `load` method invoked to complete the loading.

Note that WebCom `API` objects are not available to module constructors. Hence, the module writer is given a chance to perform initialisation dependent on a `API` reference in the `load` method. Within this `load` method, it can be presumed that the API reference is set and can be acquired via the `getAPI` method.

Initialisation not requiring the API should be done from within the constructor. This includes regular application configuration, and WebCom specific actions like registering `EventAPIListener`s. API specific module initialisation needs to be done from within the `load` module method. The initialisation, API assignation and `load` method invocation order is the specification and is guaranteed.

Modules are unloaded by explicit calls to the `ActionAPI.unload` methods or at machine termination, and involve an `unload` method call on the module targeted. Unload operations are either soft, meaning a module programmer can can explicitly refuse to comply, or hard, meaning the module programmer's preference is ignored. Module compliance is indicated in the `unload` method return value.

§43 Note on Module Addressing. *«Remarks regarding current module addressing limitations»*
Currently module addressing is based on module classnames Specific modules can be requested from the `InformationAPI` indexed by desired classname. Module names are unique, being classnames, but there is a concern that multiple modules of the same core module type may be required. Core modules are not classname addressed for legacy reasons, instead being directly referenced by module type, e.g., SchedulerModule.

A problem arises if, say, two different connection managers are desired. This is not possible currently for addressing reasons. It is intended to tackle this problem via the use of composite modules when and if the circumstance arises. A composite module is a wrapper module facilitating multiple child modules of an identical type to be loaded. Composite modules would also handle addressing for these child module, perhaps by separate message headers in the `Message` type.[9]

Message routing addressing is similarly done depending on the class of the `Message` object received. The classname of the `Message` type associated with a particular module must that of the module with "`Message`" appended. So, `BlahModule` would have to be associated with message class `BlahModuleMessage` for the inbuilt API messaging to deliver messages.

Specific message classes are desirable because they force module writers to explicitly detail permitted message types. This enforced message specification is useful documentation. Of course, strictness is usually circumvented slightly by including a catch all message type `USER_MESSAGE` to facilitate later module extension and message augmentation. [10]

4.2 Example Modules Employing the Module API Architecture

4.2.1 Statistics Module

The statistics module was the first module to exploit the Event API architecture, and was written mostly to demonstrate Module API structures. In this respect, although strictly unnecessary, it utilises aspect based events, messages and the heavyweight module messages. The module code itself being fundamentally straightforward, is nevertheless overcomplicated by the demonstrative nature.

The statistics module began as instrumentation code in the original WebCom implementation, incrementing counters at specific points in the WebCom code. Although, this code was later removed, it formed an ideal application for the aspect based Event API and was reintroduced in a module form.

The statistics modules collects statistics generated from internal WebCom actions. These statistics are not especially interesting in themselves, including items such as the number of generated and executed instructions, the number of messages sent and received, and so forth.

While the statistic module's main use is as an example for module writers, it does also have real use within the WebCom applications. When loaded and enabled, the module collects statistics for provision to other modules, and for the IDE tool upon completion of graph executions.

§44 Module Operation. *«Run through of Statistic Module operation»*

The statistics module extends the statistics framework to support the exchange of statistical data with remote WebComs. Suppose client wants to acquire statistical information from a Statistics Module on a remote WebCom, for which it somehow has an `Address`. The client calls the `requestRemoteStats` on the local `StatsModule` and busy polls `getStats` for a returned result. Instead of busy polling, the client may instead be informed of result notifications by use of Module API heavyweight events. The client would need to implement the `StatsEventListener` interface and register with the `StatsModule`, in this case.

The `requestRemoteStats` method sends a `StatsMessage.REQUEST` message to the remote WebCom `StatsModule`. This module acquires the local `Stats` object, which maintains the current statistics for that machine, and extracts a Memento of the state.[11] This memento is returned, in a `StatsMessage.REPLY` message, to the original WebCom where a new `Stats` object is constructed from the memento details. This `Stats` object is incomplete, not having a handle to the right `API`, but suffices nevertheless.

The `StatsMessage.CLEAR` message can be used to reset the cumulative statistics on the remote WebCom. A `StatsMessage.CLEAR` is acknowledged by a `StatsMessage.REPLY` even though the memento indicates zero state. This saves on a separate `CLEAR_ACK` message to update the local WebCom state.

Note the statistics module is almost completely blind to WebCom internals, knowing only how to exchange WebCom messages. The statistics collection is done using the Event API `StatsEventAPIListener` class which is more or less ignorant of the WebCom internals.

Although clunky, this draft module is useful in illustrating the three major communication tools available to module writers, namely the message system, the event system, and aspect pointcut hooks.

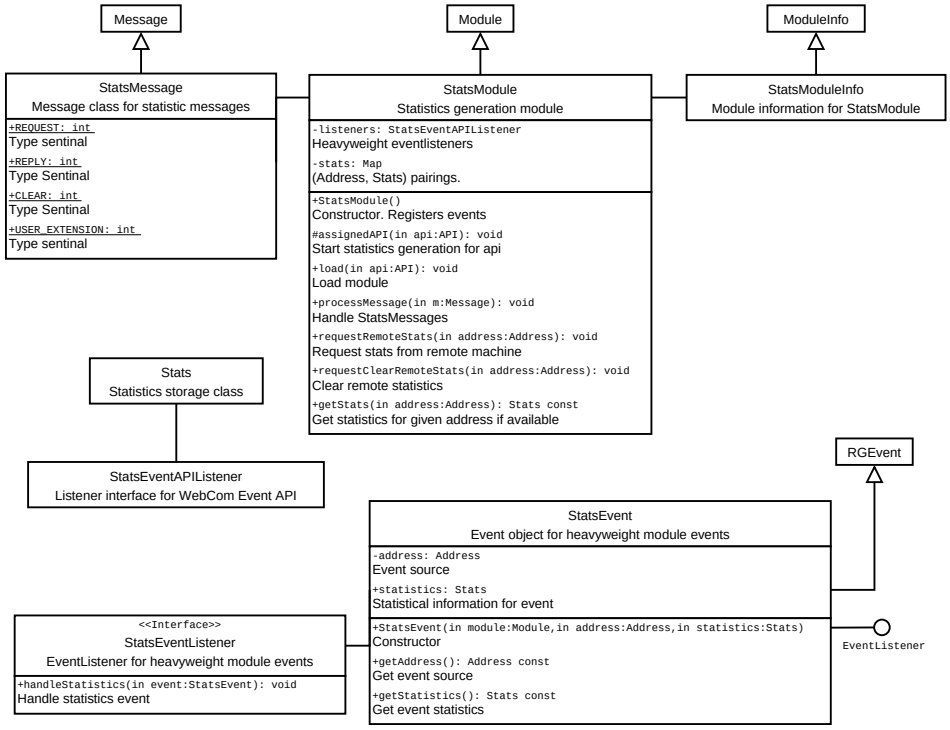

Figure 28: UML Diagram Statistics Module Part I.

§45 Implementation. *«Outline implementation of the statistics module»*

Figures 28 and 29 illustrate the `StatsModule` software. Most of the module framework constructions are detailed in Figure 28. There is an uninteresting `ModuleInfo` class for the `StatsModule`, and a standard `Message` subclass indicating the supported message types as seen in the earlier walkthrough:

- `StatsMessage.REQUEST` to request a remote machine forward statistics.
- `StatsMessage.REPLY` to return previously requested machine statistics, or to speculatively load remote caches with current statistical information. Also serves an acknowledgment role in the protocol.
- `StatsMessage.CLEAR` to request a remote machine reset its statistical information.

The `StatsEventListener` interface and `StatsEvent` class comprise a basic heavyweight module event setup. `StatsEvent` notifies interested parties that new statistical information has been received from a remote WebCom. So, clients can register as `StatsEventListener`s and not bother to busy poll the `getStats` call.

The other detailed class in Figure 28 is `StatsModule`, a pretty basic module even though it does exploit the three available communication techniques. It illustrates the convenience of the Module API that a fully featured module may be implemented with such little code. `StatsModule` simply registers the

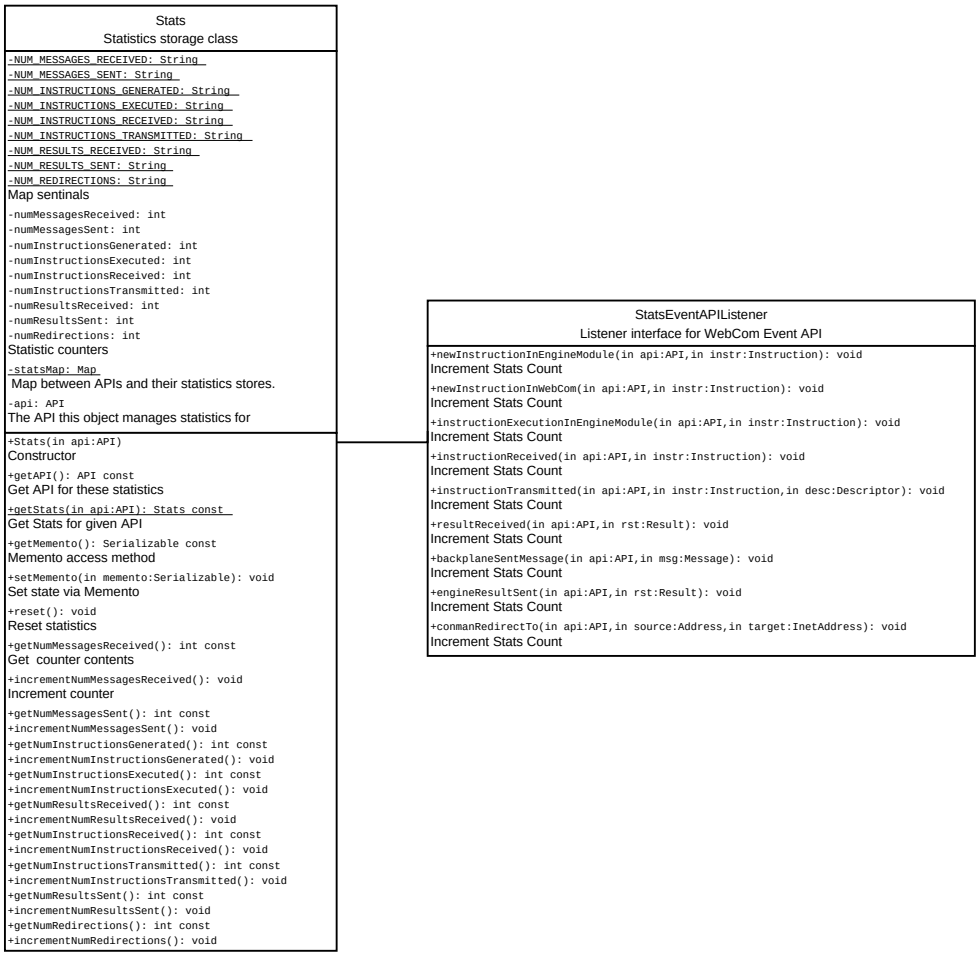

Figure 29: UML Diagram Statistics Module Part II.

`StatsEventAPIListener` and heavyweight events before just waiting for messages to arrive. These messages are processed according to the sketch protocol outlined above. All thats all, save for some sleight of hand regarding the incorporation of new `API` objects into the statistics scheme upon discovery.

Figure 29 details the software elements which perform the real statistics generation work. The `Stats` class is the main data store for statistical information, providing an `API` to `Stats` hashtable mapping, enabling the multi-`API` part of the problem to be compartmentalised. The `Stats` class also details statistics, counter increments, queries, resets and summary generations for an individual `API`. The `Stats` class also partakes in the Memento pattern externalising its state in an object safe manner.

The individual statistics in the `Stats` class are updated by `StatsEventAPIListener`, an Event API subclass. This class responds to the internal WebCom events corresponding to events of statistical interest,

and forwards increments to the relevant `Stats` object. So, when an event occurs in a particular `API`, this `API` is used to acquire the correct `Stats` object which is appropriately updated.

This module is perhaps the most complicated example here. The next examples are all of a trivial programming nature, yet achieve valuable software functions and interesting interface support. This ease and power of application confirms a key role for the Module API within future WebCom design.

§46 Future Extensions. *«Further directions for `StatsModule` development»*
There are many interesting directions in which the statistics module might be extended, the most immediate being expanding the range of statistics generated. In addition, the module would benefit from the inclusion of methods to perform basic statistical analyses. This might involve retaining a long run memory of statistical values, and using this history to estimate quantities, such as mean queue length, mean time to instruction, etc. This information would be of use in the scheduler and load balancer modules.

The limited interface to remote WebComs is a second glaring omission in the current statistics module. It would be advantageous to reduce message acknowledgment package sizes when not required, and to implement more finegrained reset and query operations on remote WebComs. These changes could be coupled with a more fully featured module interface providing statistical information on neighbourhoods in the WebCom connection tree. This would be of benefit to fault tolerance and load balancing decisions.

In respects, the statistics module is a primitive Information Manager Module, an in-progress plan to produce systems to manage and organise all kinds of WebCom metamachine data. The statistics module scratches the surface of one Information Manager feature, that of local statistics acquisition, and forms an Information Manager Module prototype case study, demonstrating the ease of statistics collection.

4.2.2 SysTray, IDE Bridge and Other GUI Modules

The idea of graphical modules is another interesting module development. These modules contain or present graphical elements for user interaction. The idea of an interactive module is the simple, but key development, since to date, modules have been user independent, requiring minimal configuration.

The IDE is the first immediate graphical component in consideration of graphical element modularisation, and there are streamlining advantages in implementing the IDE as a module. In the first place, it removes the current division between IDE GUI frontend and WebCom backend. Whilst, this division might be desirable, it is very much a traditional WebCom application operation, i.e., where WebCom is invoked separately to achieve some computation for the container application. This need not be the only interaction model.

Implementing the IDE in module form and attaching directly to a WebCom instance backplane might have benefits, including access to WebCom information via the Module API, rather than by message passing.[12] Additional modules may then independently extend the IDE using the Module API.

Moreover, the work in modularising the IDE is not significant, although there are careful choices to be weighted. There has been keen support for maintaining standalone IDE operation, and for ensuring a

decoupling of the IDE from any particular running WebCom instance is always possible. Methods addressing these concerns will be seen shortly.

§47 IDE Module and Bridge. *«Particulars regarding the IDE Module implementation»*
The use in an IDE module recasting is that rather than having the IDE start WebCom, it might be done the other way. WebCom could start an IDE. An advantage here is that the IDE could be started identically to WebCom save with a different configuration file. Also, other modules would be able to create an IDE for the user. So, for instance, a safety module tracking a graph execution might create an IDE on encountering type or security errors. The user could then edit the graph and resubmit it via the newly loaded IDE.

A particular problem with the current IDE is determining locked computations, and access to the Event API might help in establishing computation liveness. Presently there is little indication of WebCom progress when the IDE runs a graph. With the Event API, there is scope to provide visual cue updates on core events, making it possible for users to determine if a machine is stalled or just engaged in a long computation stream.

If a WebCom instance were always available during IDE operation, a graph could be started by simply sending a `TOP_LEVEL` message to the `Backplane` via an `ActionAPI.sendMessage` method invocation.

Implementing the IDE as a module does not[13] involve much effort and initially would not offer any module functionality. So, in particular, messages would be ignored. The `main` method of the IDE would be retained for launching the IDE, but would instead make a WebCom and load the IDE onto it as a module.[14]

Binding the IDE to the module architecture caused initial concern. However, implementation in module form does not preclude separate operation outside the WebCom context, and especially so in applications, like the IDE, which have low coupling with the WebCom core architecture. Implementing the module interface just means other loaded WebCom components can make use of it.

Writing code conforming to the Module API, does not necessarily bind that code to the WebCom core. In the IDE module case, there is fundamentally no coupling between the IDE and WebCom, other than preexisting work execution calls. The WebCom core is still unaware of the IDE aside from a result call direction. This is no different to the current configuration.

Given the reservations regarding tight module coupling, the IDE module was implemented as a loose bridge module instead of directly. Essentially, this involved leaving the existing IDE code in a standalone executable state, and implementing a wrapper module to invoke the IDE when loaded and destroy it when unloaded. This achieves the functionality of a direct IDE module, but at arms length, and while not optimal, is sufficient for developing and arranging GUI module coordination in an integrated tool.

§48 SysTray. *«Using modules to implement a WebCom desktop system as a loose confederation of collaborating applications»*
With this rework, it is possible to run WebCom as a desktop service and to dynamically load and unload modules via a system tray tool. This dynamic load feature enables a range of interesting user to WebCom interactions. For instance, say a graph execution goes awry, then the user might dynamically load a debugger or replayer module onto the running WebCom and debug the graph *in situ*.

Implementing the IDE using the Module interface directly, or via a helper module, means it could be easily loaded by this tray application. This WebCom SysTray tool need be little more than a WebCom instance wrapped with a GUI runtime module loader. In fact, the module loader and unloader features are not actually essential, since they may themselves be implemented in a module, albeit one which must be loaded at startup if the user wishes to change loaded module configurations.

The sum total of basic SysTray tool requirements in this design is to wrap a WebCom in generic GUI code for creating a SysTray icon and responding to menu clicks. This may not be extremely functional, but achieves the target of putting WebCom on the user desktop.

Clearly, additional features are required to make this SysTray application of any use. So, the SysTray requires a extension mechanism, but the Module API is a suitable mechanism already available within WebCom. Desired SysTray features may be implemented as WebCom modules and either loaded by the bootstrap configuration, which launches the SysTray-WebCom combination, or by the user as required.

In this way, lots of desirable design and programming advantages are leveraged directly from the Module API design right into the SysTray application. SysTray automatically benefits from component uniformity, high degrees of flexibility, short widget development time, robustness to WebCom modification and good isolated design. The tray application leverages the entire module architecture for ease of extensibility, and in fact, once implemented, the core SysTray icon code need not be reconsidered ever. Being so straightforward, it highly possible to implement it robustly and correctly the first time.

The problem of dynamically loading and unloading modules involves little more than an interface to loader methods in the `ActionAPI` and `InformationAPI` classes, to which all modules have access. So, any module may be used to implement this dynamic loading and unloading of modules, and in the SysTray case, it is convenient to implement this loader as a GUI module to facilitate user interaction.

Further, users could use the SysTray to directly load IDE instances, since just loading the IDE module bridge in the dynamic loader will kick start an IDE instance. This pattern generalises though, which makes it exciting. Log viewer modules, trace replayer modules, information and statistics viewer modules, debugger modules, etc. may all be implemented in this manner. Moreover, because they are written to the Module API interface, the modules require very little knowledge of SysTray internals.

A version of the SysTray application has been implemented according to this design, written using the SWT widget toolkit, and so should be available on any platform with an SWT implementation, including most popular desktop systems.[15] This current SysTray application includes graphical load and unload, IDE bridge, parent connection dialogue, and log viewer modules. These already provide quite a fully featured WebCom desktop environment, but would benefit from more module implementations, such as a debugger, fully featured statistics viewer, etc.

4.2.3 BeanShell

BeanShell is a Java scripting tool, providing a commandline interface programmable with Java syntax. BeanShell, like Jython, is a popular Java style scripting framework which may be embedded within user applications, providing developers with scripting access to objects within a running application.

WebCom already incorporates the BeanShell tool as support for certain scripting library nodes. Security notwithstanding, the use of BeanShell within WebCom could be beneficial. Developers could employ its versatile syntax to debugging task, or use it to extend WebCom to meet unforeseen and otherwise difficult to fix problems. Scripting could be used to "glue" existing WebCom software into more powerful tools.

But, debugging is the use of most primary interest, especially within the Module API context. A basic BeanShell bridge module provides some nice user interaction possibilities. Suppose a WebCom execution freezes, then a developer might load a BeanShellModule onto an otherwise frozen WebCom and use this interface to examine WebCom state, perhaps via available `InformationAPI` objects.

In addition, the BeanShell tool also comes with a graphical version. The implemented module bridge leverages this graphical BeanShell prompt into a WebCom module, available for use within all WebCom tools. Most especially, this BeanShell is available for use within the SysTray application, so forming another quick SysTray extension tool, and emphasising the small tool organisation of SysTray. SysTray users can load the `BeanShellModule`, via a convenience menu link, and examine internals of the running WebCom. So, with little expenditure of effort, the WebCom desktop tool already has an available console tool.

BeanShell may promote a more serious WebCom scripting system, the basics of which are already in place, being the Module API and BeanShell support. The provision of a medium sized library of useful WebCom BeanShell scripts might be helpful to developers. Scripts could be bundled to dump WebCom state, clear instruction queues, launch graphs, etc. and would complement the BeanShell application significantly.

Finally, note that the BeanShell interface may also be used remotely, and could facilitate a remote WebCom debugging shell. However, there are serious security concerns to address first.

4.2.4 Future Directions

Before leaving the topic of the Module API, it is instructive to consider some future work and applications in the area of module design. Only some brief sketch details of some proposals will be mentioned here.

§49 Core WebCom Modules. *≪Use of the Module API within the WebCom core architecture≫*
The first concern is a matter of WebCom internals. Within the WebCom core design, there is much functionality which might be componentised using the Module API. Aside from the previously discussed core modules, there are other classes working to achieve particular purposes within the WebCom machine. Isolating and factoring these roles would add additional configurability to a more decoupled internal architecture.

For example, within the WebCom core are classes dedicated to facilitating the exchange of `Class` items between peer WebCom machines. This is arranged in a server-client architecture and intended to support cases where one WebCom sends another WebCom work which that WebCom cannot complete. A common

failure in completing work is an absence of critical classes from the peer WebCom. To this end, the class server and client structure was designed to deliver `Class` descriptions to WebCom JVMs requiring them.

This functionality could be easily factored as a separate module, and would allow site administrators fine-grained levels of configuration options in deciding whether to permit such class sharing. Further, factorising such reflection dependent code helps in the task of porting WebCom to embedded architectures, which may not possess these reflection capabilities. In this case, the reflection dependent code can just be dropped.

The class loader is not the only example of modularisable code within the WebCom code. Others include, maintaining logging support, and using `WorkReceiverModule`s as *ad hoc* connection managers.

§50 API Module. *«Sketch of API exchange module»*

Without considering security, it is interesting to consider facilitating `API` object calls on remote WebCom objects. For instance, a local WebCom might maintain a reference to a `RemoteAPI` object, subclassed from `API`, where methods are implemented to involve transferring requests to a remote WebCom before execution.

The implementation of this scheme might involving backing the `RemoteAPI` class with the use of an `APIModule`, and using the message passing system to exchange call details. Naturally, the method call and parameters would have to be marshelled properly within an `APIModuleMessage` class, and concern would have to be directed toward handling instances of missing local references.[16]

Missing local references in the parameters is less of a problem on the calling side, as certain strict but manageable restrictions would avoid this problem arising. The issue of missing references is a bigger problem for returned results. For the most part, `API` query calls return primitives, but some instances require special measures, such as with local socket referencing. These measures may involve separating the `API` interfaces into remote-safe and local-safe categories.

It is worth noting the implementation strategy employed in this design, since it forms a very common module application pattern. Some available software object, in this case the `RemoteAPI`, is implemented to delegate its methods to a backing module. This backing module then performs necessary operations, perhaps involving the Module API, to achieve certain goals and results which are relayed to the original method.

§51 API Extension Support. *«Augmentation of module interfaces»*

It is essential at some future point for the API classes to provide extension points, in order that user modules may extend the API with new functionality. This models the Eclipse scenario where modules or plugins have exposed extension points and it is necessary to emulate this within the WebCom API design also.

Consider the case of the statistics module, where it might be useful to expose the operations for requesting remote statistics, for instance. However, there is presently no way for the statistics module to advertise this functionality to other modules. Currently, the best that can be done is for modules to request a list of loaded modules from the `InformationAPI` and to individually examine the class interface of each in turn. This does not indicate any especially helpful metadata to the querying classes, however, and really ought to be replaced by a superior scheme.

In fact, a COM-style interface support might be a step in the right direction, provided this interface clearly exposes and documents offered module methods. Similarly, this interface should also offer the particular extension points within the module to outside parties. Thus, to extend the core API or the API of a particular module, the developer implements a certain interface and applies it at the exposed extension point.

Chapter Notes

[1]In particular objects comprising the Backplane were suitable to represent a single WebCom instance.

[2]This addressing, introduced as a workaround, is suboptimal. However, there hasn't been pressing need yet for an update.

[3]This could also be called a modular architecture, but within WebCom the module terminology already has specific meaning. Hence, the term plugin is used to discuss WebCom components even if these components are likely to be actual WebCom modules.

[4]Although the modules are called noncore, this does not mean that functions provided are not core, in some sense. The term core refers to the very basic code, necessary for WebCom to achieve graph reductions and peer collaboration.

[5]A developing `ModuleInfo` specification deals with module dependencies. The current draft includes dependency notations, but not automatic dependency resolution, or dependency cycle avoidance structures.

[6]If the module already contains the suffix `Module`, this can be excised before consideration in this naming scheme. So, the `ModuleInfo` for `SecurityModule` can be found in `SecurityModuleInfo` instead of `SecurityModuleModuleInfo`.

[7]This restriction is easily remedied, if so necessary.

[8]This update must consider which core module methods are used to implement the methods in the `InformationAPI` and `ActionAPI` classes, of course. But otherwise, core modules should employ API classes to the fullest extent.

[9]If a composite module is ever written then it will likely dictate the addressing specification in cases of multiple core modules.

[10]It used to be the case that some of the modules sent plain `Message` objects as messages. This was refactored because it obscured the content of the messages. The additional tagging is also helpful in debugging.

[11]Yes. The design could just make `Stats` serialisable, but the Memento pattern is sharper OO, maintaining encapsulation.

[12]The IDE talks to a child WebCom instance by forwarding a work message to it, and registering for an result notification. The WebCom is otherwise an entirely separate entity from the IDE.

[13]Did not, rather. A module version of the IDE was prepared as a prototype before the final bridge module architecture was decided on. The programming effort involved in producing this prototype was minimal.

[14]Actually it would have to load a new IDE onto the WebCom, but doing this first causes no harm.

[15]It isn't possible to have a completely portable system tray application in Java. The "Write Once, Run Anywhere" Java motto ends upon contact with the modern graphical user interface. As such, the SysTray tool is only fully supported on the Windows platform.

[16]A first approach requires `API` methods parameters to be serialisable. Although, this works, more refined solutions are possible.

5

Logic Programming in WebCom

In addition to incorporating Aspect Oriented Programming techniques, producing type checking capabilities in WebCom also involves introducing elements of the Logic Programming paradigm into WebCom. There is a convenient representation of the type checking primitive actions in terms of logic expression resolution. For this reason, and because it has practical applications in other WebCom development, a resolution engine was incorporated into the WebCom code base.

This chapter examines this resolution engine and its implementation, including the formulation of basic type checking operations in first order predicate logic. This logic engine forms the final component needed to implement type checking support within WebCom, the topic of the following and final chapter.

Implementing a logic engine from scratch is not excessively difficult, being almost entirely an engineering task. And, while existing Prolog engines could easily have been incorporated, a new resolution engine implementation provided opportunities to finesse the software artifact design and to simplify its adoption within third party WebCom applications. That is, a template structure could be designed supporting straightforward access to logical data and operations thereon from within third party modules, or elsewhere within WebCom.

This logic programming harness primarily facilitates basic type checker validation calculations. However, with general application the resolver supports the theme of extending current WebCom programming methodologies. It complements available OO and AOP with possibilities for logic programming design.

There is also no reason why resolver facilities need not be available to graph designers, via logic computation nodes. Although, outside the scope of the work here, it is possible to conceive of a set of WebCom

operations enabling a graph designer to describe, parse and resolve graph level logic programs.

In any case, logic programming features are a valuable addition to the WebCom software, as will be seen in the implementation work for the type checker modules in the next chapter.

5.1 Resolver

The resolver engine is key in implementing logic programming within WebCom, and involves producing software objects to represent logic data elements, and to implement unification and resolution algorithms.

§52 Resolver Components. *≪Description of the portions of the Resolver architecture, and the approach to implementation≫*
There is particular concern in the correct logic data element design, i.e., the representation of logical structures like atoms, literals, clauses, and functions. Developers should be presented with terms appropriate to their application. For instance, in the type checker design, it is more convenient to deal with elements like types, conjunctions of types, disjunctions of types, as well as a subtype inclusion relation. The developer will recognise helpfully named dual software elements such as `Subtype` rather than unqualified mathematical terms like literal, clause, etc. This allows developers to ignore logic programming somewhat and instead represent relations between real artifacts.[1]

Aside from logic element representation, there are the two key algorithms required, unification and resolution. The implementation of these algorithms within extensible frameworks, particularly in the case of resolution, is helpful for later extension. These customisations do not typically concern logic programming endusers, unless they wish to apply extra domain knowledge to improve unification and resolution efficiency.

5.1.1 Logic Elements

Describing the logic engine must begin with basic object representation, a UML sketch of which is shown in Figure 30. These elements will be discussed in turn below. Knowledge of First Order Predicate Logic(FOPL) is helpful, but is not the real point here, so rather than discuss interpretations too much, the logic elements will be presented operationally. Just presume the set of items upon which the logic will work. What these elements are does not matter,[2] just that they are given.

§53 Basic Logic Representation. *≪The classes describing the logic elements required for FOPL implementation≫*
It is easiest to begin with the `Constant` class which represents logical constants in the FOPL system. These are the nouns of the modeled system, and are little more than identified items, a fact mirrored in the essentially blank programming implementations thereof, consisting of a constructor and a visitor callback only. An example constant might be a specific type in the type checker system, for instance.

Logic constants are special cases of logic functions. A function is an identifier with a set number of subterms, which maps these subterms to a specific item in the modeled system. Constants are functions with no subterms, so always mapping to the same item in the modeled system, i.e., constants are nullary functions.

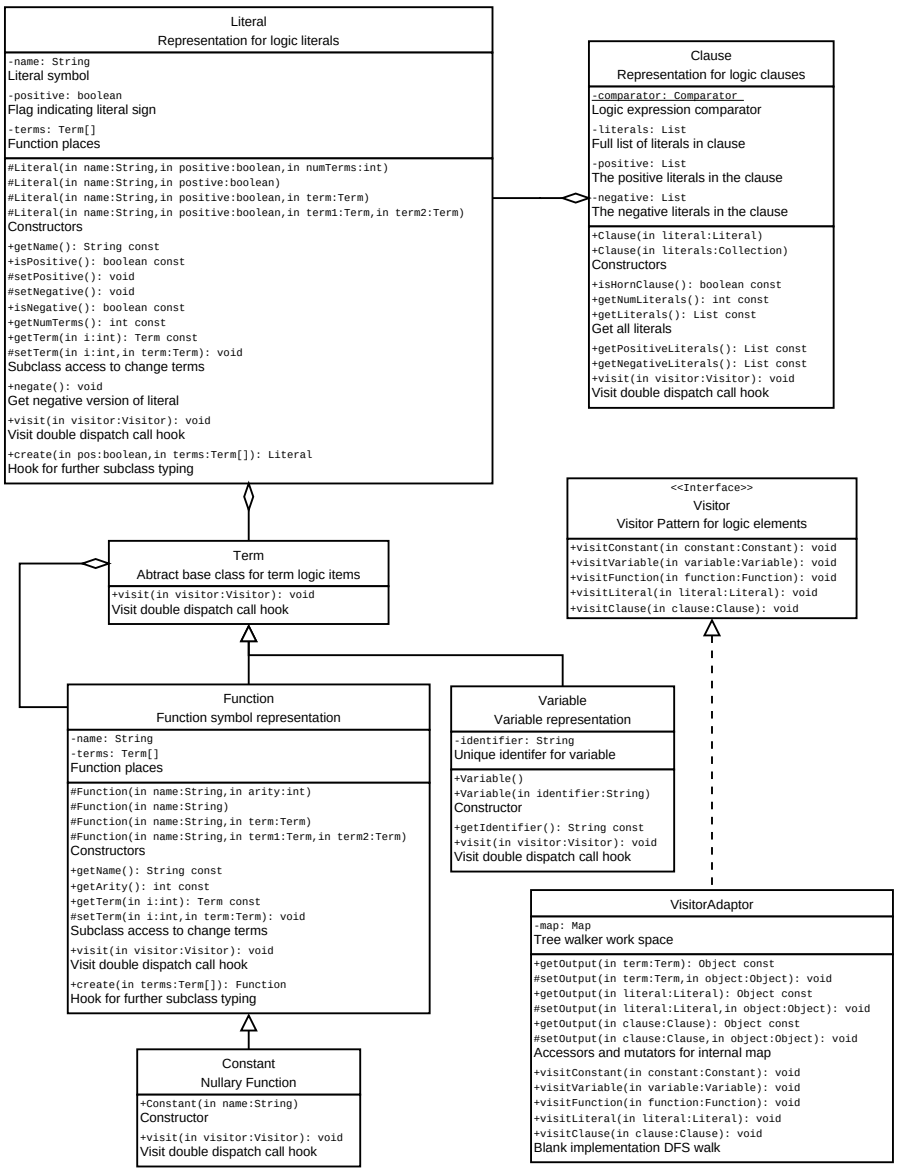

Figure 30: UML Diagram Logic Datastructures.

An example type system functions might be a function to combine types into aggregate union types, or to represent all but a specific argument type.

Functions are represented by the `Function` class, and consist of a function name identifier, together with a list of subterms underneath the function symbol. The `Function` class includes operations to manipulate these subterms, to implement double dispatch callback, and to facilitate more specific typing in subclasses.

Subterm items, the elements that functions operate on, are just variables and other functions. A logic variable can take the place of any definite logic term expression. Within the software, variables are represented by the `Variable` class, and are simple placeholders with identifier names.

Term objects themselves are not realised directly within the system, only variables or concrete functions are actually used. However, the term entity has placeholder functions, represented by the `Term` software counterpart, a blank abstract base class with just visitor callback specification.

The `Term` composite tree forms the fundamental logic element. The two remaining logic elements, literals and clauses, are compositions of `Term`s and `Literal`s respectively. The `Literal` class represents logic literals, either positive or negative predicates. A predicate, like a function, consists of an identifier and a set number of subterms. But unlike functions, predicates map the subterms to a Boolean representing the logical validity of the predicate.[3] An example type checking literal might be a subtype containment relationship.

The final logic type is also little more than a basic container object, albeit one with an important role in unification and resolution. A clause is a group of literals, logically conjoined implicitly, so mapping into a natural Boolean valuation. The simple data `Clause` class represents clauses.

§54 Logic Class Use. *«Logic representation use in custom applications, and considerations of the Visitor pattern»*
These logic classes are intended to be subclassed in practice. In particular, an application designer using the WebCom logic programming facilities must first decide on their desired logic. And given, the constants, functions and predicates of this logic,[4] they subclass corresponding logic representation classes to match.

For example, take the typing logic, described in detail later in this chapter. While naming constants in this case is not overly helpful, it is helpful to subclass `Function` with the particular functions including set conjunctive, disjunctive, and negation operations. These are respectively subclassed in `And`, `Or`, and `Not`.

Set conjunctive, disjunctive, and negation should not really be presented in logical terminology as they are here. There are reasons, doing with how developers read and interpret type strings, for using the class names `And`, `Or`, and `Not`. But this doesn't relate to the actual set based typing semantics. Consequently, this use of logic terminology for set operations may be confusing. Simply thinking of set conjunction, disjunction, and negation, as set intersection, union, and complement, respectively will fix any confusion.

Aside from source code clarity, there is a parsing value in using subclasses to represent logic elements. Automatic parser generation tools can produce parsers to generate and populate expression trees consisting of specialised subclass logic items. Specialised subclasses explicitly document the mapping between the data representations used in automatic parsing and those used elsewhere in the software application.

Before concluding with logic representation, it is worth noting the Visitor pattern for the logic representation classes. As in the earlier software designs, a Visitor pattern is available to developers. In this case, the

default Visitor adaptor incorporates an automatic depth first search through the logic expression tree. This adaptor also provides a map attribute field, into which adaptor implementations may deposit objects on a per term basic. So while walking an expression tree, adaptors can easily build up complicated result objects.

The main Visitor pattern application is as a logic expression comparison tester. This `LogicComparator` class is an important within the unifier and resolution algorithms and is implemented by depth first search in the adaptor. This testing is for deep equality but uses object reference equality for primitive logic elements, meaning especially that variables are not equal if they are distinct objects. This is bad since actual variables themselves are not important, but rather their structure and arrangement within an expression tree.

However, testing for equality modulo variable instances can be done via the unifier. That is, two clauses are tested by unifying them and checking if the unifier substitutions just match variables to other variables. This corresponds to an ignorable simple variable relabeling.

5.1.2 Unifier Code

Unification is a key element in any resolver, and a description thereof is essential in discussing resolution or logic programing. This section, included for completeness, will briefly cover the basic unification algorithm.

Unification, of course, is a algorithm for matching two FOPL expressions, providing the substitutions, or unifier, which result in the same expression when applied to both expressions. Unifiers do not always exist for FOPL expression, but if one does exist, then there is a unique most general unifier for those expression.

§55 Unification Algorithm. *≪Brief review of the unification algorithm≫*
The unification algorithm, takes two FOPL literals, l and m, and returns their most general unifier, if the expressions are unifiable. Unification works by considering corresponding terms from l and m in turn, and matching them according to some basic unification rules. Suppose the algorithm is unifying terms t and s:

- If t is a variable, the substitution $t \rightarrow s$ is first applied to the current most general unifier, then added to it. If instead, s is a variable, then the substitution $s \rightarrow t$ is used.
- Else, if s and t have the same function symbol and arity, then the subterms are added to the current goal list for the unifier. So, $t_1 = s_1$, $t_2 = s_2$, ..., also need to be unified for the unification of s and t to succeed. The substitutions required for these unifications are combined in the most general unifier. Note this also includes the case of constants, which are just nullary functions. So for constants to match, they must have the same symbol, i.e., be the same constant.

A comprehensive outline of the pseudocode for this algorithm may be found in Appendix C.

§56 Unification implementation. *≪Brief notes on the software implementation of the unification algorithm≫*
Figure 31 illustrates the unifier software UML. The approach is deliberate overkill in order to support the use of optional occurs checking and custom unifier software.

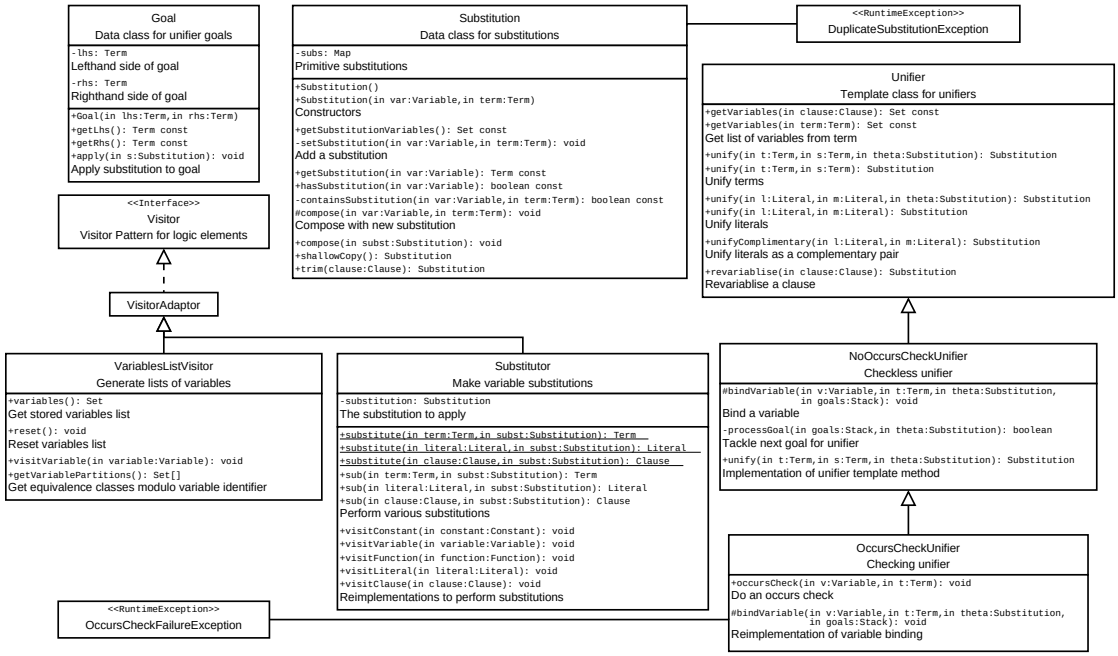

Figure 31: UML Diagram Unifier Code.

The user instantiates whichever concrete Unifier subclass preferred, and invokes the `unify` method with the `Literals` to unify. A `Substitution` object is returned containing the most general unifier substitutions, or `null` if the unification failed. Implementation-wise, the unification arrangement is handled by the `Unifier` base class, with term unification being done by a templated subclass method. `NoOccursCheckUnifier` implements a no-frills unification which is extended with occurs checking in the `OccursCheckUnifier`.

Note that applying a substitution makes use of the earlier discussed Visitor pattern. Also worth noting is that the resolver architecture, described below, is independent of unifier choice. So, for instance, a choice to use occurs checking in the resolver may easily be changed.

5.1.3 Resolver Engine

Resolution is a process for determining whether a certain clausal logic expression may be derived from a base set of other clausal logic expressions. The mechanics of resolution require negating the query clause and identifying a logical contradiction, thus proving the original query. Resolution proceeds by resolving complementary clauses to produce smaller clauses, and iterating toward the empty clause which denotes a desired contradiction. Resolution is a search problem which many clause combination strategies.

Logic programming is organised by the resolver engine, building on unifier and logic representations, to arrange the specification and execution of rule based logic programs, by resolution tree search.

With this engine, WebCom and third party developers can encode logic programs as software objects and query the inbuilt resolver to compute logic facts. So, with a handcrafted or parsed logic program object, the developer may query whether certain facts are derivable from this program or not.

The resolution engine architecture is designed to allow the incorporation of different resolution strategies. There is scope for the specification of candidate resolvents at each step of the resolution tree, which may be used to implement resolution strategies solely dependent on resolvent selections, e.g., input resolution, SLD resolution, etc. Also possible are resolvers which incorporate historical search information into next resolvent selections. However, local selection of next resolvents is the only mechanism for effecting the resolution algorithm, although this local knowledge may feedforward to later selections.

The discussion of resolver architecture begins by considering a limitation of the implementation before moving to the examination of logic program representations, and the identification of candidate resolvents or complementary clauses. The base resolution itself will then be covered, including an implementation of SLD resolution, before concluding by looking at resolution tree state management.

§57 WebCom Resolver Limitation. *≪The restriction of resolution queries to single literals, and overcoming this drawback≫*
The actual resolver implementation is limited to single logic literal fact checks. It cannot check the validity of arbitrary clauses, since these may have more than one literal. This single literal restriction is both an implementation convenience and runtime efficiency, since multiliteral clauses may give rise to multiple clause query negations which would make resolution less efficient.

Whether this is a severe restriction is debatable. With single literal queries, all the predicate relations in the logic may be directly tested. Practical applications of the logic engine typically involve the construction of reduction style rule bases, or rewrite productions, and user applications are typically only interested in knowing definitive information, i.e., testing a single predicate with particular values.

However, this restriction does mean predicate implications cannot be directly tested. So, it is not immediately possible to test if a particular predicate value implies a certain other predicate value. This concern can be handled in many ways, the simplest being a hypothesis based approach. That is, the antecedent expression is assumed in the rule base and the consequent fact is tested as a single literal query. Alternatively, propositional calculus rules may be coded within the FOPL logic in use. So, for instance, predicates for logical conjunction, disjunction, negation, and implication are added,[5] together with appropriate derivation rules.[6]

§58 Logic Program Representation. *≪Rule bases representations≫*
For the purposes of resolution, it must be convenient to deal with sets of `Clause` objects, the largest logical expression units within the software. To this end, the software contains a basic collection type for clause objects, the `Clauses` class. The use of clause aggregation for logic program representation is examined here.

A logic program may be represented as a set of clauses, an within the WebCom resolver architecture, there is no special need for the logic program software representation to be anything other than a list of clauses forming the backing rules or knowledge base. The triggering of actual computation is left the client code, which issues single literal queries against the backing rule base.

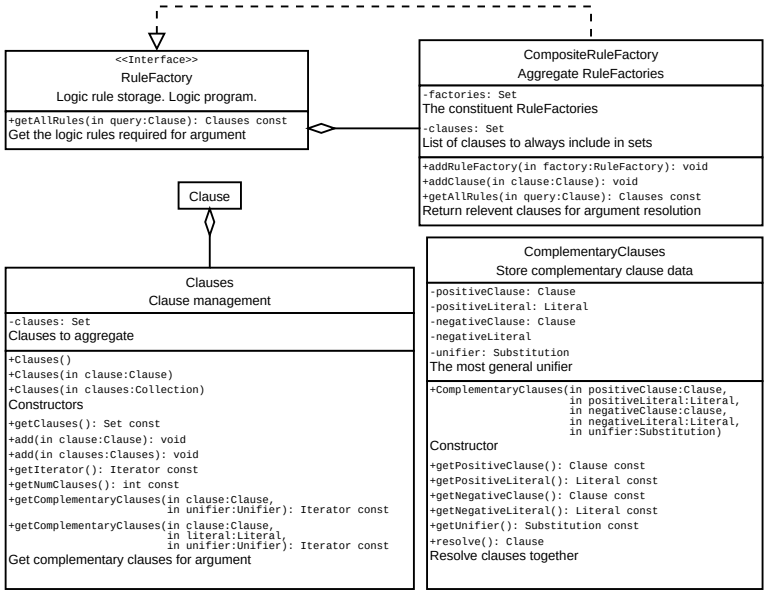

Figure 32: UML Diagram Logic Program Representation.

Logic programs may be represented as `Clauses` objects, this clause aggregation playing a fundamental container role. However, it is far more convenient to represent rule bases encapsulated within objects, which must then be asked for clauses relevant to a particular rule base query. Instead of a list of clauses, program representation is by factory objects returning lists of clauses particular given query resolutions.

Such a result list may, in fact, be a full list of rule base clauses, and as such, the relevance query view also encompasses the basic list view of logic rule bases. The two views do not coincide, since asking for relevant clauses introduces additional possibilities in terms of rule schemata and large rule base representation.

This view is dictated by the `RuleFactory` interface, consisting of the single `getAllRules(Clause)` method for establishing clauses necessary for a query. The existing library includes a concrete implementation of this interface, the `CompositeRuleFactory` class. This is a simple container for `Clauses` objects, facilitating both the basic list logic program representation, and a composition facility.

In addition to rule lists and single rule representations, `CompositeRuleFactory` facilitates the composition of `RuleFactory` objects. The `CompositeRuleFactory` contains fields for specifying rule lists always required for queries, and so always returned by the `getAllRules` method. But also included are fields for `RuleFactory` objects. These subfactories are queried for required rules as part of any `getAllRules` calls, and thus filter up required rules as per standard Composite pattern application.

§59 Complementary Clause Identification. *≪A consideration of complementary clause identification≫*
Turning to the question of complementary clause generation, a task performed almost exclusively within the

`Clauses` class. This operation involves a scan of the full clauses list, at each step of which is nested a two list complete pairwise element examination, and so at very least, comprising a somewhat quadratic order cost.[7] There is scope for `Clauses` reimplementation to optimise complementary clause identification.

A complementary clause pair is a pair of clauses sharing a unifiable literal, positive in one of the clauses, negative in the order. Resolution, the critical part of the resolution process, is the elimination of this literal. The establishment of these resolution candidates is exactly the complementary clause identification task.

In terms of resolution operation, it is best to check a particular clause against all clauses in a `Clauses` object, effectively a chunk of the fact base. This check should determine all complementary pairs involving the test clause and any clause from the `Clauses` aggregation. To do this, each test clause literal is considered in turn against each literal from each clause within the `Clauses` object. On determining a potential complementary literal, i.e., one of the same function symbol and opposite sign, a unification is attempted to reconcile the subterms. A successful unification indicates a complementary pair.

Complementary pairs are represented by `ComplementaryPair` objects, data classes maintaining the positive and negative literals and their containing clauses. A reference to the unifier substitution is also kept, since this unifier is applied on the resolvents during the resolution process.

It is critical for the resolver code to implement some indexing, hashing, or structured data storage to improve on this naïve complementary pair generation scheme. Any indexing, by literal symbol for instance, is a one time cost which is easily repaid since clauses and literals are repeatedly examined during the checking process. Since each clause in the rule base for a particular query might potentially be examined at every resolution step, the complementary clause lookup should be heavily optimised.

§60 Resolution. *«The resolution algorithm boilerplate»*

The resolver algorithm process begins with a particular clause, one of the available initial clauses in the rule base, or perhaps the negated query clause. A step is made in the resolution process by resolving a complementary pair referencing the current clause, so eliminating a literal, but also potentially introducing a number of new literals. The goal is to resolve out the entire clause.

Because there is choice in the initial clause selection, and in the complementary pair selection at each step, there is a tree of possible resolution paths. The job of the resolver is to search this resolution tree for a path producing a contradiction or empty clause, thus validating the query clause.

In the basic approach, any complementary pair may be resolved at any time to extend the set of clauses available for forming complementary pairs. In practice, such unrestricted free search is hopelessly inefficient, giving rise to slimmed search strategies of various effectiveness. These strategies typically reduce the choice available at each resolution step, thus pruning the search tree. Strategies vary in the amount of choice they permit, and in the expressiveness of the queries and rules bases on which they are accurate.

The basic common selection reductions might involve requiring half of any complementary pair to be the negated query or a descendant, or requiring half of any complementary pair chosen to be in the initial set of clauses. Other search pruning include matching on the negated literals in Horn clauses only, for instance.

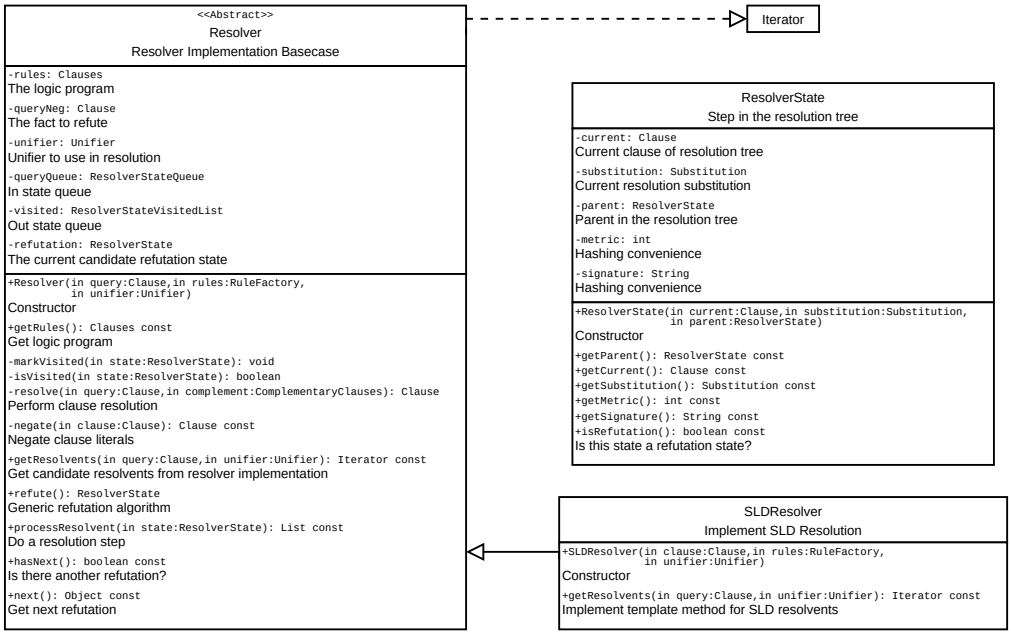

Figure 33: UML Diagram Logic Resolution.

Within the WebCom resolver engine, naïve search is implemented, together with an option to specify the set of permitted complementary pairs. In this way, different search minimization strategies may be expressed in a uniform architecture. So, although developers may apply optimizations in terms of search minimization, they cannot apply other efficiencies particular to the search implementation. This is not a handicap, in that effective optimization of search bound tasks primarily involves minimising the search tree.

The resolver search implementation is orchestrated from the `Resolver` class, or from a concrete realisation of this class. Resolver state consists of the current clause to resolve, a substitution involved in getting to this clause, and a link to the parent state. Initially, resolver states representing potential start states, namely negated query clauses with null substitutions and parents, are pushed on a queue.

At each step of the resolution process, the next state is popped from this state queue and examined. If this state is a contradiction, i.e., an empty clause, it represents a refutation of the negated query, and thus an establishment of the required query. The substitution producing this goal state is then reported to the original client, whereupon further refutations may or may not be requested.

If the state is not a refutation, a list of potential resolvents is collected by querying the concrete `Resolver` class implementation for permitted resolvents in the current state. This is where the designer has discretion to curtail the search, an example of which will be illustrated below.

Given a list of permitted resolvents, each resolvent is resolved in turn against the current clause, producing a potential next state. This next state consists of the resolvent clause, the current substitution composed with

the resolvent unifier, and the current state, now taking the role of parent state. Each next state is pushed onto the resolver state queue. To complete the resolver step, the current state is marked visited and prohibited from entering the state queue again. The next resolver state is then popped from the state queue for processing.

A long search is an unavoidable possibility, and representative of general AI search difficulties. However termination may result from an early isolated refutation, rather than ultimate search tree exhaustion.[8]

Note the `Resolver` class may be customized with the user's choice of unifier. So, either of the unifiers from previous discussion, or any custom unifier implementation, may be used in the resolution process unifications. With careful specification, it may even be possible to mix and match unifiers if required. In this case, the efficient `NoOccursCheckUnifier` may be used for common unification tasks where speed is essential, saving the more inefficient `OccursCheckUnifier` for less common verification use perhaps.

§61 SLD Resolution. *«Outline of default resolution strategy and example `Resolver` implementations»*

As mentioned, different resolution strategies are made possible by implementations of the `Resolver` class template method. One such implementation, the default resolver for type checking applications, is in the `SLDResolver` class, an implementation of Selected Linear Definite (SLD) resolution.

In SLD resolution, the permitted resolvents are those involving negative literals from the query clause. That is, one half of each resolvent pair must be a negative literal in the query clause. This significantly prunes the search tree, especially if the query clause is small and has few negative literals.

The SLD resolution implementation requires the `SLDResolver` class to extend `Resolver`. The method of interest is the concrete realisation of `getResolvents`, one of only two methods in the `SLDResolver` class due to extensive templating, the other method being a constructor. The `getResolvents` method extracts negative literals from the query, or current clause, and returns any complementary pairs in the rule base.

SLD resolution is a tradeoff between efficiency and application, in that large portions of the search tree are pruned but the resolution is only guaranteed to return the correct result is used on Horne clauses. This is not a severe restriction in reality, and SLD resolution is a very common resolution strategy. For type checking purposes , the restriction to Horne clauses is met, and SLD resolution may be satisfactorily applied.

This implementation of SLD resolution is straightforward, as are most other common resolution strategies. These other resolution strategies involve similarly small `getResolvent` methods, describing the particular resolvent sets the strategy in question permits.

§62 State List Management. *«Notes regarding the management of state queues and visited lists»*

Before leaving the discussion of resolver architecture implementation and beginning to consider example logics, it is examining state management structures in the resolver setup. Resolver state is managed in the `ResolverState` object, containing fields for the earlier outlined constituent resolver state elements, namely the current clause, the substitution to date and the parent state.

`ResolverState` objects, once constructed, are always pushed onto the `ResolverStateQueue`, a little disguised FIFO queue class masquerading with a fancy name. Once `ResolverState` objects have been dequeued and processed, they are passed to a second management structure, the `ResolverStateVisitedList`.

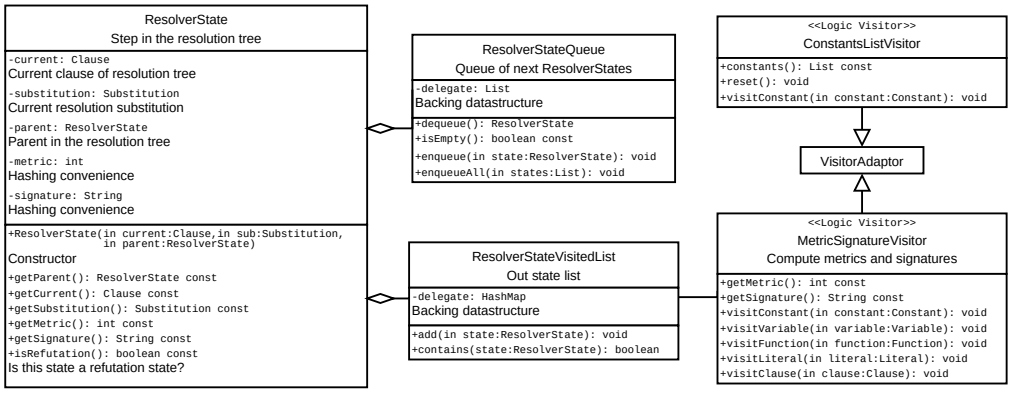

Figure 34: UML Diagram State Management.

This is a collection of all `ResolverState`s visited to date in the resolution, maintained to prevent revisiting. At its most basic, this is a simple map type, with methods to insert new states to check state membership.

Fast insertion and lookup is required in the visited list datastructures, and in this case a hashing implementation is employed. There are some serious drawbacks in hashing `ResolverState` objects, the most pressing being in terms of memory consumption. A `ResolverState` might unfortunately include a substantial quantity of information, and given that large quantities of such states might be expected during a refutation search, there are particular requirements to implement a memory optimised hashing scheme.

It turns out that just storing the clause element is sufficient since the visited list is only used to determine circularity in the search tree and is never required to produce actual `ResolverState` objects. Any required state objects are always to hand when needed. For instance, at a refutation point, the current state contains enough references to track the full refutation.

Furthermore, the actual logic objects are not required either. It is sufficient to produce a unique text signature for each state clause and containing all required information. Then, to test membership in the visited list, it suffices to compute the signature of the query state and use that as index in the hashtable.

Unique signatures may be computed by walking the logic expression tree and encoding the structure symbols in abbreviated form. Computation of signatures is done by the `MetricSignatureVisitor`, a logic Visitor pattern implementation. The signature itself is formed by marking logic expression elements in prefix walk order, and encoding the types as characters augmented with specific data. So, for instance, a function `fun` is replaced by the string “f(fun)”, a constant `const` by “c(const)” and so forth. Such string encodings are more memory efficient than their counterpart objects, even if slightly longer than absolutely necessary.

Note that variables are represented by “v(var_id)” where `var_id` is an integer identifier attributed to the variable by the signature generation process. In this way, variables which share the same identifier are the same for signature and hashing purposes. There is, however, the concern of variable substitutions and

relabellings appearing to be different. It is debatable whether an expression with a relabeled variable is the same as the original expression, especially in the context of a collected substitution.

This is managed in signature generation as follows. Identifiers are assigned incrementally to variables as they are seen in the visit order. The first variable encountered in visit order receives the identifier 1, the next receives 2, and so forth. If a variable is reencountered during the walk, it receives the identifier originally assigned. So, the local structure of variable placements is retained, even if the precise variable instances are lost. This means two logic expression signatures computed separately will give the same result if the expressions are equal modulo variable instance or naming.[9]

In any case, the current implementation depends on signatures for optimisation gain. There is plenty of garbage reclaim potential during the resolution process. It would be better not to generate the garbage in the first place, but most state data may be reclaimed with the state is pushed onto the out queue, provided none of its child states are active. Certainly, a state may be reclaimed on a post order traversal move.

5.2 Example: Typing Language

This section is concerned with the most important example logic, describing a type value expression logic. This is formulated in terms of type sets and constitutes the type checking application basis. The WebCom resolver support was initially designed specifically to express these logical type structures.

Basically, the logic expresses possible types as sets of Java classnames from a presumed universal set of all classnames.[10] Furthermore, classname constants may be combined into sets by use of intersection, union and universal set complement. Together these elements describe the possible types which may be associated to operator inputs and outputs. There is a single predicate, that of the subtype inclusion relation, really a subset inclusion relation. This is fundamentally everything required to validate graph type correctness.

§63 Formal Logic. *≪A strict definition of the type logic≫*

A more formal description of this basic type logic operation will be presented here. Begin by defining the logic $\mathit{Types} = (\mathcal{T}, \Omega, \Pi)$ where $\mathcal{T}$ is the set of possible type values. The set of all valid Java classnames only forms a strict subset of $\mathcal{T}$. A proper definition of $\mathcal{T}$ requires an inductive definition, incorporating some formal symboling business. So, define $\mathcal{T}$ as follows:

- Any Java classname is an element of $\mathcal{T}$.
- If $x \in \mathcal{T}$, then the formal symbol Not(x) is an element of $\mathcal{T}$.
- Similarly, if $x, y \in \mathcal{T}$, then the formal symbols And(x, y) and Or(x, y) are also in $\mathcal{T}$.

And, $\mathcal{T}$ is the least set with these properties.[11] This means that $\mathcal{T}$ is the set with all Java classes and any combination involving the function symbols Not, And, and Or. Note these formal symbols are intended to refer to set complement, set intersection, and set union in a standard interpretation.

The set $\Omega = \{\text{And}, \text{Or}, \text{Not}\}$ is the set of logic functions. Note, these functions are not actually yet defined. Their earlier use was intended only to be helpful in forming the $\mathcal{T}$ set, and did not actually define the objects. Luckily, the functions are easy to define, given the formal objects defined earlier. The function $\text{Not} : \mathcal{T} \to \mathcal{T}$ sends $x \to \text{Not}(x)$. Similarly, $\text{And} : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ sends $(x, y) \to \text{And}(x, y)$ and $\text{Or} : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ sends $(x, y) \to \text{Or}(x, y)$. Charmingly straightforward definitions.[12]

With the basic set of elements and logical functions done, there remains just the predicate definitions to finalise the logic syntax. The set of predicates is $\Pi = \{\text{Subtype} : \mathcal{T} \times \mathcal{T} \to \{\text{true}, \text{false}\}\}$. This single predicate captures the subtyping relation, the semantic definition of which being simplicity itself. The Subtype predicate is interpreted semantically as the subset relation on the set $\mathcal{T}$, where connectives have been given interpretations as set operations. However, the syntactic definition relation required by the resolver, is more awkward, consisting of numerous syntactic rules.

But before getting into the nasty business of outlining these syntactic derivation rules, some examples and implementation notes might be helpful. Example valid logical terms include:

- Or(`java.lang.Integer`, `java.lang.Long`, `java.lang.Double`) — Here any of the types under the Or function are permitted. Note the shorthand in writing all the terms under the same function symbol, rather than cascading Or functions. For convenience, either assuming some preprocessing, or that the Or, And functions are overloaded in the Ω set to handle arbitrary arities.[13]
- And(X, `java.lang.Byte`) — The expresses a class that is a `Byte` and also of the class or interface to which variable X is bound.
- Not(Or(`java.lang.Byte`, `java.lang.Short`)) — Any type but `Byte` and `Short`.

Implementation-wise, the *Types* logic uses subclasses as directed in the section on logic representation. These classes, illustrated in Figure 35 are all trivial extensions of their respective parent classes.

§64 Rules. *«Outline of syntactic rules»*

The *Types* logic syntactic deduction rules are represented in the software by `TypeRulesFactory`, a subclass of `RulesFactory`. These are the manipulations valid within the logic, used to produce derivations. For implementation purposes, these rules are actually represented in clausal form, but are presented here in traditional form. The *Types* logic rules include:

- Commutativity of the And, Or functions in both places under the Subtype predicate:

$$\begin{aligned}
\text{Subtype}(\text{And}(x, y), z) &\to \text{Subtype}(\text{And}(y, x), z) \\
\text{Subtype}(z, \text{And}(x, y)) &\to \text{Subtype}(z, \text{And}(y, x)) \\
\text{Subtype}(\text{Or}(x, y), z) &\to \text{Subtype}(\text{Or}(y, x), z) \\
\text{Subtype}(z, \text{Or}(x, y)) &\to \text{Subtype}(z, \text{Or}(y, x))
\end{aligned}$$

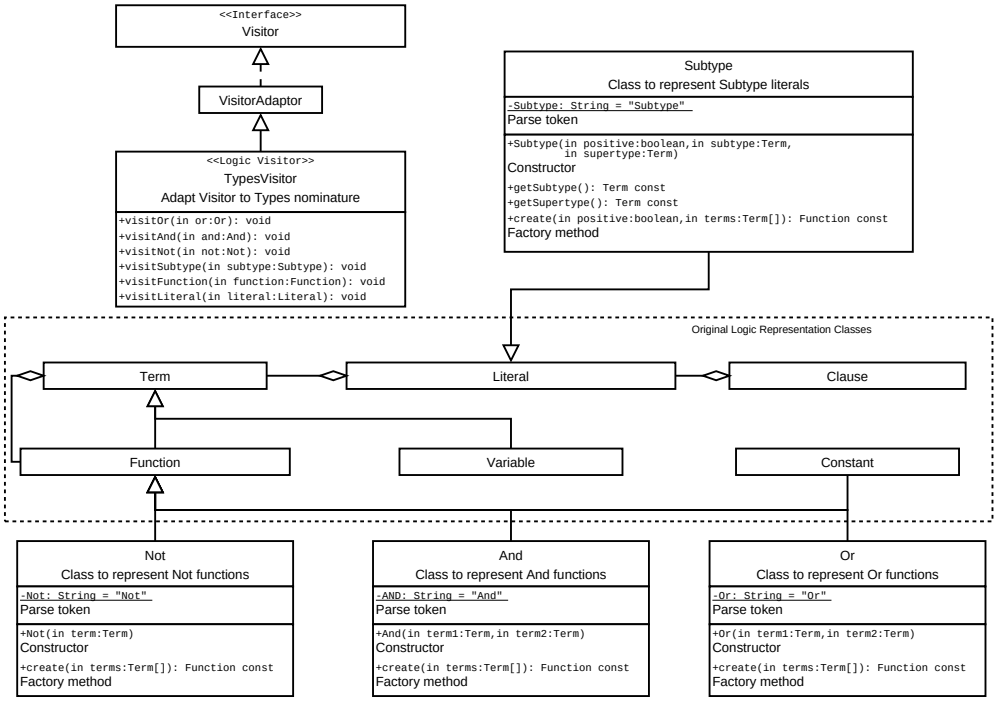

Figure 35: UML Diagram *Types* Implementation.

- Or commutativity under conjunctive normal form. And commutativity is not needed:

$$\text{Subtype}(\text{And}(\text{Or}(x,y),z),w) \rightarrow \text{Subtype}(\text{And}(\text{Or}(y,x),z),w)$$
$$\text{Subtype}(w,\text{And}(\text{Or}(x,y),z)) \rightarrow \text{Subtype}(w,\text{And}(\text{Or}(y,x),z))$$

- Associativity of the And, Or functions in both places under the Subtype predicate:

$$\text{Subtype}(\text{And}(\text{And}(x,y),z),w) \rightarrow \text{Subtype}(\text{And}(x,\text{And}(y,z)),w)$$
$$\text{Subtype}(w,\text{And}(\text{And}(x,y),z),) \rightarrow \text{Subtype}(w,\text{And}(x,\text{And}(y,z)))$$
$$\text{Subtype}(\text{Or}(\text{Or}(x,y),z),w) \rightarrow \text{Subtype}(\text{Or}(x,\text{Or}(y,z)),w)$$
$$\text{Subtype}(w,\text{And}(\text{Or}(x,y),z),) \rightarrow \text{Subtype}(w,\text{Or}(x,\text{Or}(y,z)))$$

- And-Or distribution in conjunctive normal form direction:

$$\text{Subtype}(x,\text{Or}(\text{And}(y,z),w)) \rightarrow \text{Subtype}(x,\text{And}(\text{Or}(y,w),\text{Or}(z,w)))$$
$$\text{Subtype}(\text{Or}(\text{And}(y,z),w),x) \rightarrow \text{Subtype}(\text{And}(\text{Or}(y,w),\text{Or}(z,w)),x)$$

- De Morgan Laws, in conjunctive normal form direction:

$$\text{Subtype}(\text{Not}(\text{And}(x,y)),z) \rightarrow \text{Subtype}(\text{Or}(\text{Not}(x),\text{Not}(y)),z)$$
$$\text{Subtype}(\text{Not}(\text{Or}(x,y)),z) \rightarrow \text{Subtype}(\text{And}(\text{Not}(x),\text{Not}(y)),z)$$
$$\text{Subtype}(z,\text{Not}(\text{And}(x,y))) \rightarrow \text{Subtype}(z,\text{Or}(\text{Not}(x),\text{Not}(y)))$$
$$\text{Subtype}(z,\text{Not}(\text{Or}(x,y))) \rightarrow \text{Subtype}(z,\text{And}(\text{Not}(x),\text{Not}(y)))$$

- Double negation rules. Needed under the And symbol and under conjunctive normal form:

$$\text{Subtype}(x,\text{Not}(\text{Not}(y))) \rightarrow \text{Subtype}(x,y)$$
$$\text{Subtype}(x,\text{And}(\text{Not}(\text{Not}(y)),z)) \rightarrow \text{Subtype}(x,\text{And}(y,z))$$
$$\text{Subtype}(x,\text{And}(\text{Not}(\text{Not}(y)),z)) \rightarrow \text{Subtype}(x,\text{And}(y,z))$$

- Subtype identity:

$$\text{Subtype}(x,x)$$

- Subtype transitivity:

$$\text{Subtype}(x,y) \wedge \text{Subtype}(y,z) \rightarrow \text{Subtype}(x,z)$$

- Subtype inclusion rules:

$$\text{Subtype}(\text{And}(x,y),x)$$
$$\text{Subtype}(x,\text{Or}(x,y))$$
$$\text{Subtype}(x,z) \rightarrow \text{Subtype}(\text{And}(x,y),z)$$
$$\text{Subtype}(\text{Or}(x,y),z) \rightarrow \text{Subtype}(x,z)$$

- Subtype connectivity distribution rules:

$$\text{Subtype}(z,\text{And}(x,y)) \rightarrow \text{Subtype}(z,x) \wedge \text{Subtype}(z,y)$$
$$\text{Subtype}(\text{Or}(x,y),z) \rightarrow \text{Subtype}(x,z) \wedge \text{Subtype}(y,z)$$

- Actual subtype facts. For each class or interface, c with immediate superclass or superinterface d, there must be a fact of the form Subtype(c, d). Further, for each interface f implemented by class e, there must be a fact of the form Subtype(e, f). These facts need not be explicitly specified, but they must be available from the `RulesFactory` via a schema if necessary. These facts may be generated for classes of interest by Java reflection and by walking up the inheritance tree.

§65 Example Derivations. *«Illustrative examples of derivations»*
As an example of these deduction rules, consider the derivation of the fact:

$$\text{Subtype}(\texttt{java.lang.Integer}, \texttt{java.lang.Object})$$

Firstly, by virtue of the last rule, the following class hierarchy facts are directly available:

$$\text{Subtype}(\texttt{java.lang.Integer}, \texttt{java.lang.Numeric})$$
$$\text{Subtype}(\texttt{java.lang.Numeric}, \texttt{java.lang.Object})$$

These may be combined by using the Subtype transitivity rule to give the following implication:

$$\begin{aligned}&\text{Subtype}(\texttt{java.lang.Integer}, \texttt{java.lang.Numeric})\\ &\quad\wedge \text{Subtype}(\texttt{java.lang.Numeric}, \texttt{java.lang.Object})\\ &\quad\rightarrow \text{Subtype}(\texttt{java.lang.Integer}, \texttt{java.lang.Object})\end{aligned}$$

Now, the left hand side of this is a fact by propositional logic expression.[14] So, the right hand side is derived by Modus Ponens, and happens to be the desired goal:

$$\text{Subtype}(\texttt{java.lang.Integer}, \texttt{java.lang.Object})$$

Consider instead, the slightly more complicated example of deriving:

$$\text{Subtype}(\texttt{And}(\texttt{java.lang.Integer}, \texttt{java.lang.Numeric}), \texttt{java.lang.Object})$$

This is derived from the Subtype inclusion rule:

$$\text{Subtype}(x, z) \rightarrow \text{Subtype}(\text{And}(x, y), z)$$

With the instantiations $x \leftarrow \texttt{java.lang.Integer}$, $y \leftarrow \texttt{java.lang.Numeric}$, $z \leftarrow \texttt{java.lang.Object}$. Now, since the left hand side holds by previous work, the right hand side may be derived by Modus Ponens, and happens to be the desired derivation goal:

$$\text{Subtype}(\text{And}(\texttt{java.lang.Integer}, \texttt{java.lang.Numeric}), \texttt{java.lang.Object})$$

§66 Variable Binding. *«A mention of variable binding»*
Before leaving the discussion of logic rules, there are a few points to mention about variable binding. In type checking, operation output types are validated against following operation input types using the Subtype predicate, requiring maintenance of Subtype(output, input) facts. This is straightforward in the case that

output and inputs types are constant terms, or functions of constant terms which do not involve variables.

The use of variables in input or output types might help express a lack of definite typing information, or may indicate a high degree of typing flexibility is possible. The general idea is that variables would be allowed to float and bind freely in the Subtype(output, input) expressions.

There are advantages to be gained in binding variables within the scope of an operation. So, suppose variable X appears in input type to an operation, and is bound to some unifier expression as part of the Subtype(output, input) establishment, then if that variable also appears in the operation output type, or in any of the other inputs, then it must respect the previously established restraint. In fact, the individual values for a variable must be unified to produce a most accurate typing restraint for that variable.

More will be said later about the typing mechanism employed in Condensed Graphs. In particular, the nonlocal implications of allowing local variable bindings will be discussed, together with algorithms for the type validation of what now amounts to a DAG with interconnected constraints on the arcs.

5.3 Example: Security Reduction Rules

As an example application of this logic programming structure, security reduction rules will be briefly considered. So, although designed for implementing type checker validations, the logic support may be reused within WebCom for a variety of useful purposes, not necessarily typing related.

Security reduction rules are a niche application of the sort to which logic programming is ideally suited. That logic programming may be leveraged for small concerns within a large software design, like WebCom, is of utility to the programmers of this system. With the logic programming support in place, applications with concise FOPL expressions, such as the security reduction rules, may be very effectively solved.

The backdrop to this application is the use of Keynote within the WebCom security system. Secure names are used to provide authentication mechanisms, and consist of strings describing particular element of the WebCom system, be they nodes, graph, operations, or other elements. The actual security architecture particulars within WebCom are not of interest. Instead, what is important is that precise secure names can tend to be long unwieldy strings. So, for display purposes, and more importantly for group authentication and organisation reasons, these secure names have text manipulation requirements. For instance, a user might have a particular secure name expressing membership of particular domain or company group organisation. This name must be easily reduced textually in order to extract this information.

Secure names are organised in a treelike structure which may be leveraged easily into FOPL symbols. In fact, the correspondence between secure names and FOPL expressions is very close. Logic programming excels at such text parsing applications, and is ideal for implementing secure name related tasks.

One such task is the aforementioned name manipulation. Rules are prescribed in order to transform long precise names into shorter names. These shorter names may contain only specific information required to make a desired authentication within WebCom. For instance, a WebCom secure name might contain fields for a domain and an ID. In this case, the name could be written in the form `name(domain(xxx), id(yyy))`.

Notice that this syntax, used by the secure name applications, already has FOPL structure. One reason for reduction rules, is that in many authentication cases only the domain is of interest and all requests from a particular domain are accepted, i.e., `name(domain(xxx), null)` is sufficient for authentication.

The job of the reduction rules is to determine if a given secure name matches any of a set of general acceptable names, specified in the security policies. Software is required to transform `name(domain(xxx), id(yyy))` into `name(domain(xxx), null)`, according to rules such as:

$$\texttt{name}(\texttt{domain}(Y), \texttt{null}) \leftarrow \texttt{name}(\texttt{domain}(Y), X)$$

Given this example, a logic programming based design for a reduction rule engine is straightforward. Syntactically, the function symbols `name`, `domain` and `id` are required, together with appropriate constants for the domain and ID fields. A two place predicate `reducesTo` is needed to express the reduction relationship between such elements as in the example, i.e., the earlier rule becomes:

$$\texttt{reducesTo}(\texttt{name}(\texttt{domain}(Y), X), \texttt{name}(\texttt{domain}(Y), \texttt{null}))$$

So, reduction rules may be specified textually, avoiding any need to hardcode reduction rules into the engine. This is just a single simple reduction rule example. A typical implementation would include more name fields, and more complicated name reductions. But, the basic idea is the same.

To verify an authentication, the system requires a secure name to test, and a list of the acceptable secure names in FOPL form. The resolver checks if the test name reduces to an acceptable names, by simple queries. The resolver also takes care of applying reduction rules multiple times, if necessary, backing out of reduction paths, etc. This invisible search work implementation is invaluable to the reduction rule engine programmer. She just needs to express the logic, do whatever parsing is necessary, and invoke the resolver.[15]

Note there are many similarities with the typing system. This is unsurprising given that both are search implementations, and illustrates the value of incorporating the resolver engine into WebCom. Future search applications may use this existing code to greatly simplify implementation work.

Chapter Notes

[1] Which is what they are really doing in logic programming anyways, but it helps to hide this from the novice.

[2] And in fact, this is the point. It is up to the logic designer to assign a meaning, or an *interpretation* to these items in a way that make senses for the particular application. FOPL works in the same manner regardless of the actual application and interpretation.

[3] This validity interpretation is done semantically using the interpretation function.

[4] Really just the functions and predicates. Developers are encouraged to subclass `Constant` to introduce meaningful class names.

[5] Not all of conjunction, disjunction, negation, and implication are required of course. Logical implication is sufficient and probably the most convenient. Conjunction, disjunction and negation might be implemented by a prepass syntactic rewrite.

[6] This latter approach has much overlap with the type set logic which will be seen later in relation to the type checking logic. Some implementation hints for such a propositional logic overlay may be inferred from this type checking logic.

[7]More precisely, the cost is expressed as *nm*, where *n* is the number of literals in the test clause, and *m* is the total number of literals in all the clauses managed by the `Clauses` object. It is also the number of literals in the query clause times the number of managed clauses times the average number of literals per clause in the `Clauses` object.

[8]The actual occurrence of search tree exhaustion is questionable. For certain small logics, exhaustion will come quickly and be an effective decision tool. For medium logics, exhaustion can be an expensive and undesirable computation, and for larger logics, search exhaustion will be preceded by resource exhaustion.

[9]Substitution values may differ, of course, in the carried state substitution, so a state may not be permitted on the in state queue if it was seen before but with a different substitution. This is the accepted position and independent of variable structure representation concerns, anyway. The alternative, incorporating substitution data into determinations of visited states, is not practical.

[10]The space of possible Java classnames is countably infinite, unfortunately. However, it is always possible to reason about types within the context of a particular JVM. And the set of types available to this JVM is finite, being bounded above by the set of all Java classes ever implemented, of course. So, it is possible to depend on finiteness in arguments about the type logic in practice. Typically, the denumerable space for Java classnames will be employed for convenience of expression, but with the understanding that if ever necessary, the brutality of finiteness may be subsumed into arguments. Note since the set of classnames is considered finite, so the powerset is also finite. Logic constants are taken from this powerset and not from the ground set since logic constants are sets of classes.

[11]A least such set exists by the virtue of the fact that sets satisfying these criteria may be formed into a nonempty poset under inclusion. Then some standard arguments, involving presuming no unique least element exists and deriving an easy contradiction, will ensure a set with the desired properties exists. But, it is easier just to say Zorn's Lemma downward on top of the chain, and be done with it. We know things won't disappear out from under this chain, so we are fine.

[12]Of course, this is not really true. The production of elements to populate $\mathcal{T}$ by depending on the syntactic nature of the function names is quite a very distinct thing from imbuing them with a functional definition as done at this point in the text. The difference is quite important, being the essential notion in stepping from a syntactic form to a semantic form where the semantic quality is built from the syntactic element. Which isn't a straightforward thing at all, but still charming.

[13]This doesn't matter either way, since the And, Or functions are associative both in semantic interpretation and the syntactic rules.

[14]This is the real propositional logic, not the pretend one that looks like it is hiding in the And, Or functions of the logic.

[15]This parsing and resolver invocation work appears also in the type checker application. Since these aspects of the type checking will be covered in fine detail in the next chapter, there is no real reason to illustrate them in regards to the security reduction rule application.

6

Type Checking

This chapter brings together all the key software structures previously considered in this dissertation, to provide Condensed Graph type validation software. Meta information notations specify typing information. The Event API facilitates implementation without WebCom core modifications. The Module API provides the application packaging and third party interface. And, the resolver engine implements basic typing verifications as outlined in the *Types* logic description.

However, this does not exhaust all requirements. There remains considerations in translating metainformation type strings into usable software elements, namely representations in the *Types* object hierarchy. Also warranting discussion is the type checking modularisation. The various module configurations and roles, together with the type checking application modes, are outlined in this chapter.

Finally, the checker algorithms are outlined with attention paid to the graph walking problems in the design time verification, and to dealing with possible verification paths. The chapter concludes with some notes on further directions, and recalls the developments in this dissertation.

6.1 Parser

The syntactic portions of the *Types* logic have already been outlined, consisting essentially of the functions And, Or, Not, together with the predicate SubType, and constant symbols, one for each possible Java classname string in fully quantified notation.

So far, the *Types* logic implementation has consisted of specialized extensions of basic logic software objects. This enables *Types* statements to be formed programmatically and verified by the resolver engine. But programmatic construction is unwieldy and restrains an easy dynamic expression of logic statements.

The *Types* logic thus requires a parser frontend. While parser implementation is not excessively difficult, it helps to carefully examine this implementation and interaction with existing logic software. This does represent a diversion before the main type checking discussion, however.

Once implemented, the parser converts operator `Info` type string data into actual software logic classes. So, when type checking a particular operation, the `Info` for that operation is instantiated and the type strings parsed.[1] The results of this parse are then used by the resolver to verify the satisfaction of input operand and output types with the types incident on them.

The parser is generated using the SableCC parser generator tool, written by Etienne M. Gagnon. This tool is applied to a BNF description of the *Types* logic, and produces comprehensive parser software which will be described presently. BNF outline particulars will be mentioned first.

§67 BNF Description. *≪A BNF description for the type element parsing≫*
Figure 36 illustrates the *Types* logic BNF. A complete grammar including token and helper declarations, and as used in the SableCC processing, may be found in Appendix D. The *Types* logic BNF start symbol is the Type production, describing acceptable type strings. These acceptable types are:

- A parenthesised Type production, included for convenience in writing type strings.[2]
- A string denoting a constant in the logic, i.e., a Java classname string.
- A variable type denoted by the VAR symbol and a parenthesised variable identifier.
- A Not type, denoted by the NOT symbol and argument type to invert.
- An And type, denoted by AND and a parenthesised list of types to conjoin. This list, expressed by the And_List production, may contain more than just two types for convenient cascaded And expression.
- An Or type following the model of the And type, but instead supported by the Or_List production.

§68 SableCC Parser. *≪The SableCC produced parser software≫*
SableCC is a parser generator tool, in the style of a lex and yacc combination, designed by Etienne M. Gagnon. It provides for both DFA lexing and LALR(1) parsing based on eBNF grammar syntax. The real SableCC value, though, is its OO parser generation. SableCC output consists not only of basic lexer automaton and parser classes, but also of an OO abstract syntax tree based on the input BNF. SableCC automates the task of producing a parser *and* complementary tree hierarchy. So, given a grammar, the SableCC tool may be applied without further work to produce a complete parser software architecture.

With SableCC, a software designer need only provide an input BNF, and implement a visitor adaptor to perform desired semantic operations on the generated abstract syntax tree. Ease of application and the

⟨Type⟩ ⟶ ‘(’ ⟨Type⟩ ‘)’
| ⟨Constant_Name_String⟩
| ‘VAR’ ‘(’ ⟨Variable_Name_String⟩ ‘)’
| ‘NOT’ ‘(’ ⟨Type⟩ ‘)’
| ‘AND’ ‘(’ ⟨And_List⟩ ‘)’
| ‘OR’ ‘(’ ⟨Or_List⟩ ‘)’

⟨And_List⟩ ⟶ ⟨Type⟩
| ⟨And_List⟩ ‘,’ ⟨Type⟩

⟨Or_List⟩ ⟶ ⟨Type⟩
| ⟨Or_List⟩ ‘,’ ⟨Type⟩

Figure 36: BNF *Types* Logic

extent to which SableCC integrates into an OO designed application mean SableCC is an effective tool for implementing “little language” elements in a larger software design.

When run on the *Types* BNF, SableCC produces a parser for converting metainformation typing strings into automatically generated abstract syntax tree objects. However, the existing logic tree hierarchy,outlined in the previous chapter, is logic language independent and so superior to a SableCC generated tree. Further, the resolver is designed to operate specifically on the more general tree hierarchy.

This does not present difficulties, though, since a simple SableCC visitor walker may be written to directly map SableCC trees into custom designed *Types* logic trees. Note, the resolver might have been implemented to operate with the automatically generated SableCC trees instead. This would have just meant finalising the *Types* language and generating the SableCC abstract syntax trees before implementing the resolver architecture. The resolver code could then have been implemented to employ the SableCC generated classes.

Although possible, this approach is not a good design for two reasons. The first being that it is fragile to changes in the *Types* logic specification. So, if the structure of type information strings is changed for whatever reason, there are potentially nontrivial knockon changes required in the resolver architecture. For this reason, using a mapping tree walker to produce logic trees from SableCC trees is a gain, in that it isolates the resolver structures from syntactic changes in the expression of type strings.

The second, and more important, reason is designing the resolver to use the SableCC tree means limiting it to the *Types* language only. The resolver is a valuable software development tool within WebCom and should not be restricted to just a single application.

Although the implementation chosen requires a mapping visitor class and incurs overhead on parsing operations, this is also a necessary and justified cost. The amount of actual parsing and mapping necessary

may be limited by caching and reusing previously processed logic structures. Nevertheless, the parsing is already quite efficient and need not be an optimisation focus.

Detailed examination of the generated parser and of the generated abstract syntax tree, in particular, is unenlightening. Especially, since these components are not used outside of their parsing application. However, the handcoded tree walker mapping SableCC trees into *Types* logic trees deserves some mention. This visitor class, `TermParser`, extends the automatically generated depth first visitor adaptor, `DepthFirstAdapter`. Consequently, this operation is *Types* logic parser particular, but may be adapted easily for other languages.

`TermParser` visits the SableCC tree elements which correspond directly to `Constant`, `Variable`, `And`, `Or`, and `Not` types in *Types* logic trees, and generates an appropriate *Types* logic tree element in each case. If mapped elements are constructed on the postvisit side of depth first node visiting, then any necessary referenced types will already have been created. So, for instance, in generating an `Or` element, the argument type objects needed will have already been generated by previous deeper walk visits.

It is also convenient to implement a facade `parse` method in the `TermParser` class. This method takes a type string input and produces a corresponding *Types* logic element, if possible. This method arranges the necessary SableCC lex and parse calls, then invokes the `TermParser` walker on the resultant object to generate the desired *Types* logic object.

6.2 Runtime Type Checking Problem

The basic approach to Condensed Graph type checking has already been mentioned. The idea being to associate types to operator inputs and outputs, and to verify output types are subtypes of types at operator inputs to which they are connected. This typing information is stored in operator `Info` objects and is parsed into *Types* logic tree elements by the above parser tool.

§69 Basic Runtime Type Checking. *≪Issues in type checking Condensed Graph at runtime without variables≫*
There are two primary type checking scenarios, runtime verification and beforetime verification. Beforetime verification will be considered below, so for now type checking is assumed to run at Condensed Graph execution time. Upon realisation of a full computational triple, or fireable node, in the WebCom engine, the type safety of the generated instruction might be verify. In this case, all actual operands and types are available, and this scenario involves just checking subtype relations for each operand, i.e., checking that the actual operand class type is a subtype of the required class type.

This kind of immediate type verification is common in interpreted execution. Implementing this within the proposed WebCom type checker is not difficult, and would increase graph execution confidence.

It helps to first consider the variable free implementation case, the case where no type strings use variables. This simplifies operand class type verification considerably. Checker rules include:

- For constant operand types, the checker verifies the actual operand is a subtype of the required constant.

- In the case a Not function type is required, the checker extracts the Not argument type and compares the actual operand class to that type. The check fails if the operand class satisfies this subtype relation.
- If the required type is an And type, then the checker checks the operand type against each of the And type arguments, fails the test if the operand fails against any argument type.
- Similarly, with an Or type, the checker tries the operand against all of the Or argument types, but just needs one of the subchecks to succeed.

Note none of this requires use of the resolver, and so may be efficiently implemented. Further, the *Types* logic syntactic rules are not used in this simplified decision procedure, due to the restricted query natures. No expression rewriting is needed to verify logic queries of the form Subtype(constant, constant).

This checker routine does not offer the full expressiveness available from variable use, but does provide a simple effective check if variables are either not employed or discounted. In the case that variables are to be ignored, they may be substituted with the `Object` type.

This relaxation of variables to the `Object` type will probably mean certain invalidly typed operands are passed when they ought to fail. However, this relaxation to `Object` is not as slack as might be supposed and much poor typing will still be caught. For instance, suppose the required type string is:

$$\text{And}(X, \texttt{java.lang.Integer})$$

In this case, the realised type in the check is just `Integer`, and the checker will fail anything not an `Integer` regardless of the variable typing. In particular, if the variable does not additionally restrict the type below `Integer`, then the checker will function completely correctly in this case.

This approach gives a fast simple check, providing reasonable but not complete levels of type security. For practical purposes, this might be adopted as the runtime checker algorithm of choice in a WebCom system especially if designtime verification is also employed.

§70 Variables and Runtime Type Checking. *≪Variable use semantics. Runtime checker implementation with variables≫*
Variables used in type strings may match against any permitted type in the logic, in line with FOPL variables use. This is implemented by using the resolver to organise the direct verification of types, including the case of variables. A type check involves invoking a simple Subtype query, to check an actual operand classname constant is included in the required operand type. The resolver takes care of any variable binding necessary.

It is even possible to disregard many logic rewrite rules and operate with a sublogic of *Types*, so improving resolver efficiency considerably. In particular, logic rules involving composite element rewrites on the left of Subtype predicates are superfluous. As are some rules involving rewrites on right of predicates.

The runtime check in the presence of variables might be implemented entirely without recourse to the resolver, but is considerably complicated requirements to find correct variable substitutions. This is especially problematic if the same variable many times in the target type, or under a Not negation function. The need

for the resolver, or for some clever unification, may be seen in examples such as:

$$\text{Subtype}(\texttt{java.lang.Integer}, \text{Or}(\text{Not}(X), \text{And}(X, \texttt{java.lang.Numeric}))$$

Matching against the left Or argument means binding $X \rightarrow \text{Not}(\texttt{java.lang.Integer})$, and one possible correct matching on the right would bind $X \rightarrow \texttt{java.lang.Integer}$. The problem is incorporating binding value selection into the checker in the absence of a resolver invocation. This example is manageable, but consider multiple Not symbols in the left argument, and nested in a complicated structure, and with more than one variable. Determining the correct variable bindings could be very difficult in such cases.

The above example is also interesting since the two binding choices are mutually incompatible. The choice does not matter in direct verifications of operands against targets taken pairwise, since all that is required is some instantiation making the statement true. The particular binding is not of much interest.

This is not a complete reflection of the variable type checking problem, though. Simple point to point verification is sufficient in the absence of variables, but there is more to consider with variables. Specifically, variable binding scope extends beyond simple queries and cannot be considered operandwise alone.

Variables are bound on a per node basis. So, if an operand or output node position mentions a variable, that same instantiation is visible to all operand and output positions of that node. This means that if a variable is bound during an operand verification say, and if this variable is also mentioned at a different type verification on the same node, then the previous variable binding must be respected.

So, numerous type verifications are made with shared variables in verifying node typing. Operation type verification begins with a blank variable bindings list and takes each operand verification in turn. The first operand type is verified or refuted using the resolver. Now, a full use of the resolver may give multiple satisfying bindings, as in the earlier example, and each of these bindings is a possible initial mapping from which to check the second operand. Further, verifying subsequent operands presents even more alternatives.

Verification of the second operand is done in the context of the first operand verification results. Simply applying the first result substitutions to the second operand target type might be too narrow, since the second verification may only be successful with looser variable binding than required in the first operand verification.

A solution is to verify the second operand directly, without considering the first operand verification substitution. If the second operand verification returns a candidate substitution then this is reconciled with the first operand substitution if possible. This reconciliation action is described in detail below, but can be interpreted as ensuring the substitutions may be combined without introducing inconsistencies.

If the substitutions can be reconciled, then the second operand verification is accepted and the the third operand may be considered, etc. Note, that the second operand verification might produce many candidate substitutions, which, if successfully reconciled, form the alternatives for the third operand verification.

This gives rise to an explicit search procedure, and it may be better to instead use the resolver implicit search itself. In this case, the conjunction of operand and output subtype checks for a single operation would form a composite query for resolver verification. This approach eliminates the need to reconcile substitutions, but represents a potentially very difficult resolution.

For purposes of runtime verification, resolver use is potentially too computationally expensive. Neither are search operations during graph executions justifiable, even with verification result caching. A nonvariable checker with `Objects` for variables may be the most practical option for runtime verification.

If practicality is not a consideration, the composite query approach is a straightforward solution algorithm. However, there are two morals here. The first being that the resolver presents efficiency difficulties which may be too costly at runtime. The best hope may be for practical designtime verification, where typing information may be retained for use in runtime checks. Alternatively, designtime checks might be digitally signed, and the graph thereafter used with confidence at runtime.

The second moral is in operand type checking processing and substitution reconciliation. This clumsy process illustrates how the more complicated designtime verification problem will be at least as complicated.

6.3 Designtime Type Checking

The second type checker scenario is before runtime, or designtime, static type checking, typically employed in graph design or automatic construction tools. The idea is to verify an entire graph, rather than just a single operation. This involves considering graph node interconnections, which is not terribly difficult but does require a level of computational effort beyond that in runtime type checking.

In fact, designtime type checking is not too different from runtime checking. The main differences being that designtime checking involves a series of checks of the form performed in runtime type checking, and also that input types are no longer as simple. This is because complicated type strings may propagate from outputs to next inputs without acquiring a concrete actual type as in the case of runtime verification.

§71 Variable-free Designtime Type Checking. *«Designtime type checking considered without the problem of variables»* Designtime checking will first be considered without type string variables. This is not as great a simplification as in the runtime checking case, since output strings need not be as straightforward. And because output string are fed forward to next inputs, deciding subtype inclusion at operand inputs is not as easy.

The basic problem outline begins with a DAG, G. This graph is intended to represent the Condensed Graph to be verified, but reinterpreted in operand flow terms rather than in regular Condensed Graph arc semantics terms. The graph G possesses a node for each Condensed Graph node, and an arc from node e_1 to e_2 if the output of the Condensed Graph node corresponding to e_1 is directed to an input of the Condensed Graph node corresponding to e_2.[3] Arcs are assigned directions from output ports to input ports.

Each directed arc of G is assigned the Subtype *Types* logic predicate to be tested on that arc. So, if X is the output type of node e_1 and if Y is the input type of the operand on node e_2, to which e_1 is connected, then the logic predicate statement Subtype(X, Y) is associated to the arc from e_1 to e_2.

It helps to relabel logic variables to enforce the variable scope rules described earlier. To do this, each graph node is processed in turn. At each node, the variables in the *right hand* places of subtype logic predicates on incident in arcs are relabeled, as are the variables in the *left hand* places of subtype logic predicates

on incident out arcs. The relabeling used does not matter, save that labels are globally unique on a per node basis, perhaps via the use of specific node prefixs to variable identifiers.[4]

The problem in designtime checking is to find a variable instantiation making all subtype predicates true simultaneously. Of course, this could be verified by the resolver in a manner akin to the large composite query option proposed earlier. But, the question arises as to whether there might be a more efficient approach.

So, given that type strings are assumed variable free, there is potential to adapt the variable free runtime algorithm to designtime verification. The hope is a direct pairwise verification procedure may be developed such that a complete graph verification amounts to the iterative application of this procedure to all arcs.

This may be the case, but an approach along the lines of the previous subterm decomposition technique runs into problems. The approach which closest mirrors the previous effort is to decompose the right subterm of test Subtype predicates into constant terms. This may be done in the same fashion as in runtime checking, so it may be assumed that all test right subterms are constants.

At this point, the subtype predicate may be reversed, since Subtype(X, c) is the same check as $X \subset c$, or as $c' \subset X'$, by set complementation. So, potentially composites term may be transfered to the right subterm place, leaving tests of the form Subtype(Not$(c), X$). Here, the X subterm may be decomposed as before, leaving tests like Subtype(Not$(c), d$). These, however, may be difficult to check. Although, not impossible, these may require large indices in order to verify. For instance, to determine Subtype(Not$(c), d$), it is necessary to know if the types represented by classname constants c and d intersect. However, this means, for instance if d is an interface, that all subtypes of c need to be examined to determine if they implement the interface d.

If this test problem can be overcome, this mechanism might be used as a loose designtime check mirroring loose runtime checks. As in the runtime case, variables may be replaced by `Object` type to get closer verification. This may be a more efficient approach than the strict checking described below.

Note the test problem identified above is not insurmountable, since indexing an entire JVM classname space is practical and routinely done in Java IDE tools, for instance. There is a one-time index construction penalty, and care is needed in datastructure choice, but otherwise this forms a very realistic option.[5]

§72 Variable Designtime Type Checking. *≪Considerations of the full type checking problem at designtime≫*
Including variable type expressions does not change the problem definition given earlier. The statements defining the type checking designtime problem in terms of graphs and logic expression populated arcs still applies. Furthermore, the full designtime type validation problem may still be solved by invoking the resolver on a large composite query consisting of all the properly scoped subtype expressions conjoined.

Variable binding problems still complicate the task of deriving a full designtime check algorithm without toplevel recourse to the resolver. Resolver use is unavoidable, of course, in order to validate direct arc checks since logic language rewrites may be required. So, rather, the goal is to minimize resolver use and localise resolver application to single arc problems. However, this runs into difficulty with variable reconciliation.

The idea is to check each graph node in turn, verifying operand types with preceding output types. However, this verification must be done in the presence of variable binding concerns. So, all candidate resolutions for the operand types must be reconciled before a resolution may be accepted as a valid candidate.

Operand types are validated in turn at each node and any returned resolution substitutions need to be confirmed against a maintained set of valid substitutions collected in the course of the previous verifications. The current resolution substitution is either acceptable against some of the potential global solution substitutions, in which case it is combined with these substitutions. Alternatively, the resolution substitution does not conform to some substitution candidate, in which case alternative resolutions are required to maintain those substitution candidates, or they must be dropped.

This process is effectively implementing what would be the resolver search in the case of the large toplevel resolver strategy. As such, search strategy concerns might be considered. The use of backtracking and the maintenance of search structures is also a problem in the design of this algorithm.

§73 Statement of designtime type checking algorithm. *«Description of the potential algorithm operation»*
For clarity, it may be better to simply state the algorithm operation, rather than try to talk around it. The algorithm for beforetime type checking verification begins with a Condensed Graph to verify. For convenience, it is assumed all type strings have been extracted and relabeled according to scope rules. This can be done by processing each node in turn, determining the operator at each node, [6] and instantiation the associated `Info`. Type strings may be extracted from this `Info` and variables relabeled to be globally unique. Type strings must then be stored in accessible form for later lookup, perhaps by hashing based on parent node and operand index. These type strings may also be presumed to have been parsed into *Types* logic objects.

The main algorithm body consists of processing each graph node in turn, and it may be better to process nodes in a breath first order, so as to maintain locality in the type checking. This local reference is useful in eliminating substitution alternatives invalidated in the near neighbourhood of their construction.

It is also necessary to maintain a database of valid possible solutions, together with tree walk position state, for algorithm purposes. In this way, the algorithm may also journal various backtrackings and so forth which the resolver would otherwise have managed in a large toplevel subtype check. This database is initially populated with an empty substitution and an initial state marker.

At each node, the algorithm verifies typing data for each operand in turn. For a single operand verification, the operand type and the preceding output type are extracted from the storage. At this point, the resolver is invoked to verify the previous output type is a subtype of the current operand type. The resolution will result in a set of substitutions making the statement true. Since all of these resolution substitutions might be considered, the resolver must run to completion on the test. This is unfortunate from an efficiency viewpoint.

Each candidate resolution substitutions must be vetted for consistency with each substitution in the set of substitutions maintained by the algorithm. This vetting is the reconciliation step mentioned earlier and will be discussed more in the next section. The purpose of the reconciliation step is to establish whether the current candidate resolution substitution may be combined with a particular substitution from the set held by the algorithm. If it can, then this give rises to a new substitution in the algorithm list.

More precisely, the resolver returns potential substitutions, and the algorithm maintains currently valid substitutions. Resolver substitutions are pairwise reconciled with algorithm substitutions, and successful reconciliations form the algorithm substitutions for the next step. Verification fails if this set becomes empty.

This list of substitution possibilities also tracks the backtracking which would be managed by the resolver in the large query case. There are opportunities for clever backtracking in this explicit search organisation, and for other optimisations not possible in the resolver solution. Moreover, running resolution to completion at each operand verification may not be necessary, since the search may move deeper in the search tree without waiting for all operand resolution possibilities.

6.4 Substitution Reconciliation

Substitution reconciliation is the process of examining two substitutions to determine compatibility. Compatible substitutions are then merged. For the discussion here, assume the substitutions to reconcile are θ and ϕ. Let $\Sigma = \text{vars}(\theta) \cup \text{vars}(\phi)$ denote the variables mentioned in these substitutions, and as before T denotes the *Types* logic term space, i.e., the model for the logic elements which may be referenced, or the space of objects for which a *Types* logic variable may stand. So, logic variables may be viewed as functions into T.

This setup views substitutions as functions from $\Sigma \rightarrow T^{|\Sigma|}$. There is a subset of Σ consisting of variables in the substitution θ which are bound to new values. In the function corresponding to θ, denoted θ_f, the elements of this set are mapped to the corresponding elements of T in the substitution expression. The other elements of Σ should be mapped to an element representing the whole of Σ, which for simplicity may be the element corresponding to `Object`.[7] A similiar construction is used to form ϕ_f.

Now, given these two substitution functions, $\theta_f, \phi_f : \Sigma \rightarrow T^{|\Sigma|}$, define $\theta_f \cap \phi_f$ is componentwise as $\theta_f \cap \phi_f(s) = \theta_f(s) \cap \phi_f(s)$. This represents the substitution which is the less strict refinement of both θ and ϕ, i.e., the infimum in the set inclusion based lattice for substitution functions $\theta_f, \phi_f : \Sigma \rightarrow T^{|\Sigma|}$.

Reconciliation means computing $\theta_f \cup \phi_f$ and failing if any component of this function has an empty set value. This corresponds to the case where the substitutions are not compatible. If successful, reconciliation returns the $\theta_f \cap \phi_f$ function as a substitution to the caller. The reconciliation of θ and ϕ is done as follows:

- All the variable name and binding pairs from ϕ are inserted in a queue, Q. These variable-binding pairs are processed in turn, ending up in a new substitution, μ, initialised to contain the bindings in θ
- On processing a variable-binding pair from Q, if the variable is not already in μ as a variable-binding, it is inserted by substitution composition and the next variable-binding pair from Q considered.
- On the other hand, if the variable-binding already appears in Q, then the two substitutions must be combined. Suppose $S \rightarrow T$ is already in μ and the current variable-binding is $S \rightarrow T'$. Here T and T' may have any instantiations, but the S variable must stand for itself, of course. The correct new binding should be $S \rightarrow \text{And}(T, T')$. However, it must be determined that this new binding is not empty before accepting the reconciliation. This check might be done, for instance, by the resolver in querying for an X value for $\text{Subtype}(X, \text{And}(T, T'))$. It may otherwise be checking by examining T and T'.
- This process continues until either a failure is isolated or Q becomes empty. In this latter case, the μ substitution is returned as the result of the reconciliation.

6.5 Type Checker Modules

These reconciliation and verification algorithms need to be parceled in a form suitable for incorporation in WebCom. This packaging is in the form of modules supporting the various type checking scenarios.

Separating runtime from beforetime checking into separate modules is a convenience. So, to enable runtime checking, the runtime checker module would be loaded and informed if strict or loose verification was desired. In the case of beforetime checking, the beforetime checker could be loaded instead. This would include additional functionality to support the augmentation of graphs with verified typing information.

Both modules would use essentially similar operations. The runtime version being a microcosm of the beforetime problem as already discussed. So, while the module algorithm implementations would share a number of features, their invocation hooks and module interface forms would be quite distinct, runtime being effectively invisible, beforetime requiring explicit invocation.

The two modules ought to be similar in configuration. Both scenarios above discussed the striped down and more efficient verification at the cost of accuracy, and the implemented modules ought to allow the site administrator or graph designer flexibility as to whether strict or loose verification is desired.

The runtime checker can be invisible to the user by using the Event API. So, all an administrator or user would have to do to enable runtime checking is to load the runtime checker module. This module would then employ the Event API to spy on graph execution and node production. When a node is completed, or perhaps just before an `Instruction` is executed, the runtime checker could invoke a check on the node or operation. This monitoring is easy to arranged using the Event API adaptor classes and amounts to just a few lines in an adaptor extension and a line in the module load method registering the listener.

The beforetime checker requires more explicit invocation, due to use pattern. The perceived usage is in IDE graph design. The designer may periodically run a loose check during graph construction. This quick check might then verify typing information over the completed portions of the graph.[8] The checker algorithm might also return information relevant to typing failures, so as to help isolate areas of poor typing.

When satisfied with a graph, the designer may then invoke a more timeconsuming full verification. The intention here is that this check be used to provide preverification for the runtime execution. So, the IDE might verify a graph and if successful provide a cryptographic hash indicating successful verification. The runtime environment might demand the production of such a token before agreeing to run the graph. Of course, this outline is vulnerable to deliberately malicious signings since a tool may be used to produce verified tokens for graphs irregardless of whether type verification succeeded.

The type checking algorithms also have applications in debugging WebCom code. They may be used to check initially for ill typing, or used in conjunction with a trace replayer to examine typing in a failed run.

Both modules require `Info` object and parsed type string caches.[9] Since this is common functionality, it might implemented in a common parent class, or in a module upon which both checkers depend.

In all, the module harness designed in previous chapters provides very adequate support for implementing type checking concerns, and illustrates the Module API benefit to WebCom developers. In the case of type checking implementation, most effort was spent on designing algorithms rather than on WebCom integration.

For this reason, the work to the end of the previous chapter, which may be assumed to exist in any future development WebCom work, forms a base upon which to build applications and extend WebCom.

6.6 Finally...

§74 Further Work. *«Improvements, necessary and desired»*
Some of the many areas where this type checker outline might be improved will be mentioned briefly here. There are also some suggestions as to future work, in addition to work suggestions already commented on.

The checker algorithms, especially the full verification algorithms deserve further refinement. Central is the determination of efficiency in programming the entire verification as a single logic program. More effort is also needed in improving the suggested search strategies, especially to handle better backtracking.

The full specification of classnames in the *Types* logic is cumbersome. Type string namespace conventions might be implemented, including support for import statements in `Info` objects.

A system for dealing with functional dependencies in type string specifications should be developed, especially in the case where output types depend functionally on actual operand types. Take the addition operation, for example. The output types should be tightly expressed in terms of the widest input type, but there is no immediate method to express this in the current logic. There are approaches involving variable bindings which approximate the desired effect, but the logic should have notation to express conditionality better. Even an implication statement within the logic would suffice.

Since variable resolution introduces serious time penalties, establishing when variables are required would be very beneficial. Determining cases suitable for variable elimination is naturally heuristic in nature, at best, but the potential gains warrant an expenditure of effort in this direction.

Finally, there are problems with dynamic destinations and operations in type checking. The current checker approach requires static destinations to a large degree. With the use of dynamic operations more likely in future, there are concerns about the whether the beforetime checker routines can manage this dynamism. At very least, `Info` objects might be extended to express the dynamic possibilities.

§75 Concluding Remarks. *«A reminder of where this dissertation has been»*
Before concluding this discussion, it is worth recapping on the various applications and software developed during the course of this document. The list of developments forms an eclectic list, mostly falling under the theme of supporting the future software development within WebCom.

Toward improving WebCom developer support, metadata notations were introduced to specify details relating to internal datastructures without needing to augment these actual structures.

Aspect Oriented Programming was introduced in the guise of the Event API in order to provide third parties with event information from the WebCom core with no modification to internal WebCom structures. This event implementation was particularly designed with efficiency in mind and forms the bedrock of the runtime type checker application. Wider AOP facilities beyond those in the Event API are also available.

Much work was done on the Module API, especially in reforming the view of WebCom modules to fall in line with a completely-plugin architecture. This philosophy shift is probably the most potentially useful aspect of this work. A completely plugin view may be used to ringfence the fragile WebCom core.

Another major addition to the WebCom developer toolkit was the resolver and introduction of logic programming. The utility of logic oriented programming was illustrated in the type checker design, which might have be implemented immediately as a schema for producing logic programs for resolution.

Aside from design work, some useful applications have been developed. Included primarily to illustrate the particular features, they also form a useful collection of WebCom support software. Applications have included the J2ME submission tool, and the generic WebCom launch tool, both of which build on the Submission API. The automated documentation tools also depended on the Information Framework.

There has been the execution trace application built on the Event API and incorporated in the trace module. This complements the other modules developed as part of the SysTray application providing a desktop based WebCom system. These include the statistics, IDE bridge, BeanShell and other GUI modules.

The logic resolver is key to type checking, but is also of use in supporting potential security rule rewrites and other problems. In conjunction with logic, the SableCC generated parser ought to be mentioned, in particular because it effectively combines the Information Framework with the logic tool.

Finally, there is the type checking application itself, which although the nominal goal of this project, actually formed more of a convenient endpoint for the development journey. It is perhaps best to end by hoping these new frameworks, designs and API will serve their intended purpose of improving the capacity for third parties to implement interesting new WebCom functionality.

Chapter Notes

[1] `Info` objects and type string logic structures will typically be cached, so parse per operation inefficiencies are not always incurred.

[2] In fact, the productions are suboptimal. For instance, the variable line of the alternatives could be written as just:

`'VAR'` ⟨Variable_Name_String⟩

That is, without the parentheses symbols, since the parentheses would be parsed by the first production alternative even if they are not explicitly required. Of course, then the parentheses would be optional and the function notation not required.

[3] Note that this definition restricts the applicability of the type checking algorithms to those graphs for which these dataflow paths can be established. And specifically, there is no reason why graphs which involve dynamic destinations should be verifiably in this system.

[4] In the case of a Java implementation, this relabeling might employ the object memory reference, for instance. Of course, this definition is for description purposes and isn't suitable for a Java implementation, anyways.

[5] There would be problems with exchanging classes between WebCom instances, though. The type checker would need to know *all* the classes that may be required in the verification, whether or not they are present in the local JVM.

[6] The operators cannot be dynamic in this case, or if they are, the type must be easy to determine.

[7] This is slightly deceptive in that `Object` might appear legitimately as a substitution value, and it is undesireable to confuse these entities. However, for practical purposes this approach will work.

[8] This is how it might work in the IDE case if building a Condensed Graph in piecemeal fashion.

[9] Parsed type strings can be reused also if revariablised.

Appendix A — `NodeInfo` Example

This appendix contains the promised example `NodeInfo` source code from Chapter 2, describing a `NodeInfo` for an addition operation taking two numeric parameters.

```
package webcom.nodes.core;


import webcom.cgengine.Strictness;
import webcom.graphinfo.NodeInfo;


/**
 * GraphInfo for Addition Node
 */
public class AdditionNodeInfo extends NodeInfo
{
   /**
    * Get the name of this node
    *
    * @return the name.
    */
   public String getName()
   {  return "Addition Node";
   }


   /**
    * Get the graph name this graph info specifies.
    *
    * @return the name of the graph to execute.
    */
   public String getNodeName()
   {  return "webcom.nodes.core.AdditionNode"; //$NON-NLS-1$
   }
```

```
/**
 * Accessor for string node description
 *
 * @return the stored node description string.
 */
public String getDescription()
{   return "Add the operands.";
}

/**
 * Get the number of arguments that are specified in this graph info.
 *
 * @return the number of arguments the condensed graph requires.
 */
public int getNumArguments()
{   return 2;
}

/**
 * Return the type of the i-th argument(index origin zero).
 *
 * @param i the index of the argument to query type of. Ignored
 * @return one of Argument type values
 */
public String getArgType(final int i)
{   return "OR(java.lang.Byte, java.lang.Short, " + //$NON-NLS-1$
      "java.lang.Integer, java.lang.Long, java.lang.Float, " + //$NON-NLS-1$
      "java.lang.Double)"; //$NON-NLS-1$
}

/**
 * Return the description of the i-th argument(index origin zero).
 *
 * @param i the index of the argument to query description of
 * @return String describing i-th argument
 */
public String getArgDescription(final int i)
{   return "A summand";
}

/**
```

```
    * Return the strictness of the i-th argument(index origin zero).
    *
    * @param i the index of the argument to query type of                    80
    * @return strictness of the i-th argument.
    */
   public Strictness getArgStrictness(final int i)
   {   return Strictness.STRICT;
   }


   /**
    * Get the number of outputs that are specified in this graph info.
    *                                                                       90
    * @return the number of outputs in the condensed graph.
    */
   public int getNumOutputs()
   {   return 1;
   }


   /**
    * Return the type of the i-th output(index origin zero).
    *                                                                      100
    * @param i the index of the output to query type of. Ignored
    * @return output type string
    */
   public String getOutputType(final int i)
   {   return "OR(java.lang.Byte, java.lang.Short, " + //$NON-NLS-1$
         "java.lang.Integer, java.lang.Long, java.lang.Float, " + //$NON-NLS-1$
         "java.lang.Double)"; //$NON-NLS-1$
   }

                                                                           110
   /**
    * Return the description of the i-th output(index origin zero).
    *
    * @param i the index of the output to query description of
    * @return String describing i-th output
    */
   public String getOutputDescription(final int i)
   {   return "Result of Addition";
   }
}                                                                          120
```

Appendix B — Event Trace

This appendix contains a full listing for the operation of the event trace Event API application on the odd parity test graph, depicted in Figure 37 below.

Figure 37: Odd parity testing graph.

The detailed output from the event trace module on invoking this parity test graph within WebCom is included below. The events leading up to and including the `engineRun` event essentially track the WebCom bootstrap. Graphs are bootstrapped in WebCom by first making a top level condensed node with the graph to execute as a dynamic operator. This top level node is populated with the desired operands and pushed onto the WebCom execution queues. This will cause the node to be expanded and the desired graph to be allocated and executed. By the time the `engineRun` event occurs below, the actual graph is ready to start running.

The E node of this graph is then queued and subsequently an instruction for its operation is constructed. This instruction is then executed in the engine module, the execution of which prompts the graph allocation.

Next is the node containing the Even operator. This is queued, a containing instruction made and sent to the scheduler for scheduling. The instruction is then finally passed to the load balancer for queueing. The node's instruction ends up being passed back to the engine module for execution.

The case of the NOT node is handled similarly. However, the X node is executed directly at the engine module without being passed to the scheduler, etc. The operation of the X node prompts the graph deallocation. The RGExitNode plays a role in the implementation of the X node operations and follows the ExitNode.

Back at the top level, the backplane receives a message informing it of the result, which it can pass to the user tool used to invoke the WebCom instance, i.e., the IDE or commandline tool, etc.

The full trace of events is included overleaf.

```
newInstructionInWebCom :-
webcom.core.Instruction@1e6e305
operation main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionReceived :-
webcom.core.Instruction@1e6e305
operation main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineQueueNode :-
webcom.cgengine.Node@45e228
uid -4818b48d:1035bcff0d4:-7fac
short_name main
cg_container null
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
newInstructionInEngineModule :-
webcom.core.Instruction@2b249
operation main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
schedulerScheduleInstruction :-
webcom.core.Instruction@2b249
operation main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
loadBalancerQueueInstruction :-
webcom.core.Instruction@2b249
operation main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionExecutionInEngineModule :-
webcom.core.Instruction@2b249
operation main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionExecutionInEngineModule :-
webcom.core.Instruction@1e6e305
operation main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
backplaneSentMessage :-
EngineMessage: source = agador/143.239.211.35 module value: top level,
destination = agador/143.239.211.35 module value: engine,type = INSTRUCTION,
data = webcom.core.Instruction@1e6e305
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineRun :-
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineQueueNode :-
webcom.cgengine.Node@163956
uid -4818b48d:1035bcff0d4:-7faa
short_name webcom.cgengine.EnterOp
cg_container main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
newInstructionInEngineModule :-
webcom.core.Instruction@10e434d
operation webcom.cgengine.EnterOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionExecutionInEngineModule :-
webcom.core.Instruction@10e434d
operation webcom.cgengine.EnterOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineAllocatedGraph :-
```

```
webcom.cgengine.DynamicCG@789869
uid -4818b48d:1035bcff0d4:-7fac
full_name main(-4818b48d:1035bcff0d4:-7fac)
name null
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineQueueNode :-
webcom.cgengine.Node@c063ad
uid -4818b48d:1035bcff0d4:-7fa8
short_name webcom.nodes.core.EvenOp
cg_container main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
newInstructionInEngineModule :-
webcom.core.Instruction@9abc69
operation webcom.nodes.core.EvenOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
schedulerScheduleInstruction :-
webcom.core.Instruction@9abc69
operation webcom.nodes.core.EvenOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
loadBalancerQueueInstruction :-
webcom.core.Instruction@9abc69
operation webcom.nodes.core.EvenOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionExecutionInEngineModule :-
webcom.core.Instruction@9abc69
operation webcom.nodes.core.EvenOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineQueueNode :-
webcom.cgengine.Node@78dc4c
uid -4818b48d:1035bcff0d4:-7fa7
short_name webcom.nodes.core.NOTOp
cg_container main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
newInstructionInEngineModule :-
webcom.core.Instruction@c70b0d
operation webcom.nodes.core.NOTOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
schedulerScheduleInstruction :-
webcom.core.Instruction@c70b0d
operation webcom.nodes.core.NOTOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
loadBalancerQueueInstruction :-
webcom.core.Instruction@c70b0d
operation webcom.nodes.core.NOTOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionExecutionInEngineModule :-
webcom.core.Instruction@c70b0d
operation webcom.nodes.core.NOTOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineQueueNode :-
webcom.cgengine.Node@bef361
uid -4818b48d:1035bcff0d4:-7fa9
short_name webcom.cgengine.ExitOp
cg_container main
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
newInstructionInEngineModule :-
```

```
webcom.core.Instruction@5c98f3
operation webcom.cgengine.ExitOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionExecutionInEngineModule :-
webcom.core.Instruction@5c98f3
operation webcom.cgengine.ExitOp
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineDeallocatedGraph :-
webcom.cgengine.DynamicCG@789869
uid -4818b48d:1035bcff0d4:-7fac
full_name main(-4818b48d:1035bcff0d4:-7fac)
name null
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineQueueNode :-
webcom.core.enginemodule.RGExitNode@15e293a
uid -4818b48d:1035bcff0d4:-7fab
short_name RGOperator
cg_container null
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
newInstructionInEngineModule :-
webcom.core.Instruction@1d840cd
operation RGOperator
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
instructionExecutionInEngineModule :-
webcom.core.Instruction@1d840cd
operation RGOperator
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineSendResult :-
webcom.core.Result@2f8116
source_ref null
data true
execution_time 0
resultee null
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
backplaneSentMessage :-
EngineMessage: source = agador/143.239.211.35 module value: engine,
destination = agador/143.239.211.35 module value: top level,type = RESULT,
data = webcom.core.Result@2f8116
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
engineResultSent :-
webcom.core.Result@2f8116
source_ref webcom.cgengine.Node@1f1680f
data true
execution_time 94
resultee webcom.ide.AnywareIDE[frame0,0,0,1920x1170,invalid,
layout=java.awt.BorderLayout,title=WebCom-G IDE,resizable,normal,
defaultCloseOperation=DO_NOTHING_ON_CLOSE,
rootPane=javax.swing.JRootPane[,4,36,1912x1130,invalid,
layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0,alignmentY=0.0,border=,
flags=16777673,maximumSize=,minimumSize=,preferredSize=],
rootPaneCheckingEnabled=true]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Appendix C — Unification Algorithm

This appendix contains the unification algorithm as used in the resolver outlined in Chapter 5.

Algorithm U (l, m)

---

main
if signs, symbols or number of terms in l and m do not agree
  then return (failure)
$\theta \leftarrow \emptyset$
for each t subterm of l
do
  $s \leftarrow$ corresponding subterm of l
  comment: Apply substitution to date over both t and s.
  $t' \leftarrow \theta(t), \quad s' \leftarrow \theta(s)$
  comment: Iterate the recursion with a goal stack.
  stack $\leftarrow \{t' = s'\}$
  while stack $\neq \emptyset$
  do
    $g \leftarrow$ pop(stack)
    $u \leftarrow$ Lefthand side of g, $\quad v \leftarrow$ Righthand side of g
    if u is a variable
    then
      Apply $u \rightarrow v$ over θ and stack, occurs checking if desired.
      $\theta \leftarrow \theta \cup \{u \rightarrow v\}$
    else if v is a variable
    then
      Apply $v \rightarrow u$ over θ and stack, occurs checking if desired.
      $\theta \leftarrow \theta \cup \{v \rightarrow u\}$
    else if v and v agree on function symbol and arity
    then stack $\leftarrow$ stack $\cup \{v_i = u_i\}_i$ where v_i and u_i are the subterms of u and v.
    else return (failure)
return (θ)

Appendix D — *Types* Logic Grammar

This appendix contains the SableCC grammar for the *Types* logic parser.

```
Package webcom.typechecker.parser;

Helpers
   lf = 0x000a;
   cr = 0x000d;
   tab = 0x0009;
   printable = [32 .. 127];
   letter = ['a' .. 'z'] | ['A' .. 'Z'];
   digit = ['0' .. '9'];
   dot = '.';

Tokens
   var_token = 'VAR';
   or_token = 'OR';
   and_token = 'AND';
   not_token = 'NOT';
   open_token = '(';
   close_token = ')';

   string_token = letter (letter | digit | dot)*;
   blank = (cr | lf | tab | ' ')+;
   comma_token = ',';

Ignored Tokens
   blank;
```

```
Productions
   type =
      {parenthesis} open_token type close_token |
      {constant} string_token |
      {variable} var_token open_token string_token close_token |
      {not} not_token open_token type close_token |
      {and} and_token open_token and_list close_token |
      {or} or_token open_token or_list close_token;

   and_list =
      {single} type |
      {multiple} and_list comma_token type;

   or_list =
      {single} type |
      {multiple} or_list comma_token type;
```

Bibliography

[FMQ04] Simon N. Foley, Barry P. Mulcahy, and Thomas B. Quillinan. Dynamic administrative coalitions with webcom dac. In *WeB2004 The Third Workshop on e-Business*, Washington D.C., USA, December 2004.

[FQ02] Simon N. Foley and Thomas B. Quillinan. Using trust management to support micropayments. In *Proceedings of the Second Information Technology and Telecommunications Conference*, pages 219–223, Waterford Institute of Technology, Waterford, Ireland., October 2002. TecNet.

[FQM+00] Simon N. Foley, Thomas B. Quillinan, John P. Morrison, David A. Power, and James J. Kennedy. Exploiting keynote in webcom: Architecture neutral glue for trust management. In *Proceedings of the Nordic Workshop on Secure IT Systems Encouraging Co-operation*, Reykjavik University, Reykjavik, Iceland, October 2000.

[FQM02] Simon N. Foley, Thomas B. Quillinan, and John P. Morrison. Secure component distribution using webcom. In *Proceeding of the 17th International Conference on Information Security (IFIP/SEC 2002)*, Cairo, Egypt, May 2002.

[FQO+04] Simon N. Foley, Thomas B. Quillinan, Maeve O'Connor, Barry P. Mulcahy, and John P. Morrison. A framework for heterogeneous middleware security. In *Proceedings of the 13th International Heterogeneous Computing Workshop*, Santa Fe, New Mexico, USA., April 2004. IPDPS.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[JPMP04] David A. Power John P. Morrison, Brian Clayton and Adarsh Patil. Webcom-g: Grid enabled metacomputing. *The Journal of Neural, Parallel and Scientific Computation. Special Issue on Grid Computing.*, 2004(12)(2):419–438, April 2004.

[Ken04] James J. Kennedy. *Design and Implementation N-Tier Metacomputer with Decentralised Fault Toerance*. PhD thesis, PhD Thesis, University College Cork, Ireland, May 2004.

[MC] John P. Morrison and Ronan Connolly. Facilitating Parallel Programming in PVM using Condensed Graphs. Proceedings of EuroPVM'99: Universitat Autonoma de Barcelona, Spain. 26-29 Sept 1999.

[MKPa] John P. Morrison, James J. Kennedy, and David A. Power. A Condensed Graphs Engine to Drive Metacomputing. Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA '99), Las Vagas, Nevada, June 28 - July1, 1999.

[MKPb] John P. Morrison, James J. Kennedy, and David A. Power. Extending WebCom: A Proposed Framework for Web Based Distributed Computing. Workshop on Metacomputing Systems and Applications, ICPP2000.

[MKPc] John P. Morrison, James J. Kennedy, and David A. Power. WebCom: A Volunteer-Based Metacomputer. The Journal of Supercomputing, Volume 18(1): 47-61, January 2001.

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

Bibliography

[MOH] John P. Morrison, Padraig J. O'Dowd, and Philip D. Healy. Searching rc5 keyspaces with distributed reconfigurable hardware. ERSA 2003, Las Vegas, June 23-26, 2003.

[Mor96] John P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Eindhoven, 1996.

[MP] John P. Morrison and David A. Power. Master Promotion & Client Redirection in the WebCom System. Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2000), Las Vagas, Nevada, June 26 - 29, 2000.

[MPC] John P. Morrison, Keith Power, and Neil Cafferkey. Cyclone: Cycle Stealing System. Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2000), Las Vegas, Nevada, June 26 - 29, 2000.

[MPK] John P. Morrison, David A. Power, and James J. Kennedy. Load balancing and fault tolerance in a condensed graphs based metacomputer. The Journal of Internet Technologies. Special Issue on Web Based Computing. Volume 3(4), 221-234, December 2002.

[MRa] John P. Morrison and Martin 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.

[MRb] John P. Morrison and Martin Rem. Speculative Computing in the Condensed Graphs Machine. proceedings of IWPC'99: University of Aizu, Japan, 21-24 Sept 1999.

[QCF04] Thomas B. Quillinan, Brian C. Clayton, and Simon N. Foley. GridAdmin: Decentralising grid administration using trust management. In *Proceedings of the Third International Symposium on Parallel and Distributed Computing (ISPDC04)*, Cork, Ireland, July 2004.

[QF04] Thomas B. Quillinan and Simon N. Foley. Security in webcom: Addressing naming issues for a web services architecture. In *Proceedings of the 2004 ACM Workshop on Secure Web Services (SWS).*, Washington D.C., USA., October 2004. ACM Conference on Computer and Communications Security, ACM. To Appear.