Computational Paradigms and Protection

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

Categories and Subject Descriptors

D.4.6 **[Security and Protection]**: Access Controls; D.1 **[Programming Techniques]**: General; F.1.2 **[Modes of Computation]**: Parallelism and concurrency

Keywords

Security models; protection mechanisms; condensed graphs; imperative, functional and dataflow programming.

ABSTRACT

We investigate how protection requirements may be specified and implemented using the imperative, availability and coercion paradigms. Conventional protection mechanisms generally follow the imperative paradigm, requiring explicit and often centralized control over the sequencing and the mediation of security critical operations. This paper illustrates how casting protection in the availability and/or coercion styles provides the basis for more flexible and potentially distributed control over the sequencing and mediation of these operations.

1. INTRODUCTION

The sequencing of operations in a computation may be classified in terms of three fundamental paradigms. In the traditional *imperative* paradigm, the programmer explicitly determines the sequencing constraints of operations; in the *availability* paradigm, the sequencing of operations depends only on the availability of operand data; and, in the *coercion* paradigm, operations are executed when, and only when, their results are needed.

These paradigms can be interpreted in the context of protection. Conventional protection mechanisms generally follow the imperative paradigm by enforcing explicit mediation and sequencing on operations. For example, when mediating a purchase order transaction [order; validate; invoice; payment], an imperative protection mechanism might ensure that the operations are done in the correct sequence, and that suitable separation of duties are applied at each stage.

A weakness of the imperative approach is that protection is based on the explicit control of sequencing and mediation. Explicit control can become difficult when flexibility over the sequencing of operations is required. For example, it is often desirable to allow an un-validated order to progress through the system with the assumption that validation will be done at some stage, but before payment is made. Sometimes it may even be expedient to make payment without any validation.

While such requirements can be constructed in terms of explicit sequencing control, it may be more natural to consider the requirements in terms of the data dependencies between the operations. For example, operation pay depends on operations validate and invoice, operation validate depends on operation order, and so forth. These relationships can be expressed in terms of a graph of operations (nodes) linked together by the data that is passed between them. The implicit parallelism of operations in the graph gives rise to two different paradigms for specifying and enforcing protection requirements.

In the availability paradigm, sequencing and mediation of operations depend only on the availability of operands. This data-flow like sequencing of operations gives rise to eager execution. For example, once an order is proposed then validation and invoice processing can be done at any stage (before payment). In the coercion paradigm, sequencing and mediation of operations is determined on the basis of when results are needed. This is the functional style of operation sequencing and gives rise to lazy evaluation. For example, only if and when payment is finally required should order validation be sought.

In this paper we investigate how protection requirements can be specified and implemented using the imperative, availability and coercion paradigms. This is done using the *Condensed Graphs* model of computation [8, 11] which provides a single framework that unifies these three paradigms. In addition to providing flexibility in the sequencing and control over security critical operations, these paradigms have facilitated the development of novel distributed protection mechanisms that are also described in this paper. By following the inherently parallel availability and coercion paradigms, the proposed protection mechanisms need not necessarily rely on centralized security state.

Section 2 provides a brief outline of the Condensed Graphs model. Using the purchase order transaction as an example, Section 3 considers various sequencing constraints that one may wish to enforce. This section also serves to illustrate the notation and semantics of the Condensed Graph model. A protection model based on permissions and protection domains is described in Section 4. Specific protection mechanisms for this model are considered in Section 5.

2. CONDENSED GRAPHS

Like classical dataflow [1], the Condensed Graphs ($\mathcal{CG}$) model [8, 11] is graph-based and uses the flow of entities on arcs to trigger execution. In contrast, Condensed Graphs are directed acyclic graphs in which every node contains not only operand ports, but also an operator and a destination port. Arcs incident on these respective ports carry other Condensed Graphs representing operands, operators and destinations. Condensed Graphs are so called because their nodes may be condensations, or abstractions, of other Condensed Graphs. (Condensation is a concept used by graph theoreticians for exposing meta-level information from a graph by partitioning its vertex set, defining each subset of the partition to be a node in the condensation, and by connecting those nodes according to a well-defined rule [6].) Condensed Graphs can thus be represented by a single node (called a *condensed node*) in a graph at a higher level of abstraction.

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

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

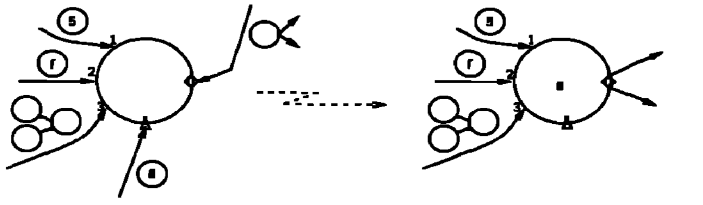

Figure 1: Condensed Graphs congregating at a node to form an instruction

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

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

A number of working prototypes that use Condensed Graphs have been developed, demonstrating it usefulness as a general model of computation. Prototypes include a sequential TM interpreter [8] and a web-based distributed computing engine [10]. WebCom [10] schedules the execution of coarse-grain computations described as Condensed Graphs. Web clients (ancillary processors) connect to a WebCom server (Triple Manager) whereupon they are served appropriate computations ($\mathcal{CG}$ operations).

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

3. OPERATION SEQUENCING

In this section we consider a variety of controls that one may wish to place on the purchase order transaction example discussed in Section 1. The examples also serve to illustrate the notation and semantics of the $\mathcal{CG}$ model.

EXAMPLE 1. Condensed node PO_I specifies the allowable behaviour for order processing (Figure 2). By definition [8],

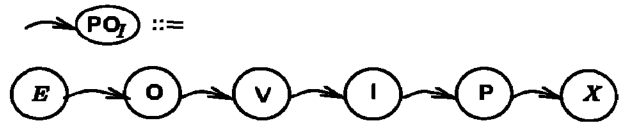

Figure 2: Imperative Style Definition of PO_I

the behaviour of a Condensed Node such as PO_I is constructed as a Condensed Graph with a single entry node (E) and single exit node (X). The other nodes in the graph represent the operations available: propose an order (O), validate the order (V), process the associated invoice (I) and make a payment (P).

Arcs represent data paths between the operations which, in this case, may fire (execute) when data arrives at their input ports. For example, when the order has been proposed a value (the details) is output from O and passed to V, which may, in turn, fire, and so forth. Firing a condensed node *evaporates* it into the graph that defines it, with input available from the E node and final output emanating from its X node.

Figure 2 may be regarded as specifying sequential ordering constraints [O;V;I;P] in an imperative style. △

EXAMPLE 2. Figure 3 specifies the purchase order transaction in a simple dataflow or availability manner. Once the order has been proposed, details are available to both V and I, which may fire in either order. Only when both inputs

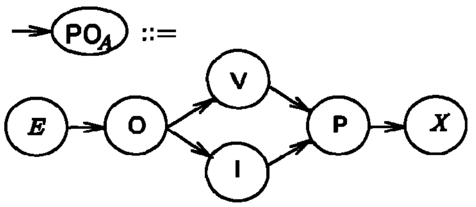

Figure 3: Availability Style Definition of $\mathbf{PO}_A$

(outputs from V and I) are available, can payment proceed. △

EXAMPLE 3. In Figure 4, orders are validated only when needed, that is, V is executed in a demand or coercion - driven manner. The V node acts as an input value to node

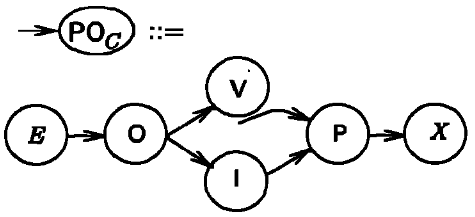

Figure 4: Demand-Driven Validation of Orders $\mathbf{PO}_C$

P, which results in V becoming fireable only when needed, as illustrated in Figure 5. If id represents a transaction identifier then $\text{id} \rightarrow \text{PO}_C$ evaporates on firing to its defining graph with id passed on from entry node E to operation O, which fires (denoted as *) generating, as output, a purchase order po (Figure 5(a)). This acts as input to V and I. Operation I fires (Figure 5(b)) since its input value is present and its output port is bound to a destination. However, while an input value is present for V, its output port is not bound to any destination, and therefore, it may not yet fire.

Once operation I has fired, operation P has values at both ports (a simple value inv, and a graph connected to node V), has an output destination, and therefore, is fireable. However, P expects the values on its input ports to be atomic values (such as po), and not an executable graph object. A 'preliminary' firing of P does not execute P, but 'grafts' node V to the input port to which it acts as a value (Figure 5(c)). As a result, P is no longer fireable; the output port of V becomes bound and fires (Figure 5(d)). As a result, operation P has atomic input values, fires, and generates a check (Figure 5(e)).

In this example, V is executed in an availability or demand-driven manner: only when a result (validation) is required, is it scheduled for execution. Execution of I may be regarded as eager, while execution of V regarded as lazy. △

EXAMPLE 4. Figure 6 specifies a variation of the purchase order graph whereby the validation requirement may be bypassed for invoices up to a certain limit. The operation lim?

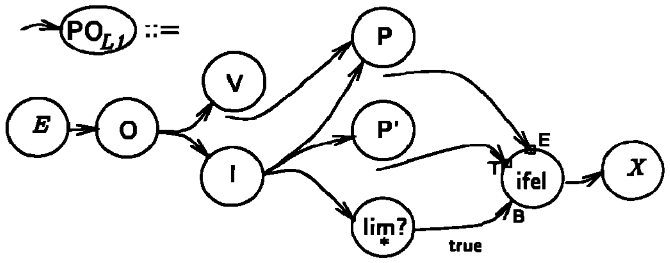

Figure 6: Lazy Order Validation $\mathbf{PO}_{L1}$.

inspects the invoice, returning true if its value is below a certain limit; otherwise it returns false. Payment operation P' has the same behaviour as P, except that it takes only one input (from I). The conditional operation ifel is provided by the Triple Manager. It takes a boolean value on its B input port, and if true then it passes the value at its T(hen) port to output, otherwise it passes the value at its E(lse) port to output.

Note that the decorations on ports T and E indicate that they are *non-strict*, that is, their input values will *not* be grafted if they are not atomic. This is unlike *strict* ports where non-atomic values are always grafted. Thus, when ifel fires the graph at its T or E port simply passes through it, depending on the value at the B port. Figure 7 illustrates part of the behaviour when lim? returns true. The input port to X is strict and the output of P' will be grafted to the input of X, making P' fireable. Note that in this case P never fires, and consequently, V never fires. If P is selected by the ifel then it becomes grafted to X and fireable (and P' never fires).

If the graph in Figure 6 had, instead, specified a direct grafted connection from outputs of P and P' to the ifel operation then then operation V may eventually fire, regardless of whether it is needed. This is analogous to a kind of speculative validation (may validate regardless), as opposed to the conservative validation (validate only if required) specified in the original graph. △

EXAMPLE 5. Non-strictness provides a degree of higher-orderedness to graphs: an operation/graph may be treated as data as it moves around the graph with execution deferred until it arrives at a strict port whereupon it becomes grafted. In Figure 8, a revised invoice-processing operation is non-

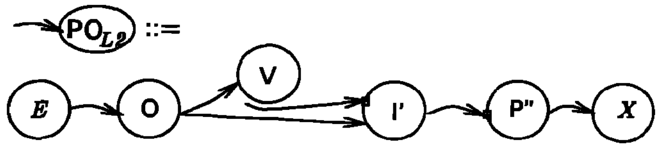

Figure 8: Lazy Validation II $\mathbf{PO}_{L2}$

strict on its 'validation' port; it checks the invoice against the order and outputs a suitable value (graph) that includes the yet-to-be executed V operation (and its inputs).

The payment operation P" also has a non-strict input port; it generates a print-check operation Ck with a dependency on the V operation (see Figure 9(a)). This graph value

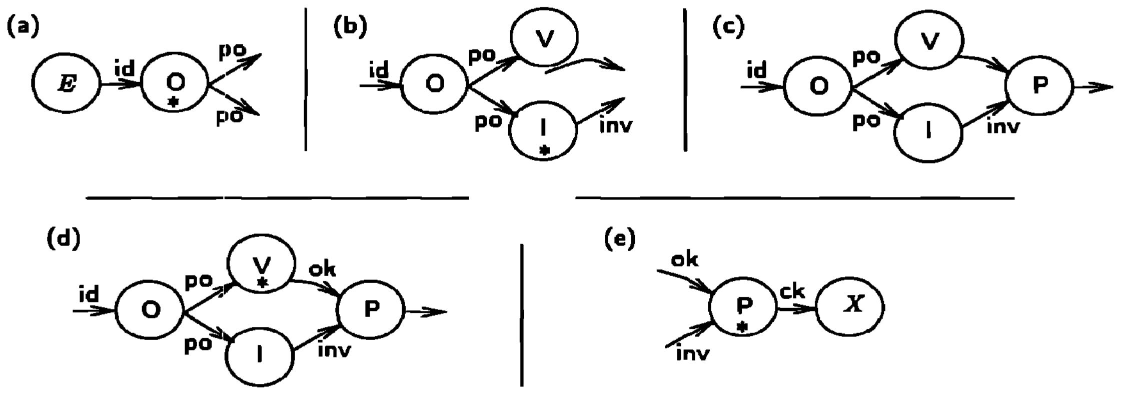

Figure 5: Firing Sequence for PO_C.

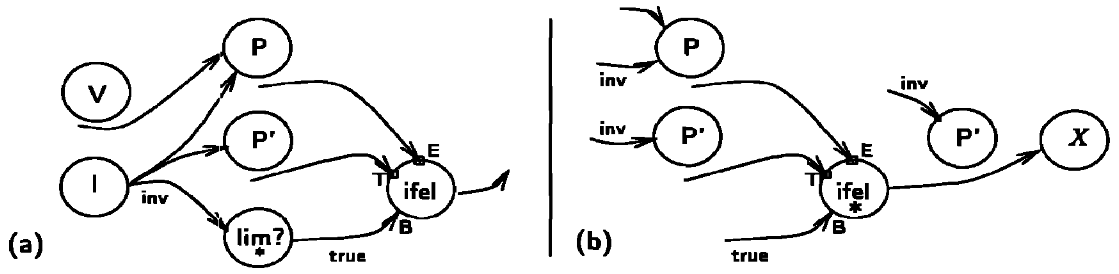

Figure 7: Snapshot of Lazy Order Validation PO_{L1}

may be thought of as representing the behaviour *"before issuing the check, validation must be done"*.

When this graph arrives at the strict port of X, it is grafted (Figure 9(b)), which in turn, results in the grafting of V (Figure 9(c)), which in turn becomes fireable. Only when validation is done, can the check be printed, and PO_{L2} completed. △

Condensed Graphs provide an executable notation that allows us to precisely specify how operations should be 'glued' together. The next section proposes a protection framework for this 'glue'.

4. PROTECTION FRAMEWORK

A Triple Manager schedules the nodes of a graph to be fired on the ancillary processors that are participating in the computation. These processors could be the components of a parallel machine, a network of workstations or a variety of heterogenous systems, connected over local networks and/or the Internet. From a security perspective, we assume that when a node fires, it does so within some *security domain*, which reflects the resources that can be accessed by the node. Thus a domain could correspond to a specific host on a network, a subnet, and so forth. However, we are not limited to a network computing model: a domain could represent a traditional protection domain [7]. For example, a node that performs a secret operation could be scheduled to domain secret. Alternatively, an authenticated domain might be represented by the public-key that speaks for it.

An operation may be scheduled to a particular security domain only if the security domain holds the correct *permission* that provides authorization to execute the operation. Each node has a permission attribute that reflects the necessary authorization (required by a domain) to execute it. The Triple Manager provides a primitive operation

```
Perm perm(NonStrict Node n);
```

where, $\mathsf{perm}(n)$ returns the permission associated with node n. Non-strictness is required since examining the permission attribute of a node should not result in its execution.

Permissions are treated as primitive value nodes within a condensed graph and are assumed to be structured as a lattice $(\mathsf{Perm}, \leq, \sqcup)$, whereby $x \leq y$ means that permission y provides no less authorization than x. A simple example is the powerset lattice of $\{\mathsf{read}, \mathsf{write}\}$, with ordering defined by subset, and lowest bound ($\sqcup$) defined by union. Thus the lowest upper bound operator may be used to compose permissions.

A Triple Manager schedules the nodes of the graph it is executing to fire in security domains that have appropriate permissions. A primitive operation is provided.

```
Perm sdom(NonStrict Node n);
```

Given node A, then $\mathsf{sdom}(A)$ returns the *permission* assigned to the domain that A is scheduled to. If A is not yet ready to fire then sdom may either return the domain planned for A or it may block until it is known and/or ready to fire. Only a single permission need be associated with each security domain since composite permissions may be constructed using $\sqcup$.

If the node A is not a primitive operation, that is, it is a condensed node, and if it is to be scheduled to the same do-

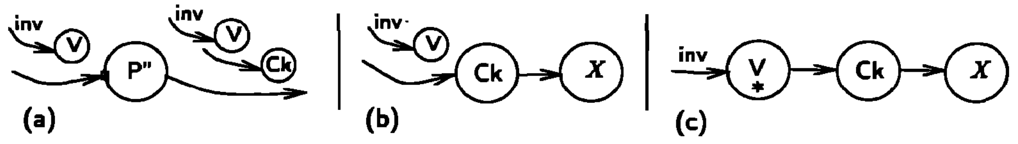

Figure 9: Strictness and Eventual Validation in PO_{L2}

main as that of the current Triple Manager, then the current Triple Manager will manage the scheduling for the graph that A defined. If A is scheduled to fire in a different domain then another Triple Manager running in this domain sdom(A) will schedule the graph that A defines. The primitive TM operation

Perm cdom();

returns the permission assigned to the domain of the graph currently executing, that is, the permission assigned to that the Triple Manager executing the graph. Figure 10 illustrates the relationship between the security-related TM primitives: a triple manager has scheduled the condensed node A (security attribute a) to be executed by another Triple Manager that is running in a domain with permission x. The graph defined by A is said to run in a security context (x,a).

The Triple Manager is regarded as a trusted component in the sense that the triples that it manages may be accessed only by the Triple Manager and that it constructs and schedules triples faithfully and according to the graph it is executing[1].

When a node with permission attribute a fires in a domain with permission x then it is said to have a *security context* (x, a). Security is defined in terms of whether a graph in one security context may schedule a node to fire in another security context (or possibly the same). A node with permission attribute b that is part of a graph with a security context (x, a) may be scheduled to a domain y if implies($(x, a), (y, b)$) holds, where implies is a partial ordering relation between security contexts. This relation, called the *scheduling constraint*, controls how graphs evaporate

We do not prescribe a specific definition for the implies relation. However, one possible definition could be based on the permission orderings.

$$\text{implies}((x,a),(y,b)) \equiv (a \leq x) \wedge (y \leq x) \wedge (b \leq y)$$

Considering Figure 10, the Triple Manager must have sufficient permissions to execute (the graph defined by) A ($a \leq x$). This Triple Manager must also have sufficient permission to schedule any node of this graph to another domain ($y \leq x$). Similarly, B must be authorized to run in this domain ($b \leq y$), and thus we have implies($(x, a), (y, b)$).

EXAMPLE 6. A Triple Manager schedules the nodes of a graph to be fired on the ancillary processors participating in the computation. Suppose that the purchase order system is implemented across a network of personal workstations connected to a trusted server.

Define the set of permissions as the powerset of {clk, mgr}, with subset as the ordering relation. Operations O and I have permission attribute {clk}; operations V and P have permission attribute {mgr}, and condensed node PO_I has permission attribute {}. Alice is a manager and is bound to permission {mgr}, while Bob, a clerk, is bound to permission {clk}.

Suppose that Alice requests that an instance of PO_I is to be executed on a trusted server (domain {clk, mgr}). This provides a context ({clk, mgr}, {}) from which the operations O, I, V and P will be scheduled. Operations O and I may be scheduled to Alice's domain (context ({clk},{}) on her workstation). Similarly, V and P may be scheduled to Bob. △

5. PROTECTION MECHANISMS

The security of a graph (based on the scheduling constraint) can be defined in an operational or denotational manner. Operationally, a graph is secure if the Triple Manager schedules only those nodes that uphold the scheduling constraint. The disadvantage of this approach is that the security mechanism must be hard-coded as part of the Triple Manager and is implementation dependent. The alternative is to define security in a denotational way, that is, define the enforcement of the scheduling constraint in terms of a Condensed Graph. We take this approach, guaranteeing that our proposed security mechanisms can be implemented, not having to worry at this stage about low-level operational details. Another advantage of defining security in this way is that we can program alternative protection mechanisms.

5.1 Fragile Protection

The *fragile* protection operation takes a node A as its input operand and if the scheduling constraint is upheld then A may fire, that is, A evaluates to A. If the scheduling constraint fails then A may not fire and the result of the evaluation is null. Figure 11 defines the operation as a condensed node. For the purposes of this

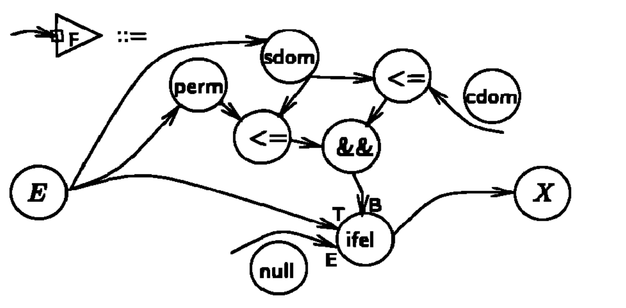

Figure 11: Definition of Fragile Protection Operator

paper we assume that the graph that is defined by

[1]We believe that assuring the correctness of the Triple Manager should be straightforward; the core of its current implementation stands at a few hundred lines of C code.

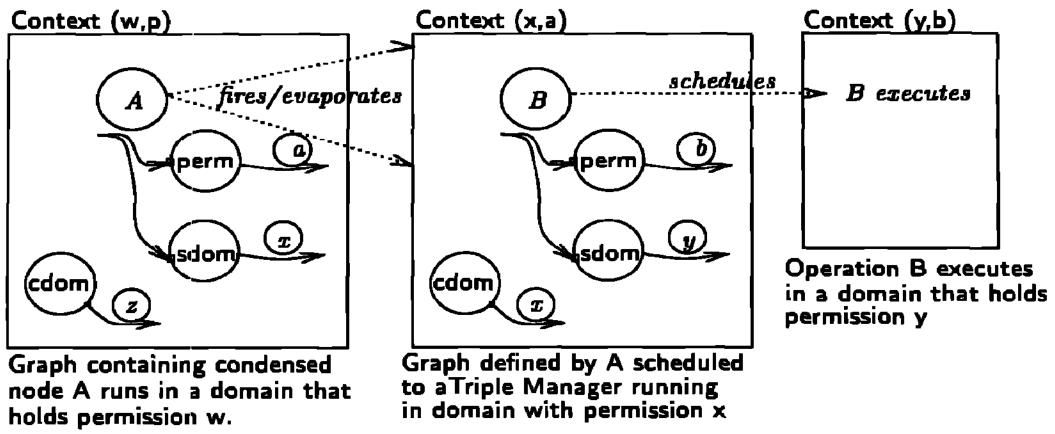
Context (w,p)
Context (x,a)
Context (y,b)
A
fires/evaporates
perm
a
sdom
x
cdom
z
B
schedules
perm
b
sdom
y
cdom
x
B executes
Operation B executes in a domain that holds permission y
Graph containing condensed node A runs in a domain that holds permission w.
Graph defined by A scheduled to aTriple Manager running in domain with permission x

Figure 10: Condensed Node B Scheduled to Fire

executes (is scheduled) in the same protection domain as its parent. This ensures that the value cdom referenced in Figure 11 corresponds to the cdom of its parent, that is, the domain that schedules the node input (A) to $\rightarrow\boxed{F}\triangleright$.

Since the input port of the fragile protection operation is non-strict, its operand A passes into its graph without grafting/firing. Lazy evaluation within the graph ensures that A passes to the X node only if it is to be scheduled to an appropriate domain, whereupon it becomes grafted to the strict port of X and fires.

Example 7. Figure 12 protects the ordering process defined in Figure 8. The *protection* nodes that protect operations O, I' and P" are immediately available to fire. Figure 13 illustrates the result of these nodes firing. An alternate firing sequence might fire the protection node of O, followed by O, and so forth. The validation operation is mediated on a lazy basis: The protected V operation (sub-graph $V\rightarrow\boxed{F}\triangleright$) passes through the I' and P" nodes. On becoming an input to the X node the protection operation becomes grafted, and fires, mediating the scheduling of V. △

In this paper we consider only the security constraints on the scheduling of nodes. How exactly a Triple Manager decides when to schedule fireable nodes must be left to the Triple Manager. It would be straightforward to implement a Triple Manager that tried to ensure that the scheduling constraint was always upheld when scheduling. In practice, we expect that a fragile protection node would be implemented as a TM primitive, rather than as a condensed node.

An implementation of the Triple Manager must also decide whether the protection operation should fire as soon as possible or whether it should wait until the node it mediates has all of its input ports bound. Immediately firing a protection node gives rise to the notion of *speculative protection*, whereby the Triple Manager schedules, in advance, an (authorized) domain for an operation before it is ready to fire. Alternatively, deferring the firing of the protection node until the operation it mediates has all of its input ports bound gives rise to *conservative protection*. Like speculative and conservative computation these can be controlled within the Triple Manager.

5.2 Tenacious Protection

The disadvantage of the fragile protection operator is that potential results are lost if the scheduling constraint is not upheld. Rather than failing, it would be preferable to reschedule the node for later evaluation, or allow it to be scheduled by another Triple Manager that has authority to assign an appropriate domain. This is achieved by the *tenacious* protection operation defined in Figure 14. Graph $A\rightarrow\boxed{T}\triangleright$ is defined recursively. If the scheduling constraint is

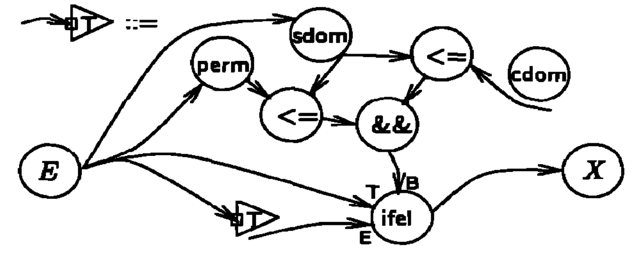

Figure 14: Definition of the Tenacious Protection Operator

upheld then node A becomes grafted to the input port of X and may be fired. If the scheduling constraint is not upheld then the result is A, lazily protected, that is, $A\rightarrow\boxed{T}\triangleright$. Unlike fragile protection, the tenacious protection operator behaves like a security wrapper that can be repeatedly probed, but can only be unwrapped (scheduled) in an authorized domain.

The tenacious protection operator could be implemented as a TM primitive. One interpretation is that the TM postpones the scheduling of a node until an authorized domain is available. However, more general interpretations are possible. For example, if the current Triple Manager cannot assign an authorized domain then the protected operation can be scheduled to another Triple Manager that can assign an authorized domain.

Example 8. Suppose that a network is partitioned in terms of a clerk subnet and a management subnet and each subnet has its own server which routes traffic to other subnets. A Triple Manager on the clerk server starts a PO

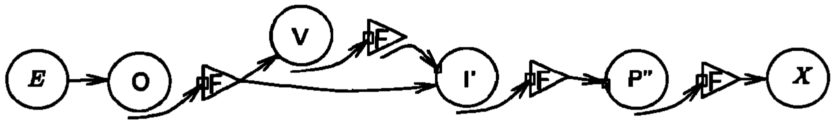

Figure 12: Protecting the Ordering Process

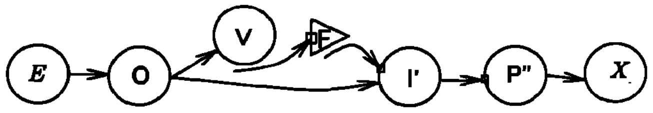

Figure 13: Protecting the Ordering Process

transaction graph, and schedules the requested O operation to a clerk's workstation. Since it cannot schedule a management V operation, it passes the 'wrapped' operation to the Triple manager on the management server for, scheduling, which in turn schedules it to an appropriate management workstation. △

Domain scheduling heuristics, such as that discussed in Example 8 should be considered part of the implementation of the Triple Manager. The tenacious protection operator could be thought of as a scout node that can be sent out across the network looking for a suitable Triple Manager to schedule the protected node. Once found, the underlying Triple Manager transparently retrieves the protected node. An alternative and speculative approach would be to multicast the protection operator across the network; as soon as one Triple Manager can schedule the protected node, the node migrates and all other speculative protection nodes are garbage collected. Low-level protocols to support, what is in effect, remote node invocation has been investigated elsewhere: a Triple Manager scheduling PVM processes [9] and a traditional dataflow system [13]. Investigating suitable domain scheduling heuristics is a topic for future research.

Example 9. Condensed Graphs are used to exploit parallelism in a computation and the Triple Manager(s) can schedule the computation across networks of workstations [10]. Figure 15 gives an example of a graph that schedules a distributed brute-force key search given known plain/cipher text. The key space is split into a series of intervals indexed as $0, 1, \ldots, maxindex$. Primitive operation

Int cr(Int interval);

searches a specified interval for the key. If found the key is returned, otherwise 0 is returned. Operation search is defined recursively and has a high degree of parallelism that can be exploited by the Triple Manager which schedules operation cr to be executed on participating processors. Operation search is passed the initial value *maxindex*.

An organization wishes to use this application to find a particular key. For the purposes of security, search operations may be scheduled only to systems within the organization intranet, while the cr operation may be scheduled to any recognized system. Figure 15 illustrates how these requirements are selectively programmed within an application. Operations search and cr are assigned permission attributes in and out, respectively. A permission may be associated with a node by introducing an additional permission input port to the node and is illustrated by using

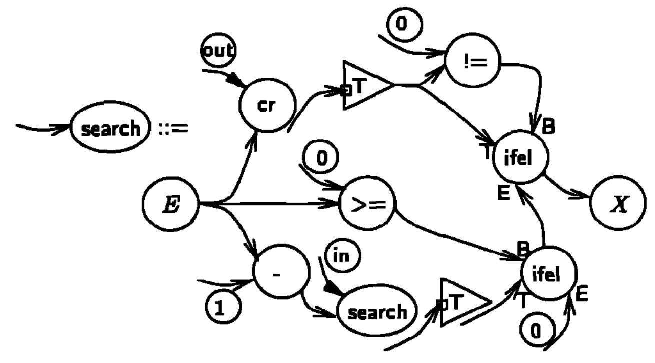

Figure 15: Programming Protection

a solid input arrow-head. Given the permission ordering (out ≤ in), then systems (domains) within the intranet are given permission in, and recognized external systems are given permission out. Schedules implies((in,in),(in,in)), implies((in,in),(out,out)) and implies((in,in),(in,out)) hold, while implies((in,in),(out,in)) does not. △

5.3 Emergent Protection

Many protection policies base access decisions on previous decisions and/or behaviour, for example Chinese Walls [3] and Dynamic Separation of Duties [12]. Condensed Graphs represent distributed computation and it is preferable not to rely upon a centralized-state approaches such as [4] to provide mechanisms that enforce these requirements.

The wrapping protection mechanism, specified in Figure 16, can be used to support, in a distributed fashion, a limited form of dynamic separation of duties. The wrapping operator W(x, A) takes as input a node A, and permits it to fire in any domain y that is strictly more authorized, or has an uncomparable authorization, to x. The result R from firing A is then 'wrapped' as W($x \sqcup y$, R). Thus, the first parameter of W is used to continue a local state (for this node) by acting as a high-water mark of the permissions of past domains.

Figure 17 gives a snapshot of this mechanism in operation. Given W(A, x) and if we have sdom(A) $\not\leq x$, then A is explicitly grafted to the second input port of a new W operation. This makes A fireable, but the non-strictness of this port of W will not graft the resulting output R. This resulting output R of A is protected and may fire only in a suitably different domain, and so forth. This wrapping operation is tenacious and is easily extended to enforce the

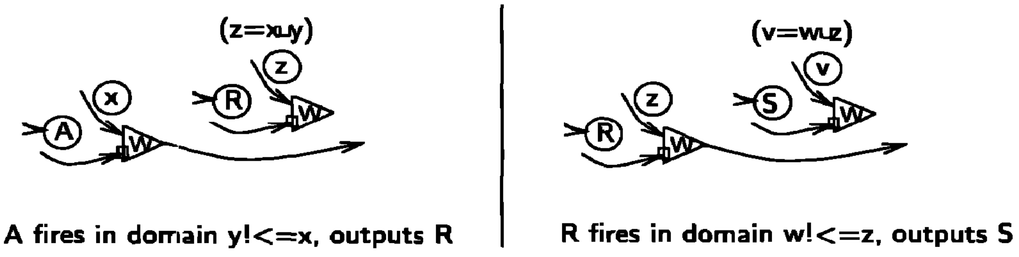
(z=x⊔y)
(v=w⊔z)
A fires in domain y!<=x, outputs R
R fires in domain w!<=z, outputs S

Figure 17: Snapshot of a Wrapping and Unwrapping

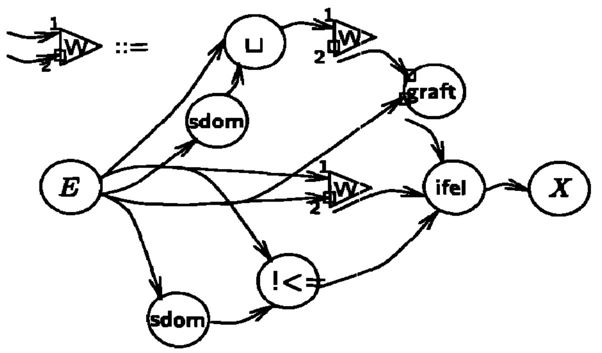

Figure 16: Wrapping Protection Mechanism

scheduling constraint.

EXAMPLE 10. Consider a simplified version of the purchase order transaction (Figure 18). The order operation takes as input an order-id, and (non-strict) a payment operation, and outputs the payment operation P appropriately transformed to include order value, and so forth. The

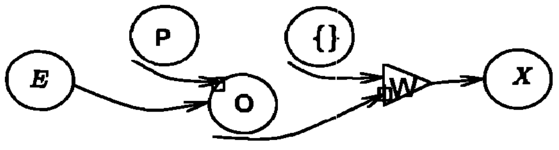

Figure 18: Dynamic Separation of Duty

order operation is mediated as W(O, {}), where {} is the empty permission. A manager (permission {mgr}) may execute the O operation, and the result is the wrapped node W(P, {mgr}). Payment P may now fire only in domains with permissions {clk} or {clk,mgr}. △

While tailored to a specific requirement, the proposed wrapping mechanism illustrates the flexibility in using Condensed Graphs to specify (and implement) protection requirements. Rather than maintaining a centralized security state, the operator W can be thought of as providing *emergent* protection: mediation results in the emergence of a further protection mechanism to mediate a subsequent operation. Investigating how this mechanism might be applied in practice and developing emergent mechanisms for general protection policies is a topic for future research.

6. DISCUSSION AND CONCLUSION

The Condensed Graphs model provides a single framework in which protection requirements can be specified and implemented within the imperative, availability and coercion paradigms. Section 3 described how combining these paradigms provide flexibility in the sequencing and control over security critical operations. Sections 4 and 5 draw on these techniques and develop novel protection mechanisms. A Tenaciously protected node (operation or data) can be repeatedly probed, and passed around, but may only be unwrapped in the appropriate domain. Referential transparency in the Condensed Graphs model means that this tenacity may be further applied to the results generated by an operation which emerge protected by a mechanism created on the fly.

Triple Managers transparently schedule graph operations to appropriate security domains. This allows protection requirements to be coded as part of the graph program, independently of the underlying architecture. Graph-based protection operators such as tenacious protection can be viewed as a protection wrapper that may be unwrapped only in an authorized security domain. Scheduling a tenaciously protected node to an authorized domain is completely transparent, even though it may have been necessary to migrate the protected node through a number of Triple Managers before it could be successfully scheduled.

Secure WebCom [5] provides one possible implementation of the protection model described in this paper. WebCom [10] Masters schedule Condensed Graph applications over remote WebCom clients (ancillary processors). WebCom Masters use KeyNote credentials [2] to determine the operations that the client is authorized to execute; WebCom master credentials are used by clients to determine if the master had the authorization to schedule the (trusted) mobile-computation that the client is about execute. This implementation can be interpreted in terms of the protection mechanisms described in this paper. Client and Master public keys provide security domains, while credentials define their associated permissions. The authorization check is similar to a fragile mediation on every node in the graph.

Much work remains to be done investigating how the protection model described in this paper might be used in practice. The protection model might also be used as part of a conventional secure system. A Condensed Graph can be regarded as a sophisticated job-control language used to schedule operations, such as multilevel transactions, to the protection domains of a separation kernel [14].

Acknowledgments

Thanks to the anonymous referees and the Workshop audience for their useful comments on this paper. This research was supported in part by Enterprise Ireland National Software Directorate.

References

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

[2] Blaze, M., et al. The keynote trust-management system version 2. Internet Request For Comments 2704.

[3] Brewer, D., and Nash, M. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy* (May 1989), IEEE Computer Society Press, pp. 206–214.

[4] Foley, S. The specification and implementation of commercial security requirements including dynamic segregation of duties. In *4th ACM Conference on Computer and Communications Security* (1997), ACM Press.

[5] Foley, S., Quillinan, T., Morrison, J., Power, D., and Kennedy, J. Exploiting KeyNote in WebCom: Architecture neutral glue for trust management. In *Fifth Nordic Workshop on Secure IT Systems* (Reykjavik, Iceland, Oct 2001).

[6] Harary, F., Norman, R., and Cartwright, D. Structural models: An introduction to the theory of directed graphs. John Wiley and Sons,1969.

[7] Lampson, B. Protection. *ACM Operating Systems Review 8* (1974).

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

[9] Morrison, J., and Connolly, R. Facilitating Parallel Programming in PVM using Condensed Graphs. Proceedings of EuroPVM'99: Universitat Autonoma de Barcelona, Spain. 26-29 Sept 1999.

[10] Morrison, J., Power, D., and Kennedy, J. 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.

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

[12] Nash, M., and Poland, K. Some conundrums concerning separation of duty. In *Proceedings of the Symposium on Security and Privacy* (Oakland, CA, May 1990), IEEE Computer Society Press, pp. 201–207.

[13] R.D. Blumofe, P. L. Adaptive and reliable parallel computing on networks of workstations. *Proceedings of the USENIX 1997 Annual Technical Symposium* (January 1997).

[14] Rushby, J. M. The design and verification of secure systems. In *Proceedings 8th ACM Symposium on Operating System Principles* (Dec. 1981). Available as ACM Operating Systems Review **15** 5.