ECLIPS, a distributed data space with induced structure and capability-based protection*

Erik Boasson

May 25, 2010

Abstract

ECLIPS is a new asynchronous distributed shared data space, integrating capability-based protection of the data space. At its core it is a descendant of SPLICE, and thus also a relative of the OMG DDS specification, and retains its important features—performance, fault tolerance, application extensibility, the simplicity of having but few operations—while bringing a novel approach to many areas, in particular by controlling the data space through its contents and its reliance on the structure of the data space to control data visibility.

1 Introduction

This article proposes ECLIPS, a design for an asynchronously updated, selectively replicated, distributed shared data space with eventual consistency semantics. The core concept is the same as that of the SPLICE architecture[1] [9], and it is designed to support the same general class of near real-time, fault-tolerant, extensible systems. Beyond the semantics of the basic read and write operations, however, the two are quite different.

The main innovations are: structuring of the data space so that data is no longer automatically globally accessible and enabling confinement of subsystems; application control over the retention of *all* data, including that stored on behalf of future subscribers; improved semantics for deleting data; integration of synchronous operations; and controlling the data space structure with its contents. The latter originates in the observation that the data space is expressive enough for describing its structure—indeed, the various SPLICE descendants all reflect the structure of the data space as data in it—and that the required operations on the data space structure can all be described in terms of the existing operations on its contents.

The environment targeted by ECLIPS is a local area network of modest size; for example a Gigabit Ethernet network with a few hundred network nodes. While this environment is currently not very fashionable, with the tremendous interest in wide-area networks, cloud computing and mobile networking technologies, it is clear that many real systems will continue to be designed for such

*Submitted to ACM TOCS, 2010-05-25

[1] First description of the concept in a company internal publication of Hollandse Signaalapparaten B.V., The Netherlands: Memo SEAT 362, 29 February 1984; first public description in 1986 in talks at New York University, Philips Research U.S.A. and G.E. Research.

an environment. It also appears unlikely that systems for which correct operation and availability is paramount, such as air-traffic control systems, will ever switch to mobile networks. So there is good cause for considering what one might do on such a network without caring much about whether it will scale to a few million nodes or work well over less reliable channels.

ECLIPS, like SPLICE, has been conceived as a peer-to-peer system with replicated storage of the data space contents, but there is no requirement that it actually be implemented that way, nor that data in the data space is stored near (e.g., replicated on the network hosts) where application processes are being executed. Even a trivial implementation with a single server (akin to a central RDBMS serving many clients) is a valid implementation, albeit with non-functional properties very different from those of a fully distributed implementation, such as relatively poor fault-tolerance, but probably with better performance of, e.g., synchronous operations.

For the purposes of this article, the OMG DDS specification [25] is considered a conceptual descendant of SPLICE, even though it also contains features inherited from the publish–subscribe message queueing systems, in particular Real-Time Innovations' NDDS 3.x product, that it also derives from. SPLICE, too, has often been described as a publish–subscribe system because of its notion of a "subscription", but it—and DDS when used wisely—is better viewed as a shared data space where "subscriptions" are used to control replication of data. This is true for ECLIPS as well.

After a short overview of SPLICE and DDS, this article first discusses the objectives, adumbrating aspects of the design; continues with an overview of ECLIPS wherein the terminology is introduced; and then deals with the details: data space structure, its representation and configuration, protection of the data space, the various components of the data sorts that form the type system, and the operations. It then concludes with a discussion and outline of further work. An appendix provides an example: an implementation of SPLICE in terms of ECLIPS.

2 Intermezzo: SPLICE and DDS

This section is a short overview of the relevant parts of SPLICE and of DDS, using a fairly generic terminology because they do not share one. Furthermore, it should be noted that while ECLIPS is very nearly a strict superset of SPLICE and supports most of the DDS specification, quite a few of the more esoteric features of the latter do not map onto it very well. Fortunately, history has proved SPLICE to be sufficiently expressive for practical use [33].

Processes subscribe to, and/or publish, typed and tagged data, such that a published datum is only visible to a subscriber if it is of a subscribed-to type and tag ("sort" in SPLICE and ECLIPS, "topic" in DDS). For each subscription taken, a local store is maintained with published data, and the data are keyed so that on the one hand multiple instances of a single type can coexist if they have different key values, but that on the other hand a datum can be overwritten be a more recent one with the same key value. It is from this local store that the subscriber reads or discards data. This is the first essential difference with publish-subscribe messaging systems, where the subscriber simply reads messages from a queue.

In DDS, publishing data of a topic requires instantiating a "DataWriter" for that topic, and taking a subscription to a topic requires instantiating a "DataReader" for that topic. DDS has many quality-of-service attributes associated with DataReaders that affect the configuration of the local stores, some of which are relevant only when using it as a publish–subscribe messaging system.

For a type marked "context" in SPLICE, or a datum published with a "transient durability" quality-of-service in DDS, the system acts as if special subscriptions to it exist regardless of whether any application process is actually subscribing to it. Each time a new subscription is taken, its associated local store receives copies, "back issues" as it were, of the data stored on behalf of those special subscriptions, if any. The effect is that the local store associated with a subscription becomes a local replica of a shared data space. This is the second essential difference with publish–subscribe messaging systems.

The equivalence between SPLICE (and DDS) and a shared data space as it is commonly understood (e.g., Linda [13]) is tenuous, in that it provides operations that break the equivalence. In particular, any operation that tests for the absence of data is problematic [7], and the read operation of SPLICE (and DDS) is both non-blocking and returns a set of values satisfying a given predicate. But, the blocking operation (wait) and use of suitable predicates easily resolves this when desired. One could also liken SPLICE to a non-coherent distributed shared memory, but, like Linda, it is content addressable, whereas memory systems traditionally are not.

Both SPLICE and DDS extend the basic model by adding a second tag to the publications, subscriptions and data. In DDS, this second tag is a set of names, called "partitions" ("worlds" in SPLICE), and a publisher and subscriber must have a partition in common for communication to occur. These names may contain wildcards.

Also in DDS is the notion of a "domain", which in essence is a completely separate DDS system, with its own topics and partitions. Every topic, DataReader and DataWriter is bound to exactly one domain, and cross-domain communication is only possible by creating a process that participates in multiple domains and explicitly re-publishes data it reads from one domain in another domain.

3 Objectives of ECLIPS

Two primary objectives have provided direction in designing ECLIPS: enabling coexistence and cooperation between mutually distrustful subsystems, and laying a foundation for merging two live systems into one while leaving the level of integration of the two to the application designer. Secondary objectives are to improve over SPLICE and DDS in various areas, particularly including the semantics for deleting data, the mechanisms used for making data available to new subscribers, and integrating a synchronisation operation in the model.

3.1 Cooperating mutually distrustful subsystems

Any number of examples of subsystems that need to communicate but do not fully trust each other can be given—an obvious example is that of military cooperation within multi-national coalitions, another is embedding subsystems from untrusted third parties containing intellectual property that its supplier

wishes to keep secret. Supporting such cases is not really the same as protecting the data space from unauthorised access, but the two are clearly related.

Considering DDS systems in particular, several implementations provide access control at the level of topics and partitions [27] or domains [26, 21]. This is *the* standard protection scheme applied to a shared data space, and should work fine for many cases. What it does not provide is a generic method for confining an arbitrary subsystem so that the subsystem can neither eavesdrop on the rest of the system nor leak information.

This latter case is one that capability-based [11] protection schemes do address. Capabilities have formed the basis for several single-address space (operating) systems (see [20] for early examples, or more recently, e.g., [14, 31]), yet capability-based *distributed* operating systems have been rather rare, although the idea is old [12] and the Amoeba system [24] is quite well known.

ECLIPS is an attempt at constructing a capability-based distributed shared data space. Rather than handling capabilities through a separate interface, it treats them as regular data, relying on strong typing of the data space to allow processes to be restricted in the passing of capabilities. Section 7 explains how the various features combine to offer a protected shared data space—confinement of subsystems, access control to named partitions similar to DDS and revocation of access. ECLIPS capabilities are *private* names, allowing the renaming of entities without affecting the processes, which proves very useful for merging live systems.

Note that many aspects of secure computing systems [2] are not explicitly addressed by ECLIPS, and that it clearly cannot be considered a secure system as-is. However, it does provide access control and addresses confidentiality within the confines of the data space, which are arguably the two most fundamental aspects: authentication of processes can be handled by the environment in which ECLIPS operates, such as an operating system; and cryptographical techniques can be applied at application level to ensure data integrity and non-repudiability, with key management either within or outside the data space provided by ECLIPS.

3.2 Merging live systems

Situations exist in which it is desirable to combine two live systems into one: a common one is the restoration of a single system after a network failure has caused the system to split into independently operating fragments. However, the systems to be merged need not always have a common ancestor system: an obvious candidate for this capability is network-centric warfare [1], where participating systems may join at any time.

In the first case, the so-called "split-brain syndrome", there will generally be a large overlap in processes and global states of the fragments. Superfluous processes can simply be killed, but the global states cannot simply be merged because the two are likely to have diverged since the split. Consider, e.g., a surveillance system that automatically tracks the objects it detects: an object detected after the split may be named differently in the two fragments, while two different objects may have received the same name in the two fragments. Solving this requires domain and application knowledge, and thus all that can be done is merging the two into a single name space—letting the application control the generation of a unified system state.

In the second case, the systems may well use identical names for different things, and in the merged system, these need to be kept separate. On the other hand, equivalent things may have different names. Clearly then, a simple namespace convention (e.g., as used for packages in Java [23]) does not suffice. Renaming things on the fly is not always a viable option either, as processes may have names stored in private state. ECLIPS addresses this problem by reducing the set of shared names to types, and these are either identical or not.

ECLIPS' particular use of capabilities allows renaming all entities in a system without affecting the processes, as processes refer to global objects using local names (the capabilities).

3.3 Deleting data

Removal of a datum from the shared data space in SPLICE, and also in DDS and in ECLIPS, happens when all subscribers remove it from their local stores, but care needs to be taken to ensure that they can all succeed in doing so, even in the presence of multiple publishers. In DDS, a publisher can indicate that a particular key value may be deleted, using an operation called *dispose*. This operation marks all data with that key value in the subscribers' local stores as having been disposed of, inserting a fictitious "invalid" datum if none exists. Each time a key value is disposed of and reintroduced, a new *generation* of that key value is introduced, but the integer identifying the generation is generated locally at the subscribers, and therefore has no global meaning. Subscribers read both validity and generation along with a datum.

This mechanism is of doubtful value: firstly, the notion of an "invalid" datum requires each program to verify the "validity" of each datum it reads from the data space before using it; and secondly, reusing the key value of a datum x remains problematic (despite the generation tags) if another datum y may refer to x by its key: then disposing of x and publishing x', a new datum with the same key, may cause y to suddenly refer to x'—a classical aliasing problem, and the generation cannot be used for disambiguation because it is local to the subscriber. Consequently x' requires a new key value to be used and the generation may be ignored.

A different approach is taken in ECLIPS: it requires the application to properly account in its data model for that which DDS attempts to address using the dispose operation—that a particular key value may now be deleted locally. ECLIPS automatically discards data to prevent overwriting of such a datum or reinstantiating of the key value after local removal because of re-ordering or concurrent publication of instances. This imposes a restriction on the reuse of key values, but so too does avoiding aliasing in references.

3.4 Availability of data to new subscribers

One aspect of a shared data space is that processes joining the data space have access to its current contents. SPLICE, DDS and ECLIPS are all based on selective replication, where data would be unavailable to processes that have not declared interest, but for the presence of a mechanism for making "old" data available to a new subscriber.

In DDS, a publisher can mark data it publishes as requiring storage independent of application interest by setting the "durability" to (the rather odd

term) “transient”, and a process can indicate that it wishes to receive copies of such data when subscribing. The independent storage is provided by a generic service, without any application or domain knowledge and which therefore necessarily relies entirely on the dispose operation for removing data. SPLICE is only slightly different, exhibiting very similar behaviour for “context” data types without giving control to individual publishers or subscribers.

Contrastingly, in ECLIPS all this is a consequence of the topology of the data space; and instead of providing a generic service it allows application processes to store the data, so that much more sophisticated data management becomes possible. ECLIPS also introduces an invariant relating the contents of the local stores of subscriptions, from which data availability follows, instead of the operational description of SPLICE and DDS.

3.5 Synchronisation

SPLICE and DDS do not have a real synchronising primitive, yet the systems built using them often require that only a single instance of a particular program may be active at any time—programs implementing name services, sensor data fusion programs, &c.—with stringent requirements on the time it takes to recover from faults such as network failures and abnormal program terminations. In an asynchronous system with very few guarantees about data ordering, this is a source of complexity and likely also of latent bugs. (Note that while DDS does have a notion of “exclusive ownership” of a datum, it is based on local decisions and does not guarantee that the same process is considered the owner of that datum throughout the system at all times.)

Addition of a (set of) well-chosen primitive(s) that can guarantee mutual exclusion and visibility of data would dramatically reduce the complexity of such applications, at the cost of increasing the complexity of the data space. It is the author’s belief that the proper place for such complexity is the data space itself. ECLIPS introduces such a mechanism.

4 Overview of ECLIPS

ECLIPS is designed to facilitate the construction of distributed *applications* consisting of a number of *processes* executing on a network of *hosts* through a set of operations from which a typed data space emerges. This data space is not stored in any one location but rather distributed over asynchronously updated local stores. Processes do not directly operate on the data space, instead a process relies on a *herald* that performs the operations on its behalf, for which the herald maintains additional internal state on the data space and communicates with its peer heralds. The latency of the network connecting the heralds is assumed to have a known upper limit.

Even though ECLIPS is expressive enough to support all direct communication between application processes [32], I/O does happen outside its scope and if the system incorporates both actuators and sensors, it may well be that the environment itself is an indirect means of communication. This article is not concerned with such alternative means of communication, nor with their effects on the properties of the system.

In ECLIPS, processes can publish data at *nablas* and declare an interest in receiving copies of such published data at *deltas*; these copies are used to update the *local stores* of the deltas, which can be queried by processes to obtain (copies of) the data. The nablas and deltas are vertices in an application-controlled dynamic directed graph that governs the flow of information: copies of published data are presented only to deltas reachable from the nabla. A process can have as many nablas and deltas as it requires.

The data space is partitioned into an extensible set of *(data) sorts*, where a sort is primarily a combination of a type and a key. Sorts have a number of other attributes that will be discussed in due course. Each nabla and each delta refers to a single sort, and a delta and a nabla must both refer to the same sort—not just to the same type—for communication to be possible. Note that this is in addition to the requirement that the delta be reachable from the nabla.

A process may be considered a publisher of a data sort when it publishes data using a nabla for that sort, and similarly it may be considered a subscriber to a data sort when it has access to a delta for that sort; this in turn suggests nablas could be called "publications" and deltas "subscriptions", as is the case in, e.g., SPLICE. However, in the more complex model of ECLIPS, the common meanings of these words introduce ambiguities that are best avoided.

The key of a data sort is a projection operator that, given an instance of the type, yields the subtype discriminating between the different *objects* of the sort that can be instantiated. (Note the similarity between a relational database table where the primary key discriminates between records, and a sort of which the type is a record and the key selects an attribute of the record.)

A datum in the data space is an *instance* of a sort and it describes the state of the object identified by the key. Upon reception by a delta it may either replace the instance describing that object if it is already in the local store of the delta (rather like updating a row in a relational table), or else cause the instantiation of a new object in the store (akin to adding a row). For linguistic convenience, "update" is sometimes used in lieu of "instance".

Note that this use of the terms "object" and "instance" follows the tradition of SPLICE; but that DDS uses "instance" and "sample", respectively, for what are essentially the same concepts. The author acknowledges that this is a likely source of confusion for those that know DDS, but having to choose, he has opted for the terms best corresponding to his construal of the system.

For a delta v for a sort, a *context* is defined as a specific subset of the deltas for the same sort from which v is reachable. (A precise definition is given in section 5, but the details are best omitted in this overview.) ECLIPS maintains an invariant relating the contents of the local store of a delta to the contents of the local stores of the deltas that constitute its context. In very simple cases this simplifies to the local store of v eventually containing the contents of the local stores of its context.

Evidently, the notion of a context in ECLIPS is closely related to that of context data in SPLICE (or of "transient durability" in DDS), yet it is fundamentally different: where all its predecessors consider it an attribute of a data sort, in ECLIPS it emerges from the structure of the data space.

The operation of publishing a datum is called *write* (a restricted variant is called *forward*), that of querying a local store and obtaining copies of the matching objects is called *read*; the matched objects can in addition be removed

from the store, this is called *local-take*. It cannot be overemphasised that data is stored in *local* stores, and that removal using local-take removes data only from the store operated on.

An object is either *current* or *obsolete*, a distinction made by evaluating the *obsoleteness predicate* on the state of the object; this predicate is defined as part of the sort. An obsolete instance represents a final state of an object. Typically a process will remove such an obsolete object from the local store of the delta using local-take; the process may, of course, still process the removed object, for example to remove internal state associated with it.

Arrival of an obsolete instance at a delta always results in updating the corresponding object in the local store of that delta if that copy of the object is current, thereby rendering it obsolete; contrastingly, a current instance never updates an obsolete object, ensuring that once an object becomes obsolete, it remains so. Two obsolete instances of the same sort and with the same key value—i.e., describing the state of the same object—are considered to be the same by ECLIPS (see section 8.6), and thus an obsolete instance will never update an obsolete object.

Furthermore, even after removing an obsolete object from the local store of a delta using local-take will arriving instances be ignored for the time needed by the application to fully process the obsolete data. This period is specified by the application.

Regular sorts are sorts for which the application has (almost) complete freedom in its choice of key and of which new objects are instantiated simply by publishing an instance with a unique key value. For such regular sorts, obsolete objects are kept track of by deltas for a minimum amount of time, in part defined by the sort and in part by ECLIPS to cover internal and network latencies, and it is therefore possible to mark an object obsolete, have every subscriber locally remove it and eventually "resurrect" the object. Once the need to retain state for an obsolete object disappears, a delta will eventually discard that state, but since it is unspecified when exactly it does so, there is no guarantee that all deltas will behave the same when attempting to resurrect an object.

An alternative to regular sorts exists in the form of *controlled* sorts, for which the key is restricted to a specific type, the *unique id* type, and of which objects can only be instantiated explicitly and only once during the lifetime of the system. Publishing an instance of a controlled sort requires the object to be known to the herald of the publishing process and instantiation is arranged for by explicitly registering a new unique id prior to the write operation that will instantiate the object. These restrictions allow ECLIPS to guarantee that an obsolete object of a controlled sort can never be resurrected.

Presence of an object of a controlled sort in the local store of a delta represents a *capability*, a right to perform certain operations on the data space. Section 7 discusses objects as capabilities.

Essentially the regular sorts blacklist obsolete objects, whereas the controlled sorts have a whitelist of current objects. Sometimes one will be more suitable, sometimes the other.

Objects of controlled sorts may be labelled using the *label* operation, an atomic store to a designated *label* field of the object of either a process identifier, VOID (the absence of a label) or DELETE (causing the object to be deleted from the system). An object labelled with a process id can be updated and relabelled only by that process. Note the exceptional nature of the label operation: it is

a (generally) *very* costly, atomic operation on *all* copies of the object, not just on those in local stores of deltas reachable from some point in the graph, and it locks out all but one process from updating the object. This is in stark contrast to the other operations which are designed to operate efficiently with only minimal coupling between processes.

The process identifier must refer to an existing process and termination of a process causes the label of all objects labelled with that process to become a *zombie label*. Zombie labels differ from normal labels in that they can be changed by any process, but such a change requires the use of a special mode of the label operation.

One possible use of labelling is implementing token passing. Many applications of token passing require visibility of data published by a token holder prior to another process becoming token holder. ECLIPS supports this model by providing a mechanism for ensuring data published prior to a label operation becomes visible before either the new label or any data published after the label operation become visible.

This mechanism is based on the notion of *sequencing objects*, a role that any object of a controlled sort may be called on to perform, simply by setting the "sequencing object" attribute of a nabla to that object's unique id. This does not affect the object itself, it is merely used to identify the nablas for which the ordering property of the label operation holds.

5 Structure of the data space

As mentioned in the overview, the data space is partitioned into an extensible set of sorts, of which instances may be published to be stored in the local stores of deltas. The nablas and deltas are vertices in a directed graph and reachability in this graph determines which deltas receive data from which nablas—always under the condition that the sorts match. The application completely controls this directed graph through the publication and the removal of objects of predefined sorts—one for nablas, one for deltas, one for neutral vertices, and one for edges. The neutral vertices are simply vertices in the graph without any behaviour.

The context of a delta has been introduced informally in the overview and is defined formally in this section. The invariant relating the contents of the local stores of the context of a delta to that of the local store of the delta itself is also defined.

5.1 Notation

Let the directed graph manipulated by the application be $G = (V, E)$ with vertices V and edges E. Each vertex in V represents either a nabla for a sort σ, a delta for a sort σ, or a neutral vertex. Let $\nabla_\sigma(X)$ be a function that selects all vertices representing nablas for sort σ from the set of vertices X, and let $\Delta_\sigma(X)$ be a similar function selecting all vertices representing a delta for σ. Let the sort operated on by a nabla or delta v, be denoted by σ_v.

An edge can be labelled with a predicate over sorts, with an unlabelled edge implicitly labelled with the predicate TRUE, such that an edge is only traversable by instances of sorts that satisfy the predicate. With E the set of all edges in

G, we denote with $E_\tau \subseteq E$ the subset of all edges traversable by instances of sort τ.

An acyclic directed graph $G'_\tau = (V', E')$ is derived from (V, E_τ) by contracting all strongly connected components. Each G'_τ has associated with it the mapping functions $M'_\tau(v)$ and $M_\tau(v')$, which map a vertex in the original graph G to the corresponding one in G'_τ and a vertex in G'_τ to the set of vertices of G it represents, respectively. Define $\Delta'_\sigma(X'_\tau) \equiv \{x' \mid x' \in X'_\tau \wedge \Delta_\sigma(M_\tau(x')) \neq \emptyset\}$.

A path p through the graph G is a sequence of vertices $\{p_1, \ldots, p_n\} \subseteq V$, where all edges $(p_i, p_{i+1}) \in E_\tau$. Let $\Delta_\sigma(p) \equiv \Delta_\sigma(\{p_i \mid i \in 1 \ldots n\})$, that is, applying Δ_σ to a path corresponds to applying it to the set of vertices visited by the path, and let $\Delta'_\sigma(p)$ be defined analogously.

The $\rightsquigarrow_{\tau,p}$ operator determines whether a sequence of vertices p is a path through G for instances of τ: $v \rightsquigarrow_{\tau,p} w \equiv p_1 = v \wedge p_{|p|} = w \wedge \forall i((p_i, p_{i+1}) \in E_\tau))$, that is, true iff p starts in v and ends at w and all steps in p correspond to τ-traversable edges in the graph; $v \rightsquigarrow_\tau w$ is shorthand for $\exists p(v \rightsquigarrow_{\tau,p} w)$. If $v \rightsquigarrow_\tau w$, w is said to be *τ-reachable* from v, often abbreviated to a plain *reachable*.

Of particular importance are paths from a delta for a sort that do not pass through other deltas for the same sort (though they may end at one): $v \longrightarrow_p w \equiv v \rightsquigarrow_{(\sigma_v),p} w \wedge \Delta_{(\sigma_v)}(p) \subseteq \{v, w\}$. Again, $v \longrightarrow w$ is shorthand for $\exists p(v \longrightarrow_p w)$. Note that the delta at which the path starts provides the sort, obviating the need for an extra index.

The operators $\rightsquigarrow'_{\tau,p}$ and $\longrightarrow'_p$ are defined analogously for G'. The forms $(v \rightsquigarrow_\tau)$ and $(\rightsquigarrow_\tau w)$ are predicates parametrised with a vertex, a destination and a source, respectively; $(v \not\rightsquigarrow_\tau w)$ is shorthand for $\neg(v \rightsquigarrow_\tau w)$. The same holds for all similar cases.

5.2 Publishing data

The write p, x operation publishes the instance x of sort σ_p at the nabla p, which causes the local stores of all deltas

$$\{v \mid v \in \Delta_{\sigma_p}(V) \wedge p \rightsquigarrow_{(\sigma_p)} v\}$$

to eventually be updated with x. There exist circumstances that may prevent an instance from updating the state of a local store, these are given in detail in section 8.6.

5.3 Context

The context of a delta for σ is defined in terms of the graph G'_σ, obtained by contracting all strongly connected components. All deltas $\Delta_\sigma(M_\sigma(v'))$ for a given $v' \in V'$ are considered equivalent.

Besides introducing a notion of equivalence that very naturally leads to a notion of redundant storage and ultimately to fault-tolerance, the use of strongly connected components also allows a neutral vertex to be used to represent such a component. In this way, context data can be provided by any of the deltas in a strongly connected component without requiring application processes to have detailed knowledge of the graph structure for publishing data or subscribing to it.

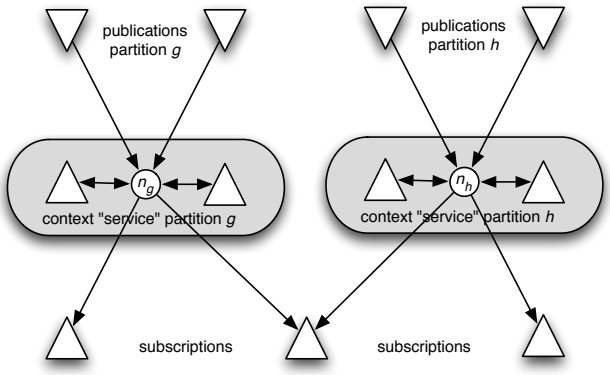

Figure 1: Example of a mapping of SPLICE partitions onto ECLIPS.

As an example of this, consider a straightforward mapping of SPLICE onto ECLIPS: a partition g can be represented by a neutral vertex n_g, publications in partition g then become nablas with a path to n_g, and subscriptions in g become deltas reachable from n_g. When a single subscription is allowed to exist in multiple partitions simultaneously, e.g., in partitions g and h, the equivalent in ECLIPS is for the corresponding delta to be reachable from multiple neutral vertices, e.g., n_g and n_h.

The context mechanism of SPLICE can then be mapped onto deltas that are in a single strongly connected component with the neutral vertex representing a partition. Figure 1 illustrates this for two partitions, g and h.

5.3.1 The context of a delta

The context $\mathcal{C}(w)$ of a delta w is defined as

$$\mathcal{C}(w) \equiv \{v \mid v \in \Delta_{\sigma_w}(V) \wedge M'_{\sigma_w}(v) \longrightarrow' M'_{\sigma_w}(w)\}$$

That is, all deltas v from which w can be reached by instances of σ_w without passing through other deltas for the same sort, evaluated on the graph obtained by contracting all strongly connected components.

5.3.2 The context invariant

The invariant on the contents of local stores can only be defined in terms of a model of local stores, but a true formal definition requires the use of a temporal logic suitable for asynchronous distributed systems, such as [18, 17], which is beyond the scope of this article. It will thus be a semi-formal definition.

First, assume without loss of generality that each local store is a process receiving an arbitrary sequence of insert, local-take and read events, and that only a single object exists, as keys partition a sort into objects and hence the objects may be treated independently.

ECLIPS defines a total order on the instances of a single object, and it is this order that determines whether or not an instance updates a local store or is discarded—see section 8.6 for the details. This allows the state of a local store to be modelled as a single value, with $-\infty$ representing an empty store.

The events (except for read) on the local store of a delta w cause changes to its state. Let the sequence of these states be $W_0 = -\infty, W_1, W_2, \ldots$: then, the effect of the events on the local store of delta w in state W_k are:

insert x	$W_{k+1} = \max(W_k, x)$
local-take	$W_{k+1} = -\infty$
read	$W_{k+1} = W_k$

The invariant on the contents of the local store of a delta and its context is dependent on the paths through the graph, which may change at any time. In a steady state, it is straightforward: if $v \in \mathcal{C}(w)$ at state V_k then $\exists m(V_k \leq W_m)$. In essence, a delta must contain or have contained data no older (as determined by the total order) than does its context, except for a time delay.

There are two changes in $\mathcal{C}(w)$ that affect this: a delta v may be added to the set—in which case there will be a V_k at which $v \in \mathcal{C}(w)$ holds at all locations in the system, and the system may once again be considered in a steady state. The other is removal of v from the set; in this case again it may take a while but eventually $v \notin \mathcal{C}(w)$ everywhere and the steady state case again applies.

The transitions are simply handled by stating that the relation need not hold.

5.3.3 Maintaining the context invariant

The write x operation on a nabla p may be modelled as

$$\forall v((v \in \Delta_{\sigma_p}(V) \wedge p \rightsquigarrow_{(\sigma_p)} v) \Leftrightarrow \text{eventually}(\text{insert } x \text{ in } v))$$

which trivially maintains the invariant.

Changing the graph G may cause the contexts of deltas to change, and in those cases where $\mathcal{C}(w)$ is changed to include a delta v, it is generally necessary to copy the contents of the local store of v to w to maintain the context invariant relating v and w.

Obviously there may be a z such that $w \in \mathcal{C}(z)$, in which case updating w may in turn require updating z to maintain the context invariant relating w and z. However, if already $v \in \mathcal{C}(z)$, then updating w with content from v clearly has no effect on the context relation between w and z nor on that between v and z. Note that this does not require the local store of z to actually contain the data to be copied to w: they may have been removed from it using local-take, or indeed it may even contain instances from other sources that are greater according to the ordering on instances of the sort (see section 8.6). This is illustrated in figure 2.

Furthermore, any delta x reachable from w only via such a z, i.e., $\{x \mid x, z \in \Delta(V) \wedge v \in \mathcal{C}(z) \wedge w \rightsquigarrow x \wedge \forall p(w \rightsquigarrow_p x) : z \in p\}$, cannot have $w \in \mathcal{C}(x)$ and therefore the updating of w need not affect x (again, see figure 2).

Accounting for data sorts and the distinction between G and G'_σ, adding v to $\mathcal{C}(w)$ requires the copying of the contents of the local store of one of $\Delta_{\sigma_v}(M_{\sigma_v}(M'_{\sigma_v}(v)))$ to a number of deltas. Such a source–destination pair is called a *context obligation*, and is notated $A \mapsto B$, with B the set of deltas in need of context data from one of the set of equivalent deltas A. The context

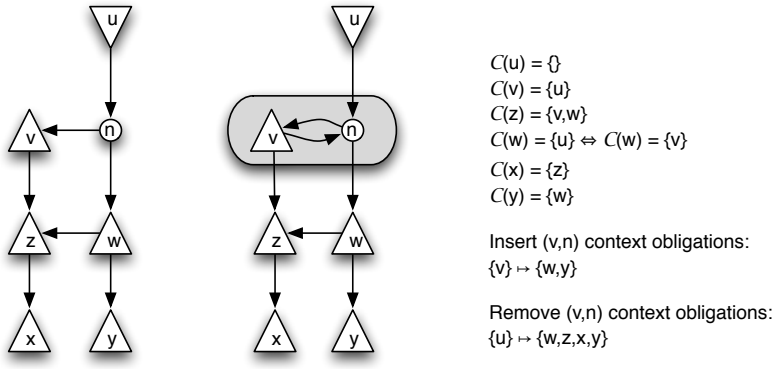

Figure 2: Example of the contexts of the deltas in a contrived example, and of the context obligations caused by inserting the edge (v, n) in the system in the left half, yielding the one in right half, and those caused by removing it. The grey rounded rectangle indicates the strongly connected component.

obligation ensuing from the addition of v to $\mathcal{C}(w)$ is

$$\Delta_{\sigma_w}(M_{\sigma_w}(M'_{\sigma_w}(v))) \mapsto \\ \{w\} \cup \\ \{z \mid z \in \Delta_{\sigma_w}(V) \wedge \exists p(M'_{\sigma_w}(w) \rightsquigarrow'_{(\sigma_w),p} M'_{\sigma_w}(z) \wedge \forall i(M'_{\sigma_w}(v) \not\rightarrow' p_i))\}$$

Note that equivalent deltas are always added to or removed from $\mathcal{C}(w)$ as a group, so if v is added to $\mathcal{C}(w)$, then all the equivalent deltas $\Delta_{\sigma_v}(M_{\sigma_v}(M'_{\sigma_v}(v)))$ are added as well.

5.3.4 Causes for changes to a context

Insertion of an edge is the most obvious cause of extending the contexts of deltas; but it may also cause the removal of a delta v for σ from the context of a delta w for σ, if the edge creates a strongly connected component containing a delta for σ such that it is on every path from v to w. Refer to figure 2 for an illustration.

Evidently, no action is required when v is removed from $\mathcal{C}(w)$. However, if insertion of an edge can cause the removal of a delta v from the $\mathcal{C}(w)$, removal of an edge may also cause the addition of v to $\mathcal{C}(w)$. For this to occur, the removal of the edge must break a strongly connected component—i.e., the exact opposite of inserting an edge. Note that the resulting context obligations are not symmetric.

Furthermore, a delta is normally an active entity, but it may become passive (see section 6.2), in which case it behaves like a neutral vertex. Thus, its becoming passive may also cause the contexts of other deltas to be extended. In contrast to inserting and removing edges, this does not cause a change in connectivity in the graph, and so when v becomes passive, it suffices to simply substitute $\mathcal{C}(v)$ for v in $\mathcal{C}(w)$ for all w s.t. $v \in \mathcal{C}(w)$.

The reverse, a transition of v from passive to active may analogously cause the addition of v to various contexts as well as potentially removing elements of

$\mathcal{C}(v)$ from those contexts. Because a passive delta has no state, when v becomes active it needs to receive content from its context. No other delta needs to receive this data, and there is no data in v's local store to distribute to those deltas for which v is added to the context.

6 Manipulating the data space structure

The structure of the data space—the data sorts and the nablas, deltas and their connections—is defined by the objects of predefined sorts in the data space, and manipulation occurs through the standard data space operations (an old idea: see, e.g., [34, 22]). Evidently, operations on these sorts have effects beyond those of operations on any other sort; however, from the perspective of application processes there is no distinction between the predefined sorts and those defined by the application.

A fundamental difference exists between the objects of predefined sorts stored in the data space that define the structure of the data space, and the elements induced by them that embody the data space and that the application processes can interact with. These elements are called *entities*, to distinguish them from the objects in the data space. The heralds mediate between the two by providing entities for the objects of predefined sorts in the data space.

If, for example and ignoring the obvious bootstrap issue, an application process P creates a connected nabla–delta pair for the sort defining deltas, and then uses this new nabla to publish a new object v of the predefined delta sort, the mere presence of v in the local store of the first delta causes the existence of the delta entity defined by v.

6.1 Model of ECLIPS behaviour

Each herald instantiates or deletes entities (deltas, nablas, &c.), updates routing information and handles context data distribution based on its own view of the objects of predefined sorts in the data space. This view consists of the contents of the local stores of a set of deltas, one for each predefined sort, and with connections such that for any delta v for a predefined sort σ in this set, every other delta w for σ is in the context of v, i.e., $\forall w \in (\Delta_\sigma(V) \setminus \{v\}) : (w \in \mathcal{C}(v))$. Obviously, edges added to the graph by the heralds to ensure this property must be invisible to the application, and must not have any effect on data visibility for the application.

As already mentioned, an application process may create new nablas and deltas for predefined sorts. The interesting case is that of the creation of a new delta: the heralds will then have to add edges to the graph such that this new delta, too, becomes a member of the contexts of the heralds' deltas, to ensure each herald's view of the system contains a complete set of all system objects.

The left half of figure 3 illustrates a possible configuration for an initial ECLIPS system with two heralds. The various deltas and nablas, one for each predefined sort, have been abstracted into a single delta and nabla. The dashed objects are private to the heralds. The right half suggests what happens when an application publishes a set of new nabla–delta pairs for predefined sorts using the initial nabla: these new objects will be stored in the local stores of all three deltas depicted in the left half, causing the creation of the corresponding entities

as well as the creation of the edge $w \to n$ by the heralds to add w to the context of their own deltas.

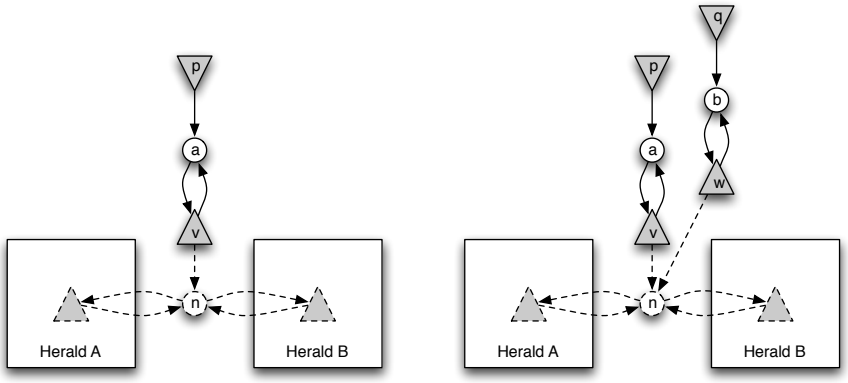

Figure 3: Illustration of the initial configuration with two heralds (left) and after the application has created additional delta–nabla pairs for predefined sorts (right). The circles are neutral vertices, the dashed lines indicate objects private to the heralds.

Some details worth mentioning: the edges $v \to a$ and $w \to b$ are not strictly necessary, but their existence enables the use of a and b for retrieving system information without requiring knowledge of the deltas for the various predefined sorts. The absence of a path $n \rightsquigarrow v$ (and $n \rightsquigarrow w$) prevents the objects private to the heralds from escaping to the application. The neutral vertex n is unique to provide the heralds with a single location for accessing the data space, although it does require the heralds to negotiate its instantiation at start-up—and note that it is no more than a location, for querying and updating the data space the heralds use deltas and nablas.

A new herald joining the system will automatically receive all objects of predefined sorts, and thus be able to participate in the shared data space, because all the heralds' deltas are in a single strongly connected component, and therefore in each other's context.

Generally speaking, the heralds will be able to detect the creation of a new delta such as wg using the regular data space operations, as an object can only exist if it is stored in the local store of a delta, in this case v as far as the application is concerned: and from v, the heralds' deltas are already reachable and thus also store the object. In case heralds are late in adding edges, the context mechanism will ensure the objects are still received by them.

There is one exception: if an application process creates a new nabla–delta pair p_1–v_1, then uses p_1 to publish a delta v_2, arranges for the local store of v_2 to contain a copy of the object v_2 (i.e., creates an edge from v_1 to v_2 so that $v_1 \in \mathcal{C}(v_2)$ and v_2 receives a copy of the contents of v_1), and finally deletes v_1, *and* does all this before some herald observed the creation of v_1, that herald may not become aware of v_2. This is resolved by requiring the heralds to always react sufficiently quickly and cooperate to prevent such a scenario, which is possible because the operations are actually performed by the heralds.

Clearly an object must be removed from the heralds' views once it has been removed from all other local stores. Of the two options that processes have for

deleting objects, the global delete using label(DELETE) is trivially handled as the views of the heralds will be updated automatically, along with the local stores created by the application processes. Local removal from all stores using local-take is handled as if a set of cooperating processes exists that eventually detect disappearance of an object from all local stores and cause it to be removed from the herald's view as if by label(DELETE).

This model causes the data of the predefined sorts to operate under the same consistency rules that application data is subjected to. For example, inserting an edge in the graph through the publication of a new EDGE-SORT instance will be observed by different heralds at different times, and multiple such actions may be observed in different orders.

6.2 Representation of entities using predefined sorts

The predefined data sorts can be divided into four groups: the data sort definitions, those defining passive entities (edges and neutral vertices) in the system, definitions of active entities (nablas and deltas) and those describing the hosts and processes in the network.

Data sort definitions are published as instances of the SORT-SORT data sort. This sort is the only regular one of the predefined data sorts. See section 8 for more information on defining data sorts.

The edges (EDGE-SORT) and neutral vertices (VERTEX-SORT) are passive entities in that they do not support any operations and do not impose any storage requirements dependent on application data. The decision to represent edges using a controlled sort is based on the possibility of using label(DELETE) for removing them efficiently as well as on the desire to allow multiple edges between a pair of vertices so that a process may create or delete edges it requires independently of other processes.

Nablas (NABLA-SORT) and deltas (DELTA-SORT) can be interacted with and require storage for application data, and it is for these reasons that they are considered active entities. However, this only holds when they are labelled with a process identifier; when the label is VOID or is a zombie label they become passive entities and behave exactly like neutral vertices. Interacting with an active entity using write, read, local-take and wait is possible only for the process with which the entity is labelled.

Upon becoming passive, all internal state of an entity is discarded, including the data in the local store for a delta. Upon becoming active, a delta gets context data from its context sources. Note that it never has any context data to deliver to other deltas upon becoming active.

Finally, there are the two sorts used to describe (not define) the active hosts (HOST-SORT) and processes (PROCESS-SORT) in the network. Instances of these sorts are generated spontaneously by the heralds to reflect the current state.

A causal relationship exists between an instance of HOST-SORT and instances of the PROCESS-SORT and other sorts for active entities residing on that host, as processes and other active entities can only exist in the context of a host. ECLIPS does not guarantee this causal relationship is reflected in the order in which automatically generated changes to objects of these predefined sorts become visible.

7 Protection in ECLIPS

7.1 Capabilities

The presence of an object of a controlled sort in a local store of a delta confers a capability on the processes that can operate on that delta. Instantiating an object of a controlled sort and publishing updates to such an object confer the same capabilities as presence in a local store, so that a publisher need not be subscribed to its own output.

Each replica of an object of a controlled sort has a strength, either *normal* or *weak* (i.e., the strength is per-object, per-local store). Normal strength is implicitly assumed, as suggested by the choice of wording. All publications made using write yield normal instances, but these become weak by traversing a *weakening edge*, which is an otherwise normal edge.

A weak replica of an object in a local store may be updated by a normal instance, or the other way around, and the strength of a replica of an object may change because of such an update. Similarly, if multiple paths exist via which an instance of x can reach a delta v, the strength of x in v will be weak if x was weak or all paths contain a weakening edge, possibly weak if x is of normal strength but some (not all) of the paths contain a weakening edge, and normal if x is of normal strength and none of the paths contain a weakening edge. Finally, if at least one of all copies of an object accessible to a process is of normal strength, then for that process that object is of normal strength.

Five different capabilities are distinguished:

forward A forward capability for an object allows a process to use the forward operation for re-publishing the current state of that object. A process has a forward capability for all objects present in its local stores.

name A name capability for x allows naming x as an endpoint for an edge, as a sequencing object for a publication, or as a process for a label. Obviously, x must be of a suitable sort; and it must be current and of normal strength.

label A label capability for x allows manipulating the label of x (using label), and requires x to be of normal strength and its label to either be VOID, the process attempting the manipulation, or a zombie label.

update An update capability for x allows publishing updates to the object (i.e., write). It requires x to be current and of normal strength, and the label of x to either be VOID or the process attempting the update.

operate Finally, an operate capability for x allows operating on the active entity: that is, if x is a delta, it allows the process to perform read and local-take operations on that delta, and if x is a nabla, it allows write and forward. For this (obviously) x must be current and of the right sort and it must be labelled with the process.

An object of a controlled sort is considered current and visible at least if it has been instantiated or updated via a nabla of the process or is stored in a local store of a delta of the calling process, and has not been made obsolete anywhere in the system. It is never considered current if it has been made obsolete by the calling process itself nor when any local store of the calling process contains or has contained an obsolete instance.

Processes can easily pass label, name and update capabilities, by ensuring the corresponding object is readable from the shared data space by the destination processes. Passing an operate capability directly is impossible, but by relabelling an object with the id of another process (or by removing the label and letting another process label it) another process will obtain an operate capability for the object.

Generally it is advisable to pass naming capabilities for neutral vertices, and create type-restricted edges to and/or from that vertex. The remainder of this section considers a way of using a combination of these for realising a protected data space.

For practical reasons all processes may be considered to always have update capabilities to all objects of a regular sort.

7.2 Confining subsystems and privacy

Consider a situation in which a supervisory system T relies on a subsystem U but where there is mutual distrust between T and U.

The ECLIPS model clearly allows for T to hand U just the nablas and deltas required by the interface: T can create new deltas and nablas for the predefined sorts and completely control their connections and, for the deltas, the contents of their local stores, prior to handing them to U. Confining U requires that it cannot use the objects it is provided with to gain access to others it has not created itself, and that it cannot publish data through the interface that cannot be published using the nablas provided by T. This is straightforward when the interface does not pass predefined sorts.

If each nabla p in the interface has outgoing edges only, restricted to p's sort, then: firstly, it is impossible to read from the data space by creating a delta reachable from p because there will not be a path; and secondly, it is impossible to publish data of a different sort by creating another nabla with a path to p, because the edge from p to the rest of the system is restricted to that one data sort. Similarly, if each delta v in the interface has incoming edges only, restricted to σ_v, then it is both impossible to publish data through v and to read data of a sort other than σ_v.

Weakening edges can be used to provide a subsystem with read-only data, and because weak objects do not confer a name capability, monitoring utilities can be built that can observe the state of the system but not modify it.

There is no guarantee for U that T will not track the system objects created by U and use this knowledge to observe or inject data internal to U—T can observe all objects created using the system objects T provided to U without U being able to observe this is being done. Hence the newenv operation: it allows U to construct a new environment similar to the bootstrap environment (and thus suitable for creating any kind of subsystem that can be built within the model), and through guaranteeing that identifiers associated with the new environment are communicated only to the calling process, it guarantees privacy to U by ensuring T cannot obtain access to the environment without the cooperation of U.

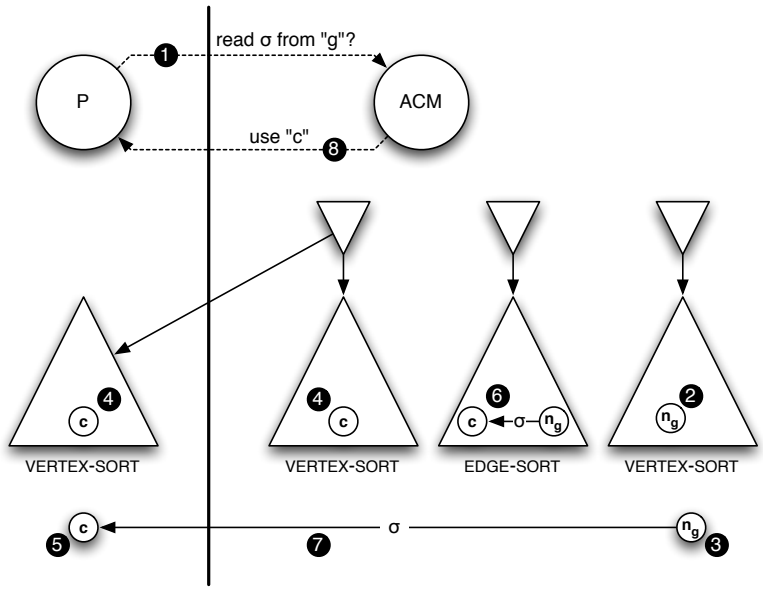

Figure 4: Illustration of P gaining read access for sort σ from partition g. See text for a description of the various labelled events. Objects drawn inside a delta represents the contents of the local store of that delta. The only entities visible to P are those that are entirely to the left of the vertical line.

7.3 Access control

The approach to access control taken by various DDS implementations is quite sensible, controlling access to the shared data space at the level of topics and partitions and/or domains. Especially partition-level access control is fairly fine grained and suitable for many applications while matching the data space structure. These implementations also offer multi-level security (see e.g., [3, 4]): for example, in [27], each application process and each partition is assigned a security level, and a process P is only allowed to read data from partitions with security levels no greater than that of P, and is only allowed to publish data in partitions with security levels no less than that of P. Biba [5] extended this model to also consider integrity levels; these can be handled in exactly the same way as the security levels.

ECLIPS does not directly provide access control to named partitions: it rather allows constructing such named partitions along with an access control system, all based on unforgeable capabilities. The appendix outlines an implementation of SPLICE using ECLIPS primitives, including a name service for such partitions; here a scheme for incorporating multi-security level access control similar to that of the DDS implementations is suggested, illustrated in figure 4. This figure distinguishes between the contents of a local store (published using write) and the corresponding entities (created by the heralds and reflecting the contents of those local stores); e.g., the neutral vertex n_g is stored in the local store of the rightmost delta, but since it is reflected in the structure of the data space, it is drawn outside a delta as well.

Let each named partition g be represented by a neutral vertex n_g, and let

an access control manager (ACM) translate partition names to unique ids—note that it could also rely on a separate name service, without affecting the principle. Here it is assumed that such neutral vertices representing named partitions are created dynamically by the ACM. Communication between an application process and the ACM can go via the shared data space, but need not.

When a process P wishes to subscribe to (or publish) a sort σ in a partition g, it sends a request to this effect to the ACM (event ❶). The ACM first verifies that P is authorised to subscribe to (publish) σ in g, taking appropriate measures if it is not. What those measures should be is intentionally left open here; the traditional response is for it to reply "permission denied", but section 7.5 suggests an alternative that may be appropriate in some situations.

If n_g does not exist yet, the ACM publishes the object n_g ❷ using a nabla for VERTEX-SORT private to the ACM, which is then inserted in the local store of a private delta for VERTEX-SORT (the only delta reachable from that nabla), which in turn causes the heralds to create the corresponding entity n_g ❸; had n_g been created earlier, the ACM reuses it. The ACM also publishes a new vertex c ❹ using a nabla for VERTEX-SORT that is connected to P's delta for VERTEX-SORT so that the heralds create the entity c ❺ and P obtains a capability for it. Then, the ACM adds an edge (by publishing an object of EDGE-SORT) restricted to instances of sort σ, from n_g to c (reversed for publishing; ❻ & ❼); and finally informs P that c is what it should use ❽.

The identifiers of both n_g and the edge from n_g to c must remain unknown to P, the former because it would allow P to gain full access to partition g, the second because it would allow P to prevent revoking the access, simply by labelling it with its own id.

Note that this same technique can also be used for any process to pass a revocable capability to another process: create a new vertex, keep the edge to/from it secret, hand the new vertex to the other process and delete the edge(s) to revoke the capability. This makes it possible to use, e.g., a direct implementation of the take-grant model [15] for securing the data space in ECLIPS. However, given that most current systems rely on access control for protection, it is especially important that that model is supported.

7.4 Revoking access

In the scheme outlined in the previous section, revoking access is trivial: one merely has to delete the edge (n_g, c) using label(DELETE) to revoke the read access of P for sort σ from partition g. However, if P has been granted access to g for multiple sorts, and all access to g needs to be revoked, an edge (potentially multiple) would need to be deleted for each sort. Introducing an extra neutral vertex r between n_g and c allows all access to g to be revoked by deleting only r (left half of figure 5).

This revocation mechanism depends on P being unable to label the object that is deleted to revoke access, in this example the edge (n_g, c) or the vertex r. There are two ways of achieving this: the first is to make sure P never receives a copy of the object of normal strength, the second is to use a designated process for revoking access and labelling the object with the id of that process. Both ensure the label operation will refuse to change the object's label, but the former has the major advantage of not requiring such a designated process.

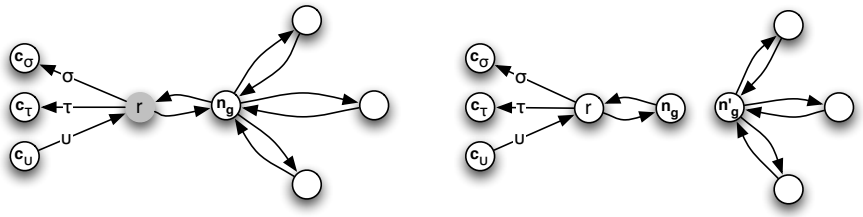

Figure 5: Illustration of two ways of revoking access to partition p. Left: deleting the grey vertex r will completely sever c_σ, c_τ and c_υ from v_p. Right: creating a new v'_p to represent p and moving all but r to v'_p effectively revokes access to partition p.

Note that this is a standard technique for revoking access (already described in [29]) in capability operating systems, and so it is in ECLIPS. If one needs to pass a "revocable capability" for a vertex v, it is sufficient to create a vertex w and two edges, (v, w) and (w, v), and then pass only the id of w. Revocation involves deleting both edges—and if revocation needs to be as simple as possible, an extra vertex such as r above can be introduced.

An alternative to revoking access of P to g exists: by creating a new neutral vertex to represent g, n'_g, and keeping P connected to n_g while reconfiguring the rest of system to use n'_g instead. With this method P is allowed to keep access, but the meaning of that which it has access to changes dramatically. (Figure 5, right half.)

7.5 Cyber defence

The alternative way of revoking access mentioned in section 7.4 creates an intriguing possibility in the context of cyber defence, especially when combined with an alternative response to denying access. Although there often may be legitimate causes for requests that need to be denied, there is also the possibility of a (would be) intruder probing the system to explore possible avenues of attack. With many systems discriminating between "permission denied" and "no such object" in their responses, attackers can often determine the presence of some objects.

Given the model with named partitions to which access is controlled, in ECLIPS these traditional responses need not be given: in ECLIPS one has the option of always granting access upon request but not necessarily to the requested partition. If the requesting process has no permission, all its access can be moved to an—otherwise useless—clone of the system, in which the requested access can then be granted without risk of compromising the real system. One could even feed this cloned system (filtered and possibly modified) data from the real system, and introduce processes to simulate feedback loops in the environment, closely mimicking a real, live operational system.

For example, in an air-traffic control system, one could imagine a shift from feeding real aircraft movements to simulated aircraft movements and the introduction of a simulated air traffic controller. To an observer, this simulated system would closely resemble a real one, but nothing done in this simulated system would affect the real system. The time it takes an intruder to notice the

difference can then be used to try to determine his identity and take appropriate action.

Such fake systems are known as honeypots or -nets, but traditionally these are separate systems. ECLIPS allows dynamically constructing a honeynet out of a live system and confining intruders to it, provided of course that *all* infected parts are removed from the operational system.

8 Data sorts

The data space is partitioned into data sorts, each of which is a combination of a data type with various attributes, described in detail in the following subsections.

Processes wishing to publish, or subscribe to, data need to first define the data sorts involved. Sorts are shared throughout the system, and so it may happen that a sort has already been defined by another process and need not strictly be defined again, but the circumstances under which this happens are undefined. In general each process has to define the sorts it operates on.

A process defines a sort by publishing a description of it as an instance of a predefined sort, SORT-SORT. This sort is somewhat of an exception among the predefined sorts as it is a regular sort: a particular sort definition will typically be published many times, but it is still *one* sort, not many instances that are similar, as would be the case using a controlled sort.

The definition is a string, the *sort descriptor*, which contains all information needed by ECLIPS. Only one valid representation exists for each sort, two valid sort descriptors that are not the exact same strings represent two different data sorts. Application processes may construct descriptors on the fly to dynamically construct new data sorts, and analyse existing ones.

The sort id is obtained by computing a crypto-hash of the descriptor string. This somewhat unusual method has several advantages over more traditional techniques, such as requiring each sort to have a unique name; in particular, it avoids the need for a naming service, it is infeasible to intentionally create an id collision, and it circumvents the need for dynamically renaming sorts during system merges. The hash space is large enough to make it exceedingly unlikely that two different descriptors hash to the same key, but when this happens, an error will be signalled eventually.

8.1 Type & key

All data in ECLIPS is typed so that local stores can be queried and format conversions in heterogeneous networks (typically byteswapping for converting between little- and big-endian hosts) can be performed automatically.

The key is a projection of the type onto a subset of that type, where two instances of the type that are projected onto the same (key) value are considered to be instances of the same object. The key may project all instances of the type onto a single value, so that only a single object of that type can exist.

Implementations are expected to provide at least the usual types, such integers, floating-point numbers, arrays and records. Typically, the data sort type will be a record and the key a subset of the fields in the record. In addition to

the usual types, a unique id type and a label type are provided as described in the following sections.

Controlled sorts must designate a single field of the unique id type as the key, and may designate a field of the label type as the label field for use by the label operation. The label field must not be part of the key, must not be referenced by the order function (section 8.6) nor by the validity and obsoleteness predicates (section 8.5), and may change even for immutable (section 8.7) sorts.

8.2 Unique ID type

The unique id type primarily exists to provide the opaque object identifiers needed for controlled sorts. Their usefulness is not limited to that and they may be used for any other purpose. Each invocation of newid yields a new unique id—any implementation will be limited in the number of unique ids it can generate, but it is assumed this limit cannot be reached in practice.

Each process has its own mapping of private unique ids to global unique ids, so that the global id may be changed independently of the private id. Giving each process its own mapping between private and global ids allows the unique ids to be used as secrets that can only be communicated through declared fields of the unique id type, as well as allowing the changing of global ids to avoid id clashses when merging systems.

8.3 Label type

The label type is a tuple of a unique id of the system variant, used for storing process identifiers, and a flag indicating whether or not it is a zombie label—i.e., whether or not it refers to a process that has terminated.

8.4 Instance validity

The sort definition defines a predicate P_{valid} for ensuring the validity of the contents of an instance, so that attempting to publish an invalid instance results in an error.

This facility is used, for example, to ensure an instance of the SORT-SORT (a regular sort) for publishing sort definitions indeed consists of a valid descriptor string and a secure hash of that descriptor string, to prevent the publication of both invalid sort descriptors and sort definitions where the descriptor and the sort id do not match.

8.5 Obsoleteness

A second predicate defined as part of the sort is the obsoleteness predicate P_{obs}: an instance x of the sort is obsolete iff $P_{\text{obs}}(x)$.

8.6 Ordering

The application defines as part of the sort a partial order $<_p$ on two instances describing the same object. This partial order is used to construct a total order

on instances of the same object in an implementation-defined way consistent with the following:

$$a < b = \begin{cases} \text{FALSE} & \text{if } P_{\text{obs}}(a) \\ \text{TRUE} & \text{if } \neg P_{\text{obs}}(a) \wedge P_{\text{obs}}(b) \\ a <_p b & \text{if } a <_p b \text{ is defined} \\ a <_1 b & \text{if } a \text{ and } b \text{ were published using the same nabla} \end{cases}$$

where $a <_1 b$ is true iff a was published before b.

An instance can only be overwritten by an instance that is greater according to this ordering. Note that this is a generalisation of the model used by SPLICE, where a time stamp is automatically added by the herald and which has later been extended to allow the application process to set the time stamp, in accordance with the findings in [16].

8.7 Immutability

A data sort may be marked as "immutable": then, an object of such a sort has its value set at instantiation and cannot be updated. For controlled sorts, object instantiation is explicit and it is indeed possible to guarantee a unique, constant state for an object of an immutable sort, simply by disallowing any writes that do not instantiate a new object.

However, for a regular sort it cannot be guaranteed that no two (or more) processes will attempt to instantiate the "same" object simultaneously with different values, in which case guaranteeing a constant state at the subscribers would risk inconsistency in the data space, with different subscribers observing different states. Yet any attempt at resolving this inconsistency would break the immutability by changing the value observed by at least one of the subscribers. Therefore, regular sorts may not be marked immutable.

9 Operations

9.1 newid

The newid operation returns a new unique id. For a nabla p for a controlled sort, newid p returns a new unique id and arranges for this id to be accepted as the id of a new object in a future write operation on p. The caller must have an operate capability for p.

9.2 newenv

The newenv operation constructs an entirely new environment similar to the bootstrap environment described in section 6 and returns the ids of the objects in the same format as they are provided to a process on startup. All objects in this new environment are unique, and none are made visible in any way other than to the caller: it provides a private subsystem that no-one can snoop on.

9.3 write

The write p, x operation creates a new normal-strength instance of σ_p with the contents taken from x and distributes copies of it to all deltas for σ_p that are σ_p-reachable from p, usually causing the local stores of these deltas to be updated by the new value. The caller must have an operate capability for the nabla p and either possess an update capability for x or have obtained the id of x from newid p.

The following steps are taken:

1. serialisation, that is, verifying the provided sort instance is well-formed and formatting a copy to conform to the internal representation (well-formedness is in part defined by the instance validity predicate, and in part by restrictions imposed by the data type);

2. for controlled sorts, verification that the object represented by x is indeed a current object of σ_p, or that it has been registered with the nabla as a new object about to be instantiated (using newid p) and has an appropriate label;

3. for controlled sorts, insertion of the key value in the set of known-obsolete objects if the state describes an obsolete object and, if it creates a new object, recording its existence and current label;

4. distribution to all deltas for σ_p reachable from p.

For controlled sorts, the contents of the label field in the instance provided to write when instantiating a new object determines the initial label. Allowed are: the id of the process itself, VOID, and DELETE to remove the registration of an id by newid p without actually instantiating an object. When updating an existing object, the label specified in the new instance is ignored and the current label of the object must either be VOID or the calling process.

Note that verifying the current label of an object to be either VOID or P is a local operation. If the current label as observed by P is VOID, the object may in fact already have been relabelled by another process, because it takes some time for the label change to propagate. Thus a process Q may have labelled the object and yet observe an update to it from P.

Figure 6 is the state transition diagram for an object of a controlled sort in a single herald. For all objects not in state "no space allocated" the herald must keep some state information, but clearly state needs to be kept indefinitely only when a local store managed by the herald contains the object.

9.4 forward

The forward p, x operation forwards the current state of the object identified by x using the nabla p and with the strength the object has for the caller. The object x must be of the controlled sort σ_p, and the caller must have an operate capability for the nabla p and a forward capability for x. Note that the current state of the object may change between the application process invoking the forward operation and the herald performing the operation, in which case it is the updated state that is forwarded.

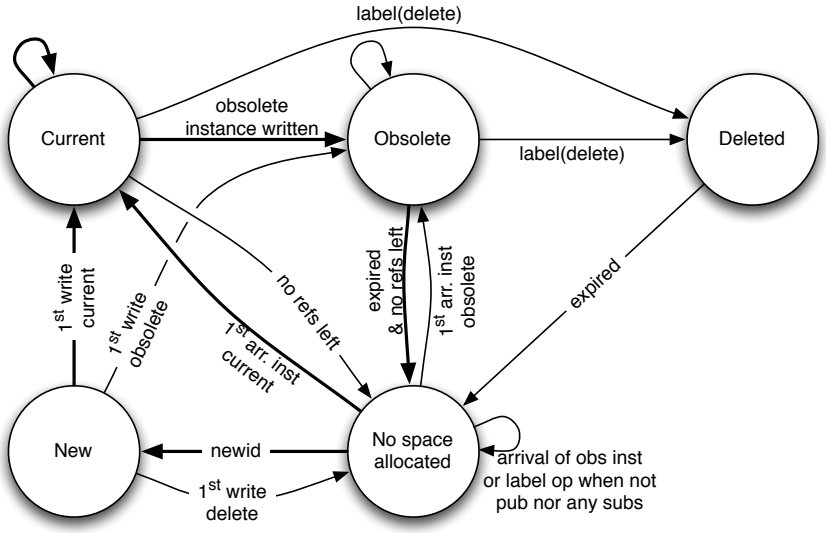

Figure 6: Object state transition diagram for an object of a controlled sort in a herald. Bold edges indicate the "normal" sequence; "expired" triggers when the time since entering the state exceeds the minimum retention time specified in the sort plus an implementation dependent margin, "no refs left" when no local stores contain a copy of the object.

For immutable sorts, one may in principle only instantiate a new object, but not publish an update to an existing one. However, this would mean the only way to forward an immutable object x would be by creating a delta v, creating an edge to it from a delta containing x, deleting all incoming edges when x has arrived in the local store of v, deleting all objects but x from the local store of v, and then adding an edge from v to, for instance, p. Using forward, this convoluted procedure can be avoided.

A weak object only confers a forward capability on an object, and behaves much like an immutable sort as the same convoluted procedure can be used to forward weak objects using only deltas and edges.

Unlike write, which requires x to be current, forward merely requires x to exist so that obsolete objects may be forwarded. The application designer specifies a length of time for which an obsolete object of a sort must be retained because of application latencies (the minimum retention time; latencies within ECLIPS are always accounted for), and a process may only in this interval forward an obsolete object. After the minimum retention time has passed, the operation fails.

9.5 read & local-take

The read q operation examines the contents of local stores of deltas, for which the caller must have operate capabilities, and returns copies of the objects satisfying the query q. If no data satisfying q is available, the empty set is returned.

The local-take q operation behaves like read but in addition to returning copies, it removes the returned objects from the local store. Note that this has

no effect on the suppressing of updates to obsolete objects but may have an effect on the ability to operate on objects of controlled sorts, as the latter is dependent on their presence in the local stores of the process' deltas.

9.6 wait

The wait $q_1, \ldots, q_n$ operation blocks until $\sum |\mathsf{read}\ q_i| > 0$. The process need not resume execution if further state changes cause $\sum |\mathsf{read}\ q_i| = 0$ before the call could return, and spurious resumptions may occur. The caller must have operate capabilities for all deltas referenced by the q_i.

9.7 label

The label x, P, f operation attempts to atomically change the label of the object of a controlled sort identified by the unique id x to P, which may be a process identifier, VOID or DELETE. When successful, all copies of the object have their label field changed to P; when not, none of them have. If x identifies an object of a sort that does not contain a label field, this operation is unavailable. The caller must have a label capability for x and a name capability for P.

If the specified object is currently labelled with a process identifier, only that process may change the label and the new label may be the identifier of any process, VOID or DELETE.

If the specified object is not currently labelled with a process identifier, the label may be changed by any process, but the new label must be the id of the calling process, VOID or DELETE. If it is currently a zombie label, indicating that the process it was labelled with has terminated without first removing its label from the object, the label can only be changed if forced by $f =$ TRUE: allowing but not requiring separation of normal operations (where the calling process sets $f =$ FALSE) and recovery operations (where $f =$ TRUE).

Labelling an object with DELETE causes immediate removal of the object throughout the system.

Changing the label of an obsolete object is allowed, and in particular so is labelling an obsolete object with DELETE.

A label operation on an object x has an ordering side effect: sort instances published via nablas that have x designated as their sequencing object and updates to x itself by the calling process, and published prior to the label operation, will become visible before the new label becomes visible; and those published subsequent to the label operation will not become visible before the new label does. This side-effect exists even when the new label equals the old label.

Furthermore, for any objects of predefined sorts for which the ordering side effects of the label operation holds, the heralds' processing of these objects will occur before the label operation returns. For example: after completion of label(x,DELETE), x has been deleted; and after write(p, x);label(y) with p a nabla for, e.g., EDGE-SORT and with y as sequencing object, and assuming the existence of a delta for EDGE-SORT reachable from p, the edge described by x exists.

10 Discussion and further work

ECLIPS currently is but a thought experiment. There are three avenues along which progress may now be made: formalising the model, constructing an implementation, and enhancing the weaker areas. Enhancements affect both formalisation and implementation, construction of a formal model may yet show the existence of a fundamental flaw, and an implementation may well indicate performance problems. This strongly suggests formalisation should be given priority.

The observation that testing is no substitute for proof of correctness is likely to hold even more for security claims, considering the difficulty of testing protection mechanisms—this has long been recognised [3, 19]. ECLIPS is no exception.

In addition to modelling security properties, it also necessary to formally model the derivation of the data space structure from its contents. If ECLIPS were a synchronous, sequential model, it could easily be shown to be well-behaved, but its asynchronous and concurrent nature gives ample space for inconsistent behaviour to escape notice in informal reasoning or testing. The most obvious question to be answered is: does it matter in which order two edges are inserted or deleted? Particularly in cases where the contexts of deltas for predefined sorts are affected, proof that the system remains in a consistent state is highly desirable.

Areas of particular concern in implementation are the computational complexity of operations on general directed graphs, and the label operation and its ordering properties. Much progress has been made in dynamic graph algorithms, for example in maintaining the transitive closure [10, 30]. ECLIPS does not rely directly on the transitive closure, but it appears that the problem is of similar complexity, and to have an algorithm of quadratic complexity at the core of topology changes may well prove a limiting factor. On the one hand, the way SPLICE has always been used suggests that many opportunities exist for reducing the number of edges and vertices to be processed; on the other, utilising the protection mechanisms described in section 7 likely changes the usage patterns.

An example of an opportunity for reducing the complexity is when a number of deltas exist with but a single, incoming edge, all originating at a single neutral vertex: then these deltas may be considered to be represented by that neutral vertex, reducing the number of edges and vertices to be considered when the graph changes.

With respect to the label operation: synchronous operations always cause trouble in distributed systems. It appears that tracking two things for each object suffices to simplify many cases: the first is whether or not an object has "escaped" the herald at which it has been instantiated, and the second whether or not an object has ever been used anywhere as a sequencing object. Only when both are true does one encounter the full complexity of the label operation, and this may well prove to be a rare case.

Finally, the most obvious and desirable enhancement is that of a measure of transactional integrity, especially treating a number of updates to or instantiations of objects as an atomic unit where either all happen, or none do. The integration of this in the absence of changes to the topology of the data space is straightforward, but that is not a very realistic restriction. It seems sensible to first formally model the data space and only then consider such an enhancement.

11 Conclusion

A design for a novel shared data space has been proposed in this article, and directions for further work have been discussed. The design includes a fair number of novelties and minor inventions, some of which have merit of themselves—and regardless of whether the complete design will withstand the tests of formal modelling and implementation, these elements may well be useful in the design of other systems.

Appendix

This appendix outlines a SPLICE implementation using ECLIPS operations. A number of things are assumed to exist: a delta–nabla pair for publishing and reading partition-to-vertex mappings (*partition-map-sub* and *partition-map-pub*, with the partition name the key), another pair for a set of ids of context daemon processes (*context-id-sub* and *context-id-pub*), and an object of a controlled sort present in the local store of a delta (*token* and *token-sub*). SQL select statements are used to query local stores.

Also assumed is the existence of nablas and deltas for publishing and reading the predefined sorts such as EDGE-SORT and NABLA-SORT. Again the nablas are referred to with a "pub" suffix, such as *edge-pub* and *nabla-pub*; the deltas with a "sub" suffix. Finally, *vertex-pub* and *partition-map-pub* are assumed to both use *token* as their sequencing object.

It is interesting to consider the relative performance of this SPLICE implementation with respect to that of a direct implementation of SPLICE. Evidently, this is very much dependent on the quality of such implementations—but ECLIPS itself is not yet implemented, let alone optimised. Therefore, only very general, qualitative remarks are possible, illustrating why the implementation on top of ECLIPS *should* perform similar to the direct implementation.

Firstly, consider regular reading and writing of data by application processes in a stable data space topology. SPLICE does not have controlled sorts, only (what in ECLIPS are called) regular sorts. The semantics of write and read are almost the same, with careful consideration given to reducing the complexity of implementation of ECLIPS relative to that of SPLICE by reducing the number of exceptional cases. ECLIPS is more complex in the way the set of subscribers is determined, but this can well be computed during topology changes or computed on-demand and cached, and thus be amortised over all the write operations performed by the application.

Secondly, declaring a partition. A direct implementation of SPLICE can choose between communicating partition names or providing a name service (however implemented) for maintaining a bijective mapping of partition names to internal identifiers; the former has negative consequences for the message format and the overhead in distributing instances, but the latter requires a fault-tolerant name service. An implementation on top of ECLIPS is forced to use a name service (again, however implemented). For supporting applications that use few partitions, have high data rates and require low latencies—i.e., those for which SPLICE was originally designed—the design trade-off is likely to favour the use of a name service. The implementation on ECLIPS is therefore likely to exhibit performance characteristics similar to a direct implementation: deter-

mining the internal identifier for a partition is a relatively expensive operation for new partitions, and very inexpensive for cached ones.

Thirdly, context data. SPLICE relies on a set of daemon processes that are notified the first time a process declares its intent to publish a context data sort, and no instances of a context data sort may be published until these daemon processes have made their preparations. A good implementation will exploit the semantics and merge the various steps to minimise overhead. The SPLICE implementation on top of ECLIPS described here employs a similar model, but it is unclear whether comparable optimisations exist. Fortunately, declaring an intent to publish is a rare operation compared to publishing data and the cost of these operations will usually be low becasue of the absence of synchronous operations.

Finally, the overhead of all explicit graph manipulations in the ECLIPS-based version. Evidently, this is a major handicap: the ECLIPS semantics inherently induce *much* greater algorithmic complexity than do the straightforward data structures required by SPLICE, and with all operations performed explicitly, there are few opportunities for optimising sequences of operations. This has already been remarked upon in section 10, and performance simply depends on the applicability of various optimisations such as those mentioned there, the actual size of the graph and the actual complexity of the implementations of the various operations. Suffice it to say that the implementation on top of ECLIPS will have a higher overhead than a direct implementation, but also that the cost amortised over all write operations is likely still very low, and that the latency on the critical path probably is low, too, because of the asynchronocity of the involved operations.

The essential parts of the SPLICE interface are: declaring sorts and partitions, subscribing to publications of a sort in a set of partitions, declaring oneself to be a publisher of a sort in a partition, and reading and writing data. Reading and writing of regular sorts in ECLIPS is sufficiently like the operations in SPLICE to be used as-is. The others are straightforward but for the declaring of partitions and the treatment of "context" sorts.

Declaring a sort becomes publishing an instance of SORT-SORT with the descriptor and the (derived) id:

```
PROCEDURE declare-sort(S: descriptor)
  x := new Sort(id=>hash(S), descriptor=>S)
  write(sort-pub, x)
```

Declaring a partition is more involved: only the first time a particular partition is declared a new neutral vertex and a mapping entry must be created. This particular implementation uses the label operation on *token* to ensure mutual exclusion and data visibility, and will recover from crashes (by virtue of selecting *force* mode). (Note that it suffers from the thundering herd problem.)

```
PROCEDURE declare-partition(P: name)
  if lookup-partition-id(P) = undef then
    retry = true
    while retry do
      wait(SELECT 1 FROM token-sub
             WHERE label = void OR iszombie(label))
      retry = (label(token, self, true) /= success)
```

```
    od
    if lookup-partition-id(P) = undef then
      x := new Vertex(id=>newid(vertex-pub), label=>void)
      y := new Partition-Map(id=>x.id, name=>P)
      write(vertex-pub, x)
      write(partition-map-pub, y)
    fi
    label(token, void, false)
  fi
```

ECLIPS does not guarantee that a crash in declare-partition cannot result in only the partition-map entry being visible, unless one inserts label(*token*, *self*, FALSE) between the two write operations. This implementation instead solves it by requiring both objects to exist in lookup-partition-id. Note that a crash may cause an unreferenced object of VERTEX-SORT to remain in the system.

```
FUNCTION lookup-partition-id(P: name)
  return read(SELECT id
                FROM partition-map-sub AS m, vertex-sub AS v
                WHERE m.name = P AND m.id = v.id)
```

Subscribing to a sort S in a set of partitions is straightforward: create a new delta for S and, for each partition, add an edge to it from the neutral vertex representing that partition.

```
FUNCTION subscribe(S: descriptor; Pset: set of names)
  id := new-delta(hash(S))
  for each p in Pset
    new-edge(lookup-partition-id(p), id)
  return id
```

Declaring oneself to be a publisher is slightly more involved if the sort is a "context" one: this example implementation follows SPLICE closely and uses a "context daemon" to automatically subscribe to "context" sorts, but this service needs to be given time to perform the necessary actions before the first instance of the sort is published.

```
FUNCTION publish(S: descriptor; P: name)
  id := new-nabla(hash(S))
  new-edge(id, lookup-partition-id(P))
  if iscontext(S) then
    wait-context(hash(S), lookup-partition-id(P))
  fi
  return id
```

Waiting until a context daemon has subscribed is fairly easy if it is known what the unique ids of the context daemons are: one merely needs to determine if a delta exists for the sort that is, firstly, owned by a live context daemon—done by checking the label against the set of known context daemon process ids and requiring the label not be a zombie label—, and secondly, connected to the partition with an edge from the partition to the delta.

```
PROCEDURE wait-context(S: sid; P: uid)
  PREPARE q AS SELECT 1 FROM delta-sub WHERE
    sort = S
    AND label IN(SELECT id FROM context-id-sub)
    AND NOT iszombie(label)
    AND EXISTS (
      SELECT 1 FROM edges
        WHERE edges.src = P AND edges.dst = delta-sub.id)
  while |read q| == 0 do
    wait q
  od
```

The context daemon process simply selects all nablas for context sorts for which it has no delta, and adds subscriptions and edges for each of these. If multiple context daemons exist, they will all do so.

```
PROCESS context-daemon
  write(context-id-pub, new Context-Id(id=>self))
  PREPARE q AS SELECT DISTINCT
    sorts.sort AS sid, edge-sub.dst AS pid
    FROM nabla-sub, sort-sub, edge-sub WHERE
      sort-sub.sort = nabla-sub.sort
      AND edge-sub.src = nabla-sub.id
      AND iscontext(sort-sub.descriptor)
      AND NOT EXISTS (SELECT 1 FROM delta-sub WHERE
        delta-sub.sort = nabla-sub.sort
        AND delta-sub.label = self)
  while true do
    wait q
    S = EXECUTE q
    for each s in S do
      id = new-delta(s.sid)
      new-edge(s.pid, id)
      new-edge(id, s.pid)
    od
  od
```

The remaining helper functions are trivial, and given without further comment:

```
PROCEDURE new-edge(A: id; B: id)
  e := new Edge(id=>newid(edge-pub), label=>void,
                src=>A, dst=>B, predicate=>true)
  write(edge-pub, e)

FUNCTION new-delta(S: sid)
  d := new Delta(id=>newid(delta-pub), label=>self, sort=>S)
  write(delta-pub, d)
  return d.id

FUNCTION new-nabla(S: sid)
  n := new Nabla(id=>newid(nabla-pub), label=>self, sort=>S)
```

```
write(nabla-pub, n)
return n.id
```

Acknowledgements

The author is very grateful for all the discussions with and comments from several people. Despite their efforts, no doubt errors and omissions remain, and it is the author who bears the sole responsibility for those. Above all, he wishes to thank M. Boasson for his patience with the expression of ideas still nebulous and his scrutiny of multiple drafts; the others are, in alphabetical order: W.M. Beynon, N.R. Howes, J.W. Lokin and R. Torenbeek.

References

[1] David S. Alberts, John J. Garstka, and Frederick P. Stein. *Network Centric Warfare*. National Defense University Press, 1999.

[2] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *Dependable and Secure Computing, IEEE Transactions*, 1(1):11–33, 2004.

[3] D.E. Bell and L.J. LaPadula. Secure Computer Systems: Mathematical foundations. 1973.

[4] D.E. Bell and L.J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. pages 1–134, Aug 1976.

[5] K.J. Biba. Integrity Considerations for Secure Computer Systems. pages 1–68, Jan 1975.

[6] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. *ACM SIGOPS Operating Systems Review*, 21(5):138, 1987.

[7] Roel Bloo, Jozef Hooman, and Edwin de Jong. Semantical aspects of an architecture for distributed embedded systems. pages 149–155, 2000.

[8] Maarten Boasson. Technical Memorandum SEAT 362. Hollandse Signaalapparaten B.V., Feb 1984.

[9] Maarten Boasson. Control Systems Software. *IEEE Transactions on Automatic Control*, 38(7):1094–1107, 1993.

[10] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 381, Washington, DC, USA, 2000. IEEE Computer Society.

[11] Jack B. Dennis and Earl C. van Horn. Programming semantics for multiprogrammed computations. 1966.

[12] J.E. Donnelley. A Distributed Capability Computing System (DCCS). *Third International Conference on Computer Communication, Toronto, Canada*, pages 432–440, Feb 1976.

[13] David Gelernter. Generative communication in Linda. *ACM TOPLAS*, 7(1):80–112, 1985.

[14] Norm Hardy. KeyKOS Architecture. *ACM SIGOPS*, 1985.

[15] M.A. Harrison. Theoretical issues concerning protection in operating systems. *Advances in Computers*, 24:61–100, 1985.

[16] Jozef Hooman and Jaco C. van de Pol. Verifying replication on a distributed shared data space with time .... *Proc. 2nd Workshop on Embedded Systems.*

[17] Norman R. Howes. *Volume 6: Distributed Systems Theory.* IEEE ReadyNotes Series in Distributed Systems Software Engineering. IEEE Computer Society. *Forthcoming.*

[18] Norman R. Howes. A Theory of Distributed Systems. pages 205–213, 2006.

[19] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys (CSUR)*, 13(3):247–278, 1981.

[20] Henry M. Levy. Capability-based computer systems. 1984.

[21] Karl MacMillan and Chris PeBenito. Securing RTI Data Distribution Service with SELinux. RTI Whitepaper, `http://www.rti.com/`, 2009.

[22] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3(4):184–195, Aug 1960.

[23] SUN Microsystems. Code Conventions for the Java Programming Langauage. `http://java.sun.com/docs/codeconv/`, 1997.

[24] S.J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289, 1986.

[25] Object Management Group. Data Distribution service for Real-Time Systems, version 1.2, 2007.

[26] G. Pardo-Castellote. Secure DDS. `http://www.omg.org/news/meetings/workshops/RT-2007/05-2_-Pardo-Castellote-revised.pdf`, 2007. Presentation given at RTESS, Washington D.C., July 2007.

[27] PrismTech. *OpenSplice DDS Security Configuration Guide*, Sep 2009.

[28] Real-Time Innovations. NDDS 3.x. `http://www.rti.com/`.

[29] David D. Redell. Naming and protection in extendible operating systems. *Ph.D. thesis, University of California, Berkeley*, Jan 1974.

[30] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, page 191, 2004.

[31] Jonathan Shapiro. *EROS: A capability system*. PhD thesis, 1999.

[32] Jaco C. van de Pol. Expressiveness of Basic Splice. Technical report, 2000.

[33] J.H. van 't Hag. Data-Centric to the Max—The SPLICE Architecture Experience. page 207, 2003.

[34] John von Neumann. First Draft of a Report on the EDVAC. pages 1–51, Feb 1945.

[35] Wikipedia. Publish/Subscribe. `http://en.wikipedia.org/wiki/Publish/subscribe`.