# The Separation and Krenz Specifications

# Version 3.0

# table of contents

# List of figures

# 1 Objective and Introduction

## 1.1 Introduction to version 1.0

The purpose of this report is to present the work to build the separation specification on the Programatica project. In fact, this report will also present the beginnings of the Krenz specification on the Programatica project. The reason for combining these two is that the Krenz specification depends upon the Separation specification. The specifications have been designed as a hierarchy of Haskell classes and instances. The Krenz specifications are instances of classes that are sufficiently general to handle the separation specifications and several versions of the Krenz specifications.

The statement of work for the Programatica project calls for a Krenz model at several levels, including the enterprise, network, and platform levels. The Haskell classes presented in this report have been parameterized to cover the required Krenz levels, as well as other instances of Krenz not anticipated in the statement of work. This parameterization of the Haskell classes to encompass many instances of Separation and Krenz is a significant accomplishment of this work.

The enterprise Krenz model calls for a concept of topology to be introduced. The framework of Haskell classes presented in this report will lay the foundation for the concept of topology to be introduced in the enterprise Krenz report. The objective of the topological concept is to be able to answer questions such as the following:

- **Conformance to Security Policy:** Given an enterprise constructed from many network, platforms, processes, threads, etc., does the enterprise conform to the Krenz model of information flow specified for the enterprise?
- **Security of alteration:** Given a proposed change to an enterprise (e.g. an additional network, an additional platform, etc.) is the resulting network still in conformance to the Krenz model of information flow for the enterprise? If not, what filters must be added to bring the proposed change into conformance?

The second question (security of alteration) reveals another objective of the hierarchy of Haskell classes. The architecture should be dynamic, so that new elements and connections can be added. For example, new networks, network connections, platforms, etc. can be added to an enterprise. This is a significant advance over the Mathematically Analyzed Separation Kernel (MASK). MASK supported only a static set of processes, threads, and communication paths between them.

The objectives of the hierarchy of Haskell classes are summarized in Table 1. This report presents the hierarchy of classes in a form sufficient to support the objectives listed in Table 1 for the separation property. To support the Krenz properties will require some extensions. In particular, the hierarchy of classes at this time does support the addition of elements in the architecture, but not the addition of connections between the elements of the architecture.

A final objective of the hierarchy of classes (not accomplished at the time of this report) is to extend the parameterization of the classes. The classes are polymorphic in the type of elements and connections that comprise the architecture. In Haskell, this means that

the elements and connections can be of any type, but they must be consistent throughout the architecture, i.e. all the elements are of the same (polymorphic) type, and all the connections are of the same (polymorphic) type. It would be nice to have an architecture that is built from elements of one type in one part of the architecture, and of a different type in another part of the architecture. For example, it would be nice to use Linux platforms in one place and NT platforms in another place. Haskell has mechanisms to permit this kind of extended polymorphism. In future versions of this hierarchy of classes, we will investigate the use of these Haskell mechanisms to extend the polymorphism.

*Table 1: Objectives of the hierarchy of Haskell classes*

| Objective | Description | Further information |
|---|---|---|
| **Parameterized model** | The ability to instantiate the hierarchy of classes to capture many versions of Separation and Krenz. | Section 5.3. |
| **Hierarchy of levels** | The ability to specify a hierarchy of elements, each having its own separation or Krenz property | Section 5.4. |
| **Topology** | The introduction of a topological concept, permitting what if questions about modifications to the architecture. | Version 2.0 of this report (see section 1.2) |
| **Dynamic architecture** | The ability to add elements to the architecture (adding connections not yet done) | Section 0. |
| **Extended polymorphism** | The ability to compose an architecture of elements with many different types. | Future report |

## 1.2 Introduction to version 2.0

This version of the specifications adds a concept of topology to the Separation and Krenz specifications. The notion of topology used is that of a Grothendieck Topology, as described in the PhD thesis of Srinivas [8] and in other texts [1]. Underlying the concept of Grothendieck Topology are the concepts of Category [3, 4, 5, 6], Sieve [1, 8], and Sheaf. [1, 8].

For the separation specification, the basic idea underlying the addition of Grothendieck topology is that each instance of the separation specification is an object in a category (called **Sep**), and a separation homomorphism is an arrow in the category. In the category (**Krenz**), objects are instances of the Krenz specification, and arrows are homomorphisms that preserve the Krenz structure. The Krenz Assurance Graph lives in a special instance of the **Krenz** category, called **KAG**.

The reason for adding a topology to the Separation and Krenz specifications is to provide the structure in which pattern matching takes place. The development of a pattern-matching algorithm, based on the Knuth-Morris-Pratt (KMP) algorithm, is the subject of

the thesis of Srinivas (8). Based upon this pattern-matching algorithm, questions can be asked such as:

- Is a complex system an instance of a Separation specification?
- Is a complex system an instance of a Krenz specification?
- Is a complex system an instance of a Krenz Assurance specification?
- If a modification is to be made to a system, will it still satisfy a (Separation / Krenz / Krenz Assurance) specification?

The Krenz specifications are also intended to be the basis for formal security models for computer platforms and networks. The Krenz specification has a protection profile associated with it, which is the starting point of seeking security assurances under the common criteria (CC).

A significant success of the version 2.0 effort is that the Haskell specifications from version 1.0 were reused without modification. Two new files (CategoryC.hs in section 0 and GraphCategory.hs in section 7.12) were added to the version 1.0 specifications. This speaks highly of the reusability of Haskell definitions and code.

## 1.3 Introduction to version 3.0

The most important changes in this version of the specification are:

- **Theorem proving:** The theorem proving aspect of the specification has been considered. Several theorem provers were investigated (Hol, PVS, Maude, and Isabelle), and Hol was chosen as the theorem prover to use on the Krenz specifications. The simpler definition of recursive graph was carried through to all of the other definitions (such as the Krenz System) built on top of the recursive graph concept. The theorem proving considerations have had three major effects:
  - **Inductive definition of recursive graphs:** The definition of the recursive graph data structure was modified to use an inductive definition, based on the work of Martin Erwig ([1]). The previous definition of recursive graph had no clear structure, and it was unclear how to map it into Hol for theorem proving.
  - **Simplification of other parts of the specification:** Based on the simpler method of defining recursive graphs, the other specifications built on the recursive graph were also simplified.
  - **Dynamic version of recursive graphs:** As a natural byproduct of the inductive definition of recursive graphs, the recursive graph structure is now dynamic, i.e. nodes and edges can be added to or deleted from a recursive graph.
- **Testing:** The deepen and flatten methods of the recursive graph were tested on several graphs. Testing the functions is a necessary first step before the theorem proving effort begins.

- **Network Krenz:** The work of the previous two specifications has already introduced the structures necessary for the network version of the Krenz specification, so no additional work was needed here.

The informal portion of this document (section 2 through section 6) remain unchanged in version 3 of the report. The Haskell source (section 7) has been updated with the new Haskell source. The Hol source (partial at this point) is now included in section 8.

# 2 Informal Description of Separation

This section describes the separation property. The separation property first arose in the context of an operating system. The separation property is first described at the operating system level in section 2.1. This property is abstracted to suit any instance of the separation property in section 2.2.

## 2.1 Separation concept at the operating system level

Separation at the operating system level is depicted in Figure 1. Many operating systems (such as NT and Unix) are good at separating processes in terms of providing separate address spaces, such that reads and writes to one address space are independent of reads and writes to another address space. However, these operating systems are poor at separating processes in terms of the operating system itself. What this means is shown in Figure 1. By means of operating system interfaces, one process can influence the internal state of the operating system. This influence can later be detected by another process running under control of the same operating system.

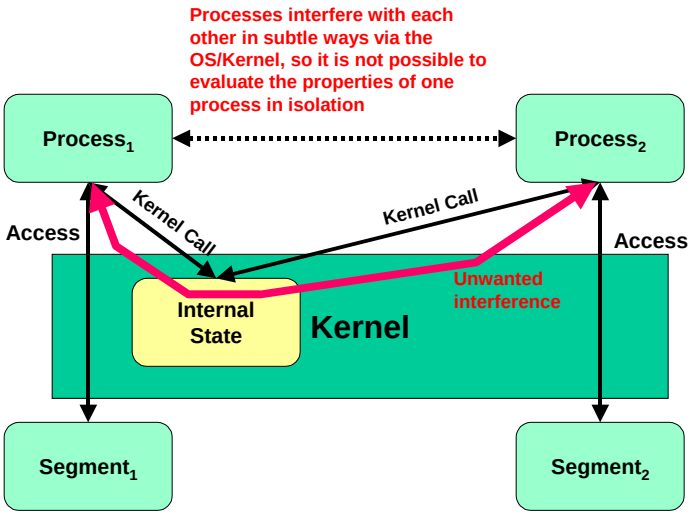



*Figure 1: Separation at the operating system level*

Allocation and de-allocation of resources is a common example of such paths of influence. One process allocates resources, and the other process can detect that resources have been allocated, often by using operating system status tools that return the level of resource allocation. The “disk free (df)” utility of Unix is a good example of this. Widely used operating systems have hundreds (even thousands) of interfaces, and abundant

opportunities to communicate information by means of operating system interfaces that were not intended as communication mechanisms.

A separation kernel is designed to eliminate information flows by means of the kernel, and permit only explicitly allowed communication by the communication mechanisms provided by the kernel. The separation property captures the resulting separation between the processes. Informally, the separation property is:

> **Informal Separation Property:** Processes A and B are separated if the actions of process A cannot influence the actions of process B, and the actions of process B cannot influence the actions of process A.

*Figure 2: Informal separation property*

Note the word "cannot", rather than "does not". Two processes might be designed so that they "do not" influence each other, even though there are mechanisms that might enable them to influence each other. "Cannot" means there is no mechanism for the two processes to influence each other, except for the explicitly allowed communications mechanisms of the operating system.

## 2.2 Abstraction of the separation concept

The separation property can be abstracted from the context of operating systems, to apply to any instance where separation is of interest. Other examples of separation are separation between the bands of different radio stations, separation between compartments in a battleship (provided by bulkheads), and separation between communications channels in a network. The abstract separation property to capture all these instances of separation is depicted in Figure 3.

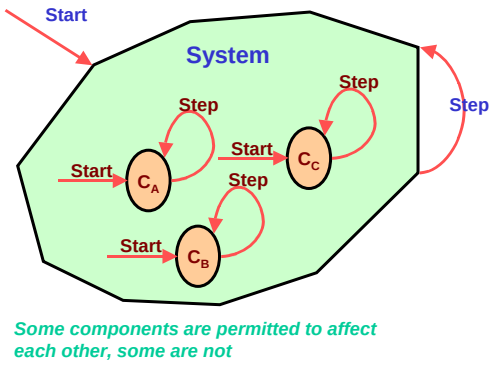



*Figure 3: Abstract version of separation*

In Figure 3, separation is described as the separation of the components of a larger system. The system has two operations, *start*, and *step*. The *start* operation places the system in an initial state, and the *step* operation advances the state of the system, resulting in updates to some or all of the system state.

The system is built from components. Each component has its own *start* and *stop* operation, which initialize and advance the state of the component. The system *start* and *stop* operations are defined in terms of the component *start* and *stop* operations. For

example, the system start operation could invoke the start operation of each of the components of the system. This version of separation requires a *select* operation that returns a component of the system, given the ID of the component. Thus (select s a) denotes component *a* of system *s*.

In this abstract context, an informal statement of the separation property is:

> **Abstract separation property:** System component A is separated from system component B if the results of the Start and Step operations of component A are not influenced by the Start and Step operations of component B. In other words, the operation of component A is the same, no matter how operations by component B are interleaved with the operations of component A.

*Figure 4 Abstract separation property*

This statement maps down to the operating system version of separation (section 2.1). The processes are the components of the system, and the system consists of all the processes, together with the operating system itself. The process operations (such as executing machine language instructions, or making calls to the operating system) become versions of the abstract *Step* operation. The select operation corresponds to the operating system scheduler, which determines the next process to run.

# 3 Informal Description of The Krenz

Like the separation concept, the Krenz concept originated as a property of interest for an operating system. The Krenz concept at the operating system level is described in section 3.1. The Krenz concept is then extended to arbitrary information flow policies in section 3.2.

## 3.1 Krenz concept at the operating system level

The Krenz concept at the operating system level is depicted in Figure 5. The platform controlled by the operating system is divided into different information processing modes. Examples of modes are classified and unclassified, proprietary and public, protected and unprotected. The Krenz information flow policy requires specifies that when information flows between two modes, it must flow through a filter. For example, information flowing from an unclassified mode to a classified mode must be sent through a filter that includes encryption, and possibly a dirty word scan.

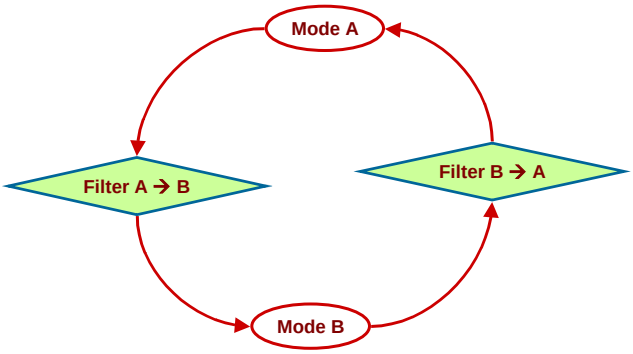



*Figure 5: Informal description of the Krenz*

At the platform level, different realizations of the Krenz are possible:

- The Krenz could be an enhancement to the boot up. The desired mode of processing is determined, and the user is permitted to boot up in the desired mode. Only resources appropriate to that mode are made visible to the user.
- The Krenz could be implemented as an envelope to the underlying platform and operating system. All I/O by the platform, operating system, and applications would be intercepted by the Krenz, which would establish the abstraction of different information processing modes and the filters between them. This realization would permit hot switches between modes.
- The Krenz could be a higher-level software layer, such as VMWare. VMWare is used to control access by the user mode software to the underlying hardware, and therefore to any I/O as well. This realization of the Krenz also permits hot switches between modes.

The Krenz concept depends upon the separation concept. If the only information flow from mode A to mode B is via a specified filter, then the underlying operating system must ensure that there are no flows between mode A and mode B other than via the explicitly allowed communication mechanisms.

## 3.2 Abstraction of the Krenz concept

Like the operating system concept, the Krenz concept can be abstracted away from the operating system context. This abstraction is shown in Figure 6.

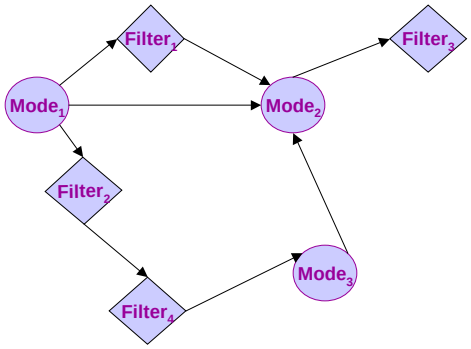



*Figure 6: Extension of the Krenz Concept*

The Krenz concept is formulated as a directed graph.

- **Nodes:** The nodes in the directed graph are either
  - **Modes:** The information processing modes, such as classified or unclassified.
  - **Filters:** The filters between the information processing modes.
- **Edges:** The edges in the directed graph are uni-directional information flows

The filter nodes are trusted to perform their filter functions correctly, whereas the mode nodes are untrusted. As shown in Figure 6, filters can be composed. For example, filter 2 and filter 4 have been composed.

This model represents an abstraction of the Krenz concept from the operating system level, since nodes and edges have no fixed semantics. Nodes could be networks, domains, threads, etc, and the edges the corresponding communication links between the nodes.

In this model, we have chosen to model the filters as nodes. Another choice would be to model the information modes as nodes, and the filters as edges. It is not clear which is a better choice at this time, and both models will be explored in the course of the Programatica project.

## 3.3 The Krenz assurance graph concept

The Krenz concept can be further extended as depicted in Figure 7. The extension is that properties have been assigned to the edges in the Krenz graph. In the Krenz graph, the information processing modes were untrusted, and had no useful assurance properties. In the Krenz assurance graph, an information-processing mode could be trusted to satisfy the specified property on the specified output arc. This permits discussing a limited trust (limited to the specified property) for the information-processing mode. Untrusted modes would have “True” as their output property, meaning that there is no property (other than tautologies) that can be asserted about the output of the information mode.

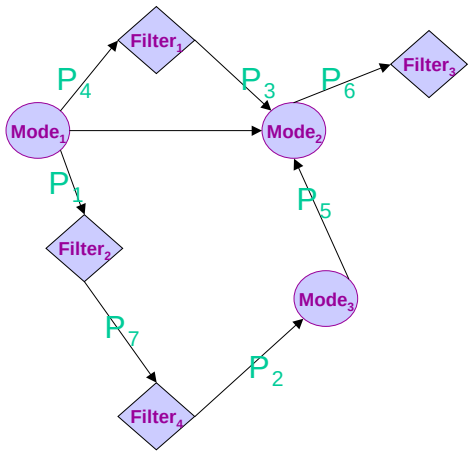



*Figure 7: The Krenz Assurance Graph*

# 4 Hierarchical system concept

The separation and Krenz concepts both started at the operating system level, and were later abstracted to arbitrary information flow situations. With this abstraction, it is tempting to apply the separation and Krenz concepts to other systems requiring security assurance. In particular, we have the goal of applying the Krenz concept to complex systems that span a number of levels of complexity. An example of a complex system spanning many levels is shown in Figure 8.

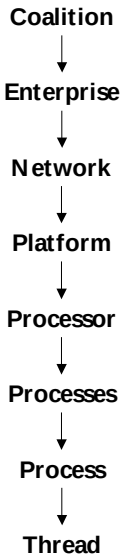



*Figure 8: Hierarchical levels within a complex system*

Each level in the hierarchy has its own separation and krenz concerns. For example, networks can be separated, and connected by filter components such as network encryptors and firewalls. The coalition may be formed from enterprises, and there may be rules (filters) governing the flow of information between the enterprises. The Krenz model should enable the formulation of the following types of questions:

- **Conformance to information flow policy:** Does an architecture, spanning levels from coalition down to thread, conform to a specified Krenz information flow policy?
- **Security of changes to the architecture:** Does a proposed change to the architecture, such as a new process, a new network, a new connection (etc.), still conform to the specified Krenz information flow policy? If not, what filters should be inserted to maintain conformance to the information flow policy?

If the model enables formulation of these questions, then it should also enable automated tools to answer these questions.

## 4.1 Hierarchical System concept at the operating system level

There is already a notion of a hierarchy of systems in the operating system level separation concept, as shown in Figure 9. The system level has *start* and *step* operations, and its components also have *start* and *step* operations. This suggests that a minor

modification to the separation concept is to make each of the components into a separation system, with its own subcomponents.

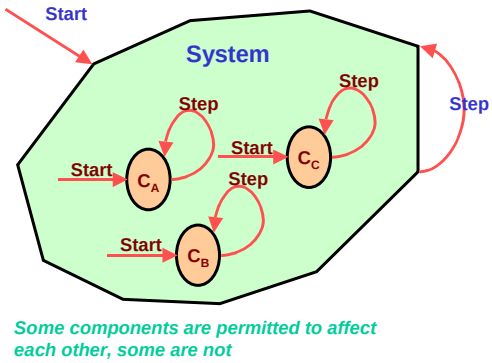



*Figure 9: Recursion in the operating system separation concept (Repeats Figure 1)*

The separation concept is displayed as Haskell classes and signatures in Figure 10. The system level has its own *select*, *start*, and *stop* operations, shown in the upper left of the square. The system has four components, each of which is a system in its own right, having *select*, *start* and *stop* operations.

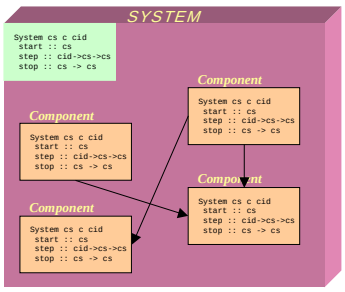



*Figure 10: Separation as Haskell signatures*

## 4.2 Abstraction of the hierarchical system concept

The extension of the separation concept to more levels is depicted in Figure 11. The hierarchy now has three levels, a super system, comprised of fours systems, each of which is comprised of four subsystems. It is clear that this nesting of systems can be extended to any number of levels in the hierarchy.

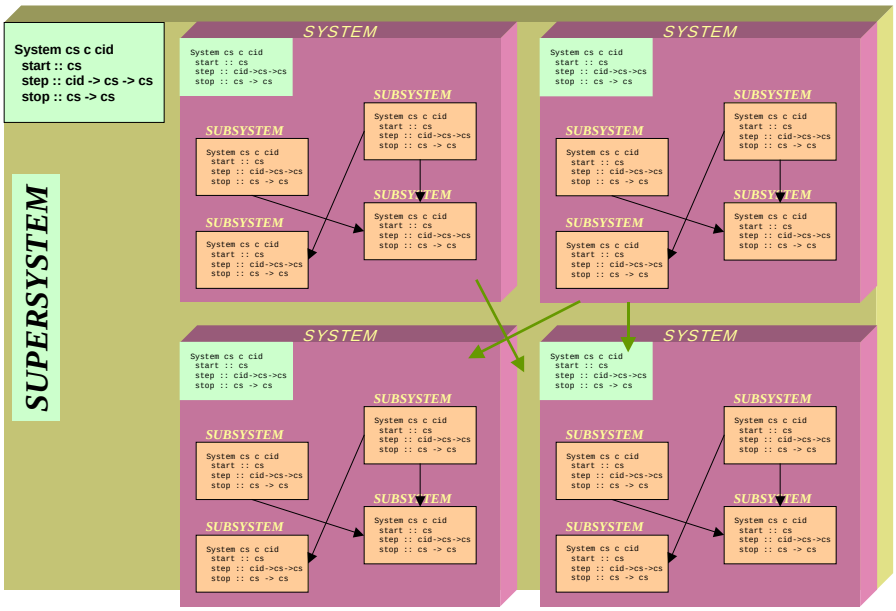



*Figure 11: Extending the separation concept to a hierarchy*

Figure 11 shows extending the separation concept to an arbitrary number of levels. Now we want to extend the Krenz concept to an arbitrary number of levels. The separation concept is presented nicely as nested systems. The Krenz concept has been modeled (in section 3) as a graph, so extending the Krenz concept to an arbitrary number of levels requires a multi level graph. This development is described in the following sections (see sections 5.1 and 5.4).

# 5 Description of Separation / Krenz Hierarchy

This section informally describes the hierarchy of Haskell classes and instances used to achieve the objectives stated in section 1. The description begins with the hierarchical graph class in section 5.1. Then the dynamical system specification is described in section 5.2. The use of parameters to make Separation, Krenz, and Krenz assurance specifications is described in section 5.3. The hierarchy of classes and instances is then described in section 5.4.

## 5.1 Recursive Graphs

The recursive graph data structure is used to capture the hierarchical separation and Krenz concepts discussed in section 4. The recursive graph data structure is illustrated in Figure 12.

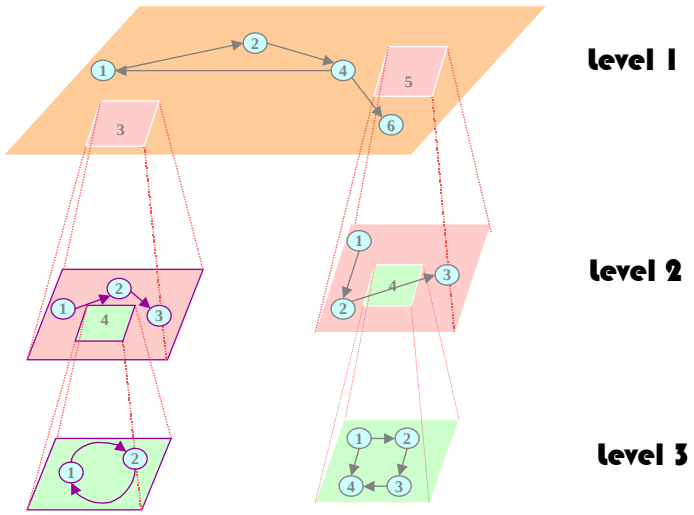



*Figure 12: Hierarchical Graph*

The example recursive graph shown in Figure 12 has three levels. The highest level (level 1) has three simple nodes, labeled 1, 2, 4, and 6, and two complex nodes, labeled 3 and 5. Edges are shown between some of the simple nodes in the level 1 graph. The complex nodes are recursive graphs themselves. The level 1 node labeled 3 is a recursive graph at level 2, having simple nodes 1, 2, and 3, and a complex node labeled 4. The rest of the recursive graph is described similarly.

All of the edges shown in Figure 12 are edges between simple nodes of the same level. In addition, it is possible to have edges between the nodes at different levels in the recursive graph, or across levels in the recursive graph, as shown in Figure 13.

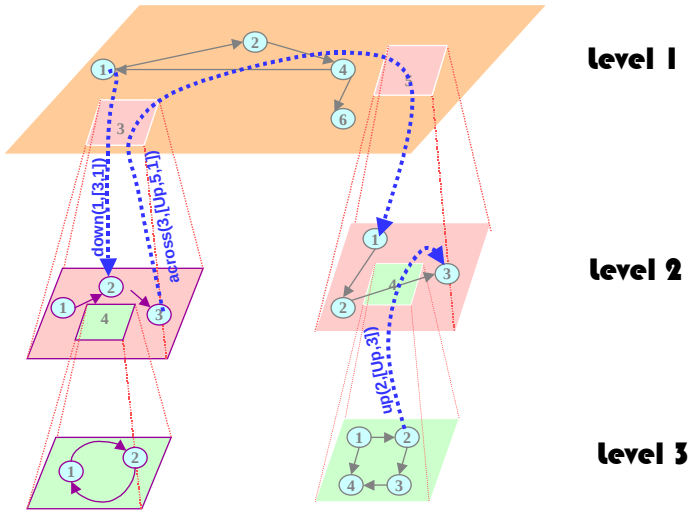



*Figure 13: Inter-level edges in the recursive graph*

The graph of Figure 13 has the following inter-graph edges:

- Down: An edge down, from node 1 at level 1 to node 2 in subgraph 4.
- Up: An edge up, from node 2 in the right subgraph at level 3, to node 3 in the right subgraph at level 2.
- Across: An edge across from node 3 in the left subgraph at level 2 to the right subgraph at level 2. Note that the edge is specified as the list [Up, 5, 1]. This specification means to go up one level, then to node 5 at that level (which turns out to be a complex node), and then to node 1 inside the complex node.

The Haskell class declaration for the recursive graph class contains a *flatten* and a *deepen* function. The flattened version of the recursive graph of Figure 13 is shown in Figure 14. The recursive graph has been flattened to have only one level. The complex nodes are expanded into the level above them.

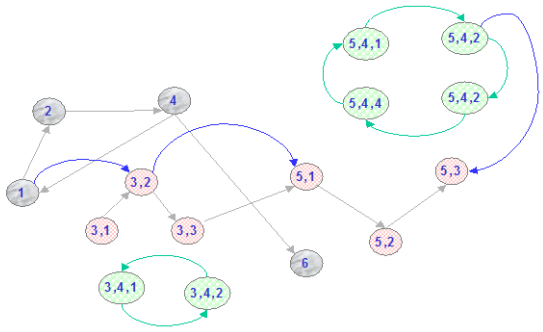



*Figure 14: Flattened Hierarchical Graph*

Note that the structure of the recursive graph of Figure 13can be recovered from the flattened graph of Figure 14, because the information about the structure of the graph was stored in the names of the nodes. For example, the node named 3,4,1 indicates simple node 1 of complex node 4, of complex node 3 at the top level.

The recursive graph is the framework for defining the separation, Krenz, and Krenz assurance graph concepts that span levels of complexity. The use of this framework will be described in section 5.4.

## 5.2 Dynamic Systems

As stated in section 1, an objective of these specifications is to enable a dynamic separation kernel, a dynamic Krenz system, and a dynamic Krenz assurance system. This means that nodes and edges can be added to or deleted from an instance of separation, Krenz, or Krenz assurance graph. At present, the dynamic addition and deletion of nodes is defined only in the simplest version of the Separation specification. Extending this capability to the hierarchical separation and Krenz specification requires only the addition of functions to the recursive graph specification that will permit the addition and deletion of nodes and edges.

## 5.3 Parameterizing Separation vs. Krenz

The recursive graph specification (section 5.1) will be used to capture the separation, Krenz, and Krenz assurance graph concepts. This is done by parameterizing the recursive graph data structure. The parameters are called freight, and the freight is carried by each node and edge in the graph structure. The freight has the following characteristics:

- **Node Freight:** The simple nodes carry the freight, not the complex nodes (see Figure 13). However, the simple nodes that comprise a complex node can carry freight. If needed, freight can be added to the complex nodes if this extension to the specification is needed.
- **Edge Freight:** The edges can also carry freight.
- **Polymorphic Freight:** The freight is specified as a polymorphic parameter, so that the nodes and edges can carry freight of any type. The node freight parameter is different from the edge freight parameter, so the nodes and edges can carry freight of different type.

As noted in section 1, the polymorphic parameter implies that all the nodes at all the levels of the graph have freight of the same polymorphic type. An objective of future versions of this specification is to lift this restriction, so that different subgraphs of a recursive graph can have node freight of different types.

## 5.4 Hierarchical Separation and Krenz

The separation, krenz, and krenz assurance graph specifications can be realized as data types that are instances of the appropriate classes.

- Separation: Each of the separation, krenz, and krenz assurance graph specifications are instances of the separation class. From this class the inherit the concept of a system that can be initialized (the start operation), advanced (the step operation), and they inherit the separation property. The separation class is described in section 5.4.1.
- **Recursive graph:** From the recursive graph class, the separation, krenz, and krenz assurance graph specifications inherit the hierarchy of nodes and subgraphs. The recursive graph concept was described in section 5.1, and the specifications will be described in section 5.4.2.

By combining inheritance from the separation class and the recursive graph class, the specifications gain the separation concept, applied at each level of the hierarchy given in the recursive graph. It is this combination which permits us to achieve the objectives of the specifications laid out in section 1.

### 5.4.1 System and Separation classes

The system and specification classes are built up from lower level classes as shown in Figure 15.

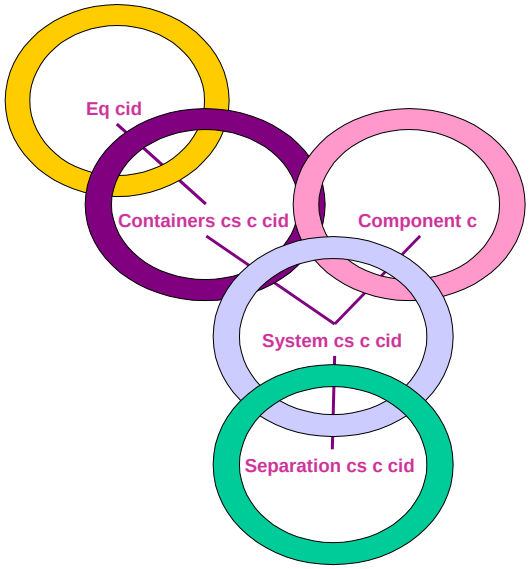



*Figure 15: Separation Specification Hierarchy Classes*

A brief description of each of the classes of Figure 15 follows:

- Eq: The equality class comes from the standard Haskell prelude. It contains equality and inequality operators.
- Component: The component class defines a basic component of a system, with its own *start*, *step*, and *stop* operations. These operations are sufficient to describe an object with an internal state that can be initialized (start), advance (step), and terminate (stop).
- Containers: The container class defines the basic operations of any dynamic container. The elements in the container each have an identifier. The basic operations are:
  - **select:** Given an id, find the element (if any) of the container that has that identifier.
  - **addElem:** Add an element (with its id) to the container.
  - **deleteElem:** Delete and element (with its id) from the container.
- System: The system class defines the concept of a *system* that *contains components*. Thus the system class inherits from both the containers class and the component class. The system class adds the idea that both the system level and the component level have start, stop and step operations.
- **Separation:** In terms of standard Haskell, the separation class is exactly equal to the system class. In terms of Programatica, the separation class adds properties to the system class. The separation property is shown in Figure 16.

```
property FirstSeparation = All ps. All x. All y.
        select (stepSystem ps x) y =/= select ps y ==>
            ((interactionMatrix ps x y) ps x y === True)
property SecondSeparation =
  All ps. All x. All y.
    select (stepSystem ps x) y =/= select ps y ==>
      (Exists f.
        select (stepSystem ps x) y === f (select ps x) (select ps y))
```

*Figure 16: The Haskell separation properties*

### 5.4.2 The Recursive Graph class

The recursive graph class, like the separation class, is constructed from more primitive classes. The hierarchy of classes leading to the recursive graph class is shown in Figure 17. The recursive graph class inherits from the node and edge classes, which define the minimum characteristics of nodes and edges, respectively. Both nodes and edges are defined in terms of paths, inherited from the path class. The nodes require paths, because the ID of a node can be a path. This capability is used when a recursive graph is flattened, as described in section 5.1. The id of a node in the flattened graph is a path that contains enough information to reconstruct the recursive graph from the flattened graph.

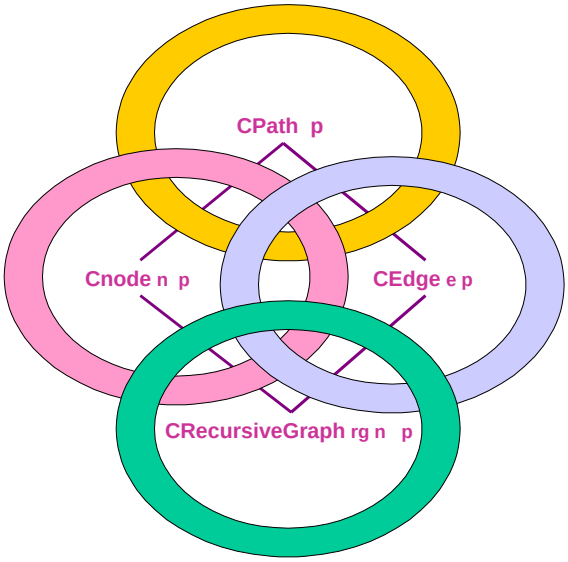



*Figure 17: Recursive Graph Hierarchy of Classes*

A brief description of each of the classes of Figure 17 follows:

- CPath: The archetype of the path class is the list.
  - `isup :: p -> Bool:` Determine if the path is an "Up" path, from a lower level in the graph to a higher level.
  - `concatpath :: p -> p -> p:` Concatenate two paths

- `restpath :: p -> p:` Determine the remainder of the path, after the first element is removed.
- `pathlength :: p -> Int:` Determine the length of the path.
- `matchfirst :: p -> p -> Bool:` Determine if the first elements of two paths match.
- `addup :: p -> p:` Add an "Up" element to the front of the path.
- `emptypath :: p:` Construct an empty path.

- CNode: The node type of the graph. It must have at least the following functions defined:
  - `pathofnode :: n -> p:` Determine the path of the node.
  - `constructnode :: n -> p -> n:` Construct a node from an input node and a path.
  - `newnode :: p -> n:` Create a new node from a path.
- CEdge: The edge type of the graph. There are several utility functions defined on the graph class. The two most important functions are:
  - **flatten:** Flatten as recursive graph, as described in section 5.1.
  - **deepen:** Deepen a flattened recursive graph, as described in section 5.1.

Because flatten and deepen are defined on the graph class, anything than can inherit from the graph class gets flatten and deepen for free. This is used to provide flatten and deepen for the separation, krenz, and krenz assurance graph specifications in later sections.

### 5.4.3 Graph with properties instances

The first construction performed uses the recursive graph class, described in section 5.4.2. It adds freight to the nodes and edges of the recursive graph, and this freight is then used to enable the nodes and edges of the recursive graph to have associated properties. The instantiation of the recursive graph class to establish the graph with freight (and hence the graph with properties) is shown in Figure 18.

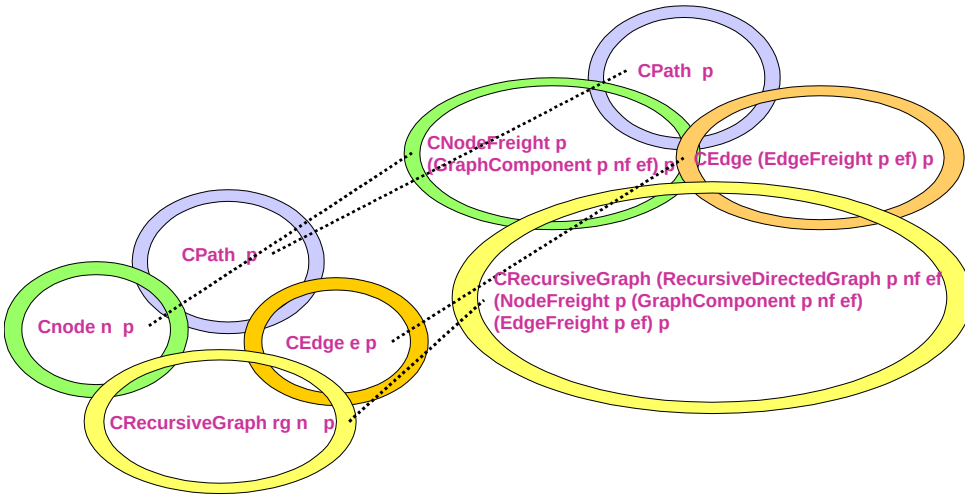



*Figure 18: Recursive Directed Graph as an instance of the Recursive Graph*

The recursive directed graph is the instance of the recursive graph that carries freight on its nodes and edges. Because it is an instance of the recursive graph class, and because

the recursive graph class inherits (via the Haskell class constraint mechanism) from Cnode and Cedge, the implies some requirements on the recursive directed graph instance. In particular the node type of the recursive directed graph instance (NodeFreight p (GraphComponent p nf ef)) must be an instance of Cnode. Furthermore, the edge type of the recursive directed graph (EdgeFreight p ef) must be an instance of the edge class. Figure 18 shows these requirements upon the recursive directed class instance.

The recursive directed graph is specialized to the Graph Property data structure, in which the node freight and edge freight have been specialized to node properties and edge properties, respectively. The node and edge properties both incorporate the type Prop, which is the Programatica tool 0 type for program properties. The graph with properties data structure is then made an instance of the separation class, as shown in Figure 19.

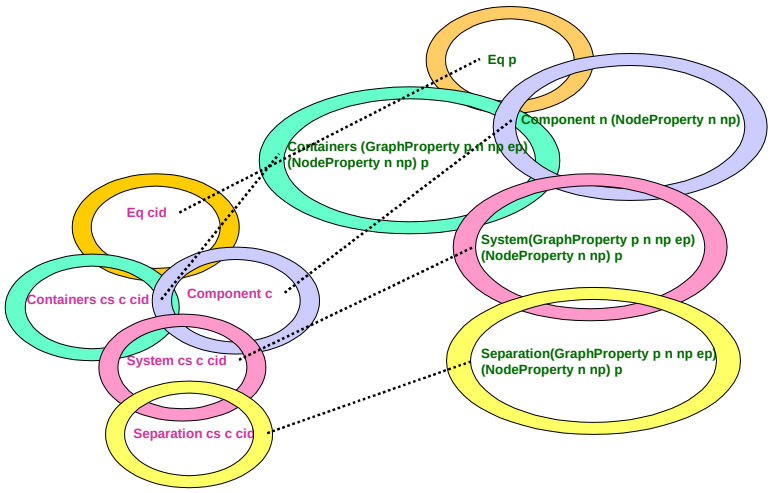



*Figure 19: A graph with properties, as an instance of the Separation class*

Because the graph with properties data structure is defined using the recursive graph class (via the recursive directed graph instance), and using the separation class, it inherits from both. This data structure has the hierarchical structure of the recursive graph, properties carried by each node and edge in the recursive graph, and the separation property applied to each node in the recursive graph. This data structure is rich enough to support separation, krenz, and krenz assurance instances.

### 5.4.4 Krenz System instances

The krenz system is defined using the graph property data structure. The node property is defined as a flag that determines if the node is a filter or not. If the node is a filter, then a filter property is associated with the node. This achieves the abstraction described informally in section 3.2. Because the Krenz is a version of the graph property data structure, the Krenz, like the graph property data structure, becomes an instance of the separation class. The fact that the Krenz system is an instance of the separation class implies some other class / instance relationships, which are shown in Figure 20.

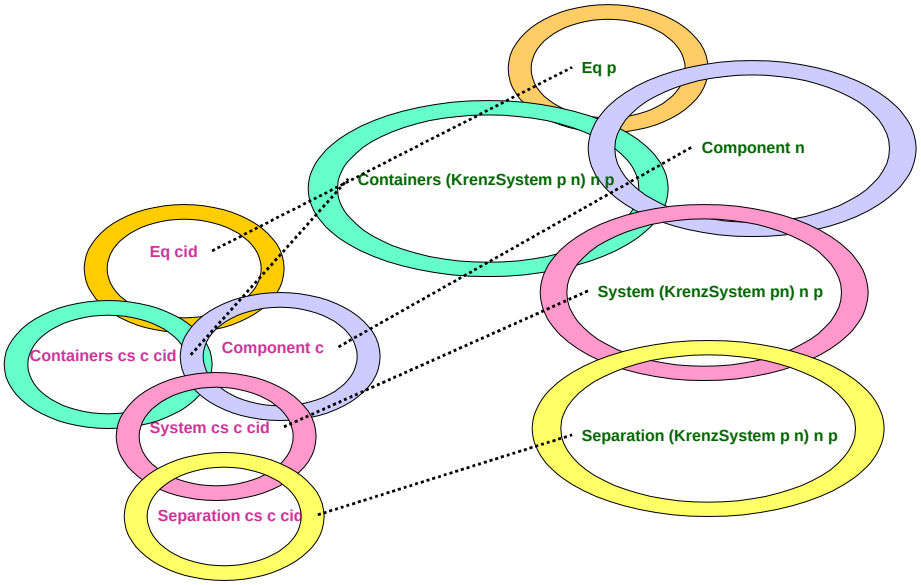



*Figure 20: Krenz system as an instance of the separation class*

### 5.4.5 Krenz Assurance Graph instances

The Krenz Assurance Graph is a slight modification to the Krenz System (section 5.4.4). The Krenz system makes no use of the edge properties of the underlying recursive directed graph. The Krenz Assurance Graph uses the edge freight to carry edge properties in order to realize the abstraction described in section 3.3. The instantiation requirements of the Krenz assurance graph are similar to those of the Krenz System. The requirements for the Krenz assurance graph are shown in Figure 21.

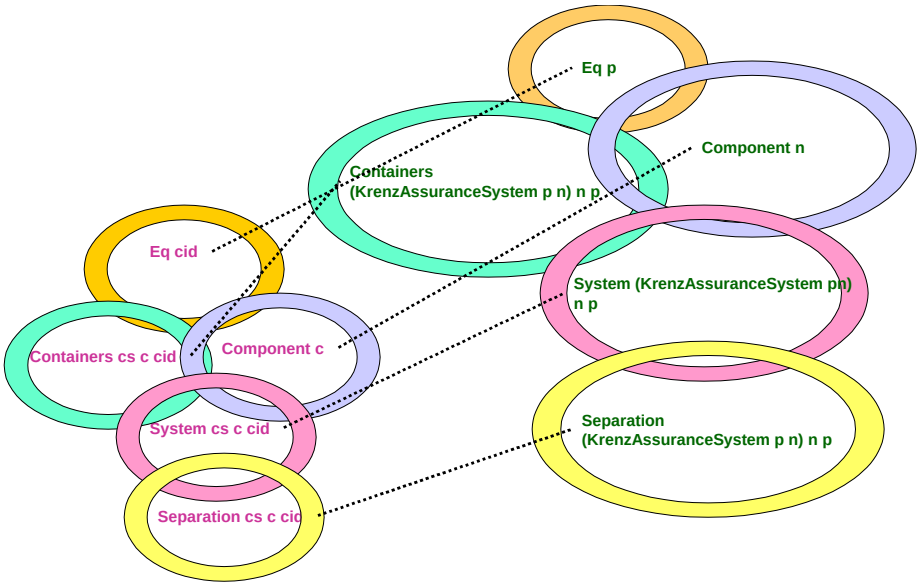



*Figure 21: Krenz Assurance System Instances*

# 6 Topology (and the Enterprise Krenz)

This section describes (informally) the concept of topology that is layered on top of the Separation and Krenz specifications. Section 6.1 describes the Grothendieck topology concept in the context of the Separation specification, section 6.2 describes the Grothendieck topology concept in the context of the Krenz specification, and section 6.4 describes the Grothendieck topology concept in the context of the Krenz assurance graph specification.

The new Haskell source for the specification containing categories, sieves, and Grothendieck topologies, is in section 0. The Haskell specification that makes the recursive graph (and recursive graph homomorphims) an instance of the category class is given in section 7.12.

## 6.1 Topology and pattern matching for the Separation specification

The first concept of topology for instances of the separation specification is shown in Figure 22. There is an instance of separation (called the pattern), which consists of two nodes ($Node_1$ and $Node_2$), and a single arrow between them. There is another instance of separation (the target), which has three nodes, with several arrows between them.

- Node n can match collection of nodes [m], together with the edges between nodes in [m]
- When n matches N, and m matches M, then the edge (n,m) can match all the edges from N to M

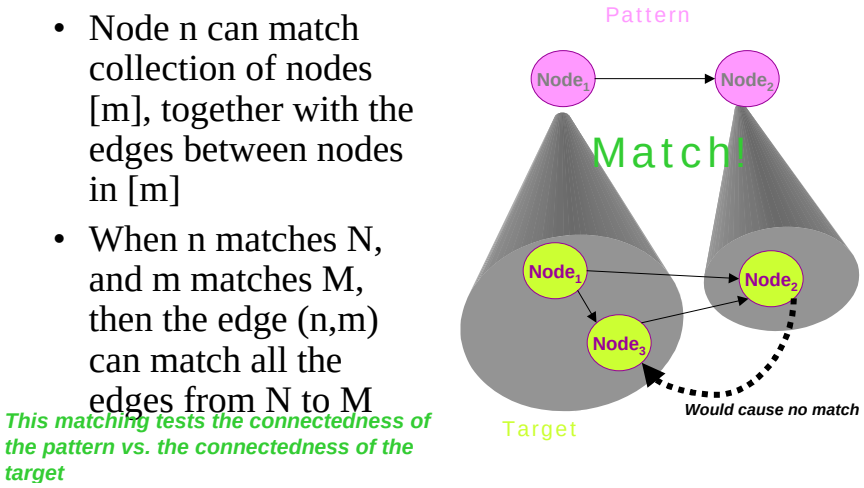



*Figure 22: Separation / Subgraph Matching*

Suppose the pattern of Figure 22 is a separation specification, and the target is the configuration of several processes running on a separation kernel. Then the pattern indicates that there are two domains, and communication is permitted to flow in one direction from the domain represented by $Node_1$ to the domain represented by $Node_2$. The target indicates that there are three processes, with the separation kernel permitting interprocess communications as shown in the figure. With this interpretation, Figure 22 shows one way of grouping the processes such that they form an instance of the pattern specification. If $Node_1$ and $Node_2$ of the target are grouped together to match $Node_1$ of the pattern, and $Node_3$ of the target is used to match $Node_2$ of the target, then the only flows from the processes matching $Node_1$ of the pattern are flowing to the process(es) matching $Node_2$ of the pattern. Thus there is a match based upon these choices. If the dashed arrow is added to the target process configuration, then the choices made to not

constitute a match to the pattern. However, with the dashed line added, other choices can be made, as shown in Figure 23. With these choices, mapping $Node_2$ and $Node_3$ of the target to $Node_1$ of the pattern, and $Node_1$ of the target to $Node_2$ of the pattern, then there is once again a pattern match.

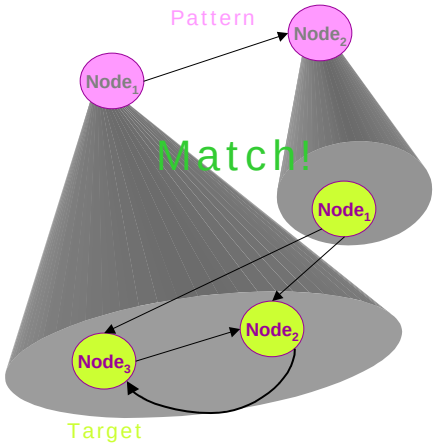



*Figure 23: Retry on Separation pattern matching*

These examples demonstrate that the pattern matching requires choices, and some of the choices may result in a match, while other choices do not result in a match.

The basic notion of pattern matching here is that of an embedding of one instance of separation into another instance of separation. A more general example is shown in Figure 24, Here there are three instances of the separation specification, and embeddings are shown from one instance (the pattern instance), to the second instance (the intermediate instance), to the third instance (the target instance).

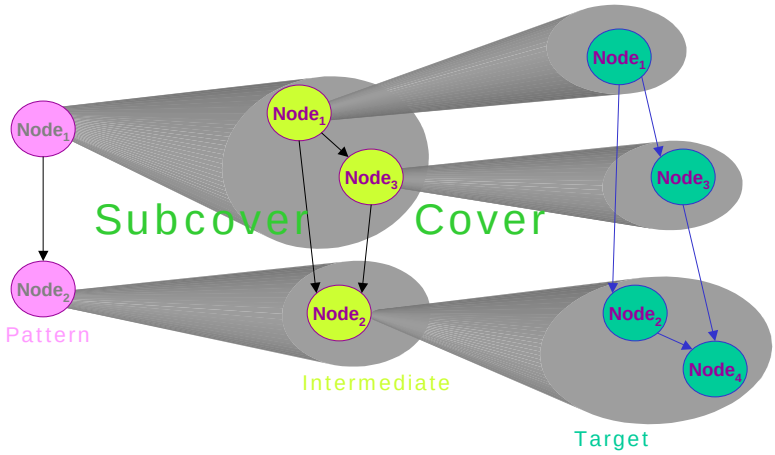



*Figure 24: Separation / Subgraph Site*

The two embeddings of Figure 24 can be composed into another embedding, which is shown in Figure 25. This demonstrates that the embeddings are mappings that can be

composed. The preserve structure in that the structure of the pattern still exists in the target, via grouping of nodes and arrows in the target.

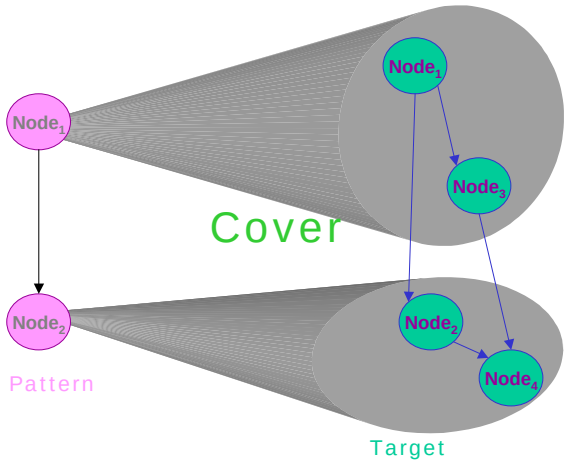



*Figure 25: Composed embedding*

## 6.2 The categorical framework for pattern matching (Separation)

There are two levels at which the concept of category might be applied:

- Intra-instance: Within an instance of separation, each node can be viewed as an object in a category, while the connections between nodes can be viewed as arrows in a category.
- Inter-instance: Each instance of separation is considered an object in the category, and each homomorphism between instances is considered an arrow in the category.

The first case (intra-instance) is not useful here. The arrow in the category represents the relation “directly communicates”. The composition of two arrows may no longer represent a “directly communicates” path supported by the underlying separation kernel. The second case is what is described in section 6.1, and this is the basis for defining the category **Sep** as follows:

- Objects: The objects are instances of the separation specification.
- Arrows: The arrows are homomorphisms between instances of separation.

With this definition of the category Sep, the machinery of a Grothendieck topology, including sieves, sheaves, and hom sets, can be defined. This machinery is built up in the Haskell file Category.hs (see section 0).

## 6.3 Topology and pattern matching for the Krenz specification

The concept of a Grothendieck topology can be defined for instances of the Krenz system specification in the same way as it was defined for the separation specification (section 6.1). An example of pattern matching, in the same spirit as discussed in section 6.1 is illustrated in Figure 26. In the pattern match attempt shown, $Node_1$ of the target is

matched to $Node_1$ of the pattern, $Filter_1$, $Filter_2$ and $Filter_4$ of the target are matched to $Filter_1$ of the pattern, and $Node_2$ and $Node_3$ of the target are matched to $Node_2$ of the pattern. In this case, the match has failed, because $Filter_3$ of the target is left unaccounted for. In the pattern match attempt shown, there is no way to match the target to the pattern. If $Filter_3$ is deleted from the target, then the pattern match shown succeeds.

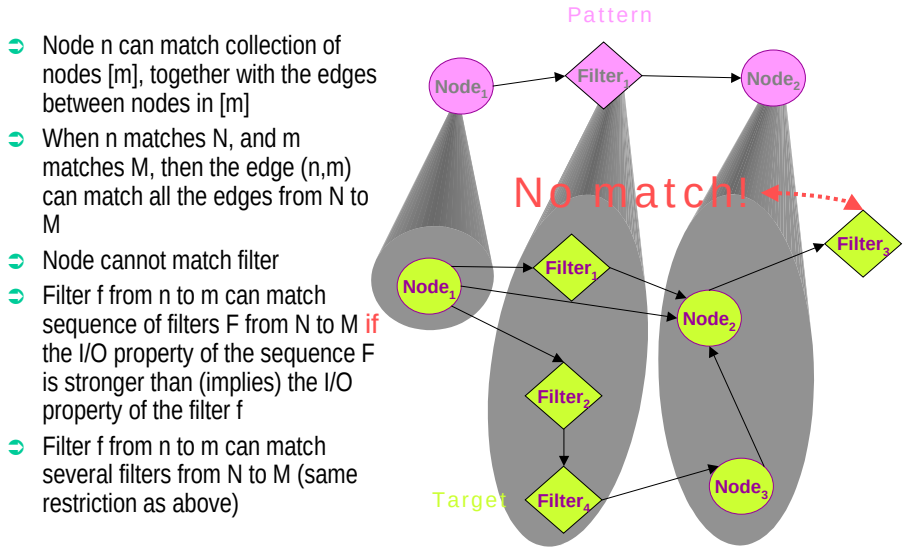



*Figure 26: Krenz System Matching*

The embeddings of instances of Krenz system specification can be composed, as with embeddings of instances of the separation specification. The second cover fails, but with $Filter_3$ of the target deleted, the second cover succeeds. Note that "cover" is a special type of Krenz system homomorphism.

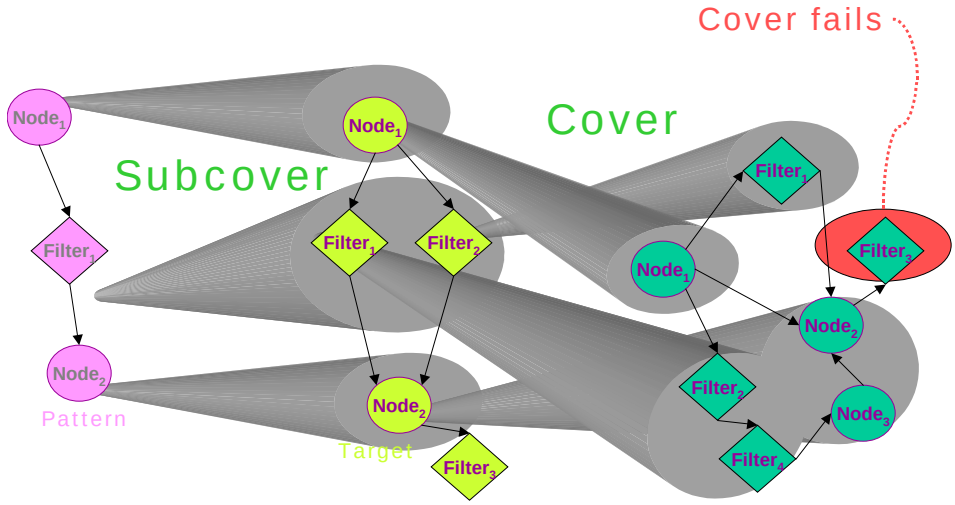



*Figure 27: Krenz System Site*

The composition of the two covers in Figure 27 is shown in Figure 28.

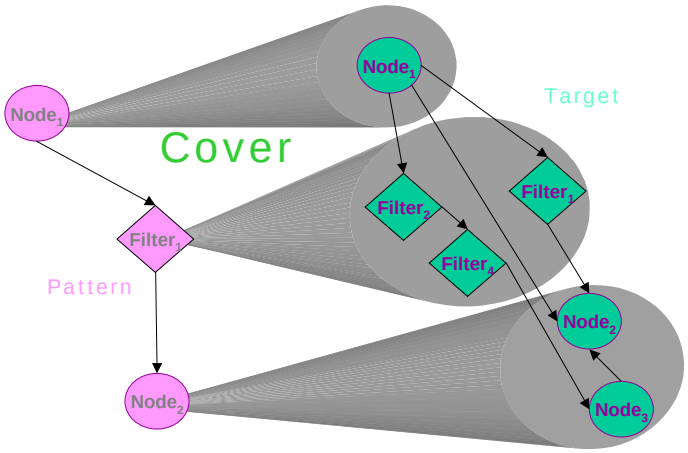



*Figure 28: Composed cover for a Krenz system*

The Krenz specification is the basis for a category **Krenz** as follows:

- **Objects:** The objects are instances of the Krenz system specification.
- **Arrows:** The arrows are homomorphisms between instances of the Krenz system specification.

## 6.4 Topology / pattern matching for Krenz assurance specification

The concept of pattern matching in a Krenz Assurance Graph (KAG) is defined in the same spirit as the pattern matching in section 6.1 and section 6.2. The concepts of embedding and composition of embeddings carries through as with the previous cases.

- Same rules as Krenz with additions for the assurance properties ($P_j$)

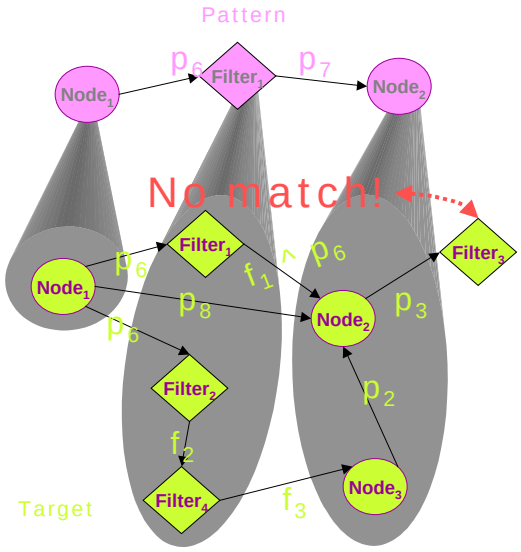



*Figure 29: Krenz Assurance Site*

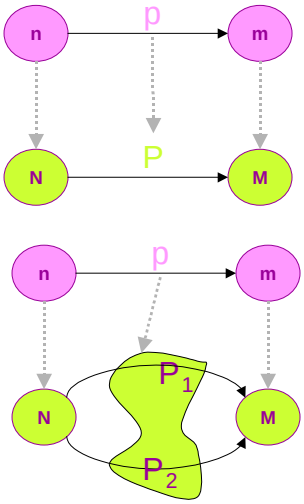
Based on the pattern matching concept shown in Figure 30, the category **KAG** can be defined as follows:

- Objects: Instances of the Krenz assurance graph specification.
- Arrows: Homomorphisms (and the special cases of embeddings) between instance of the Krenz assurance graph specification.

Structurally, the pattern match for Krenz assurance graphs shown in Figure 29 is the same as the pattern match shown in Figure 26. The pattern match in the Krenz Assurance Graph system carries with it additional requirements. These additional requirements are discussed in 6.4.1.

### 6.4.1 Matching the assurance properties in the Krenz Assurance Graph

As shown in the previous examples, a node in the pattern may match several nodes in the target, and an arrow in the pattern may match several arrows in the target. In the category KAG, the arrows carry assurance properties, and there are requirements on the assurances for a match between an arrow in the pattern and a group of arrows in the target. These requirements are illustrated in Figure 30.




- If n ➔ N and m ➔ M then p ➔ P only if P is stronger the p, I.e. P => p
- If n ➔ N and m ➔ M then we must have $P_1$ or … or $P_n$ => p (the disjunction of the refinements implies assurance proposition

*Figure 30: Refining an Assurance*

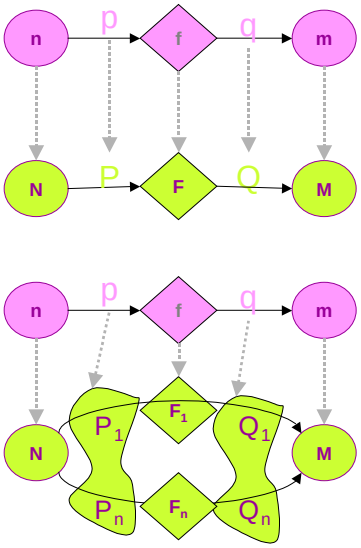
The requirement is that the disjunction of the assurance requirements of the arrows in the target must be at least as strong as the assurance requirement of the arrow in the pattern. This requirement is stated more formally in Figure 31.

**Krenz assurance matching property:** $P_1 \vee \ldots \vee P_n \Rightarrow p$

*Figure 31: Krenz assurance matching property*

This means that when a Krenz assurance graph is refined, the properties assigned to the arrows in the refined (more detailed) graph must be strong enough to imply the properties in the unrefined (less detailed) graph.

A similar requirement applies to the filter properties in the Krenz assurance graph. This additional requirment is depicted in Figure 32.




- Refinement requirement: P and F and Q => p and f and q
- Refinement requirement: ($P_1$ and $F_1$ and $Q_1$) or … or ($P_n$ and $F_n$ and $Q_n$) => (p and f and q)

*Figure 32: Refining an Assurance and a Filter*

This requirement takes into account not only the property associated with the filter, but also the assurance properties associated with the arrows to and from the filter. The filter is viewed as an intermediary between two nodes, and the total property seen between the two nodes is the conjunction of the two arrow assurance properties with the filter property. With this understanding of the "property of a filter", the requirement on the filters in the refinement is that the disjunction of the properties of the filters in the target that refine a filter in the pattern must be at least as strong as the property of the filter in the pattern. This requirement is stated more formally in Figure 33.

**Krenz filter matching property:** $P_1 \vee \ldots \vee P_n \Rightarrow p$

*Figure 33: Krenz filter matching property*

## 6.5 Axioms of a Grothendieck Topology

This section illustrates the definition of a Grothendieck topology. This section is based on the PhD thesis of Srinivas (8). A Grothendieck topology **(C, J)** has the following two components:

- Category C:
- Cover J: The function J assigns to each object a of C a set J(a) of sieves on a. The elements $R \in J(a)$ are called *covers* of a.
- The Grothendieck Topology satisfies the following three axioms:
- Identity cover: For every object a of C, the maximal sieve $\{ f \mid cod(f) = a \} \in J(a)$.
- Stability under change of base: If $R \in J(a)$ and $f : b \rightarrow a$ is an arrow of C, then the sieve $f^*( R ) = \{ g : c \rightarrow b \mid f \circ g \in R \}$ is in J(b). This axiom is illustrated by Figure 34.

- Stability under refinement: If R ∈ J(a) and S is a sieve on a such that for each arrow f : b → a in R, we have f*(S) ∈ J(b), then S ∈ J(a). This axiom is illustrated by Figure 35.

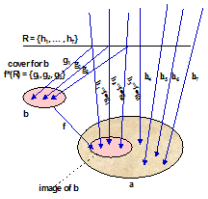


*Figure 34: Stability under a Change of Basis (from Srinivas Thesis [8])*

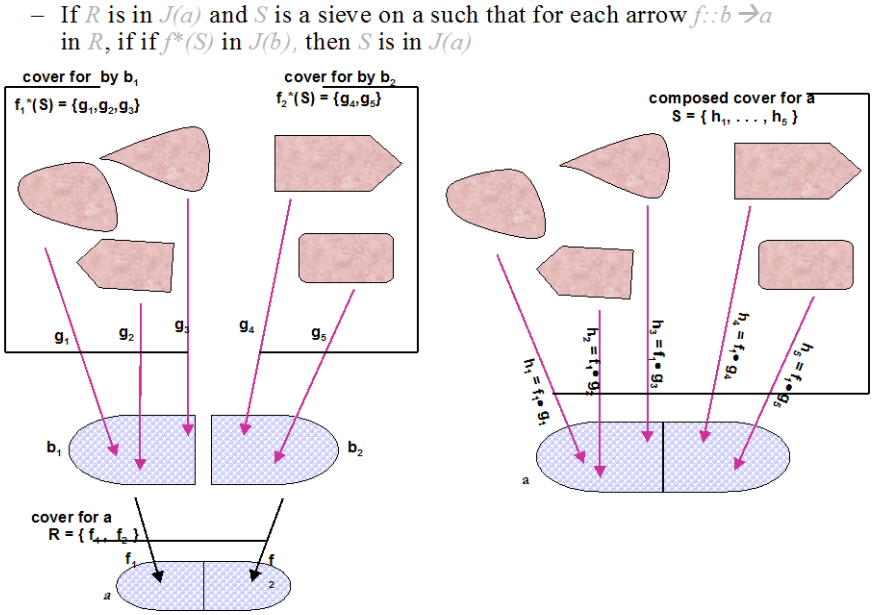



*Figure 35: Stability under Refinement (from Srinivas thesis [8])*

# 7 Haskell Source

This section contains the Haskell source code for the current version of the specifications. Each subsection of this section contains one Haskell source file. The files were compiled using Programatica tool 0, which is a variant of Haskell 98.

## 7.1 Container.hs

```
module Container
  (
```

```
  Containers (select, isAssigned, addElem, deleteElem),
  AssignedNew,
  AssignedDelete
  )
  where

import List

class (Eq cid) => Containers cs c cid | cs -> c, cs -> cid where
    select :: cs -> cid -> Maybe c
    isAssigned :: cs -> cid -> Bool
    -- Defined it using "case" to avoid requiring (Eq c)
    isAssigned cs cid =
      case select cs cid of
        Nothing -> False
       Just ida -> True
    addElem :: cs -> cid -> c -> cs
    deleteElem :: cs -> cid -> cs
    emptyCont :: cs
    elements :: cs -> [c]
    idelements :: cs -> [(cid,c)]

-- An assigned pid cannot be created again
property IsAssigned cs cid = (isAssigned cs cid) === True -- HACK HACK HACK
property AssignedNew =
   All cs cid c. IsAssigned cs cid ==> (addElem cs cid c === cs)

-- An unassigned pid cannot be deleted
property NotIsAssigned cs cid = not (isAssigned cs cid) === True -- HACK
property AssignedDelete =
  All cs cid. NotIsAssigned cs cid ==> deleteElem cs cid === cs
```

## 7.2 RecursiveContainer.hs

```
module RecursiveContainer
  (
  IdList,
  RecursiveContainer,
  Containers,
  )
  where

import Container
import List

class (Containers cs c cid) => RecursiveContainer cs c cid where
    mkComplex :: cid -> [c] -> Maybe cs
    mkSimple  :: [(cid,c)] -> cs
    isComplex :: cs -> Bool
```

## 7.3 System.hs

```
module System
  (
  Containers (..),
  Component (..),
  System (..)
  )
  where

import Container

class Component c where
    stepComponent :: c -> c
```

```
    stopComponent :: c -> c
    startComponent :: () -> c

class (Component c, Containers cs c cid) => System cs c cid where
    startSystem :: cs
    -- Cannot inherit Process for ps CUZ step for ps requires the extra
    -- parameter. Is there a way we could do this with a property?
    stepSystem :: cid -> cs -> cs
    stepSystemN :: cs -> cid -> Int -> cs
    stepSystemN cs cid 0 = cs
    stepSystemN cs cid 1 = stepSystem cid cs
    stepSystemN cs cid (n+1) = stepSystem cid (stepSystemN cs cid n)
    stopSystem :: cs -> cs

-- Stepping a stopped system causes no change
property StopProp = All x. All y. stepSystem x (stopSystem y) === x

-- The fibration of the system
property Fiber = All cs. All cid. All c.
  select cs cid === Just c ==>
    select (stepSystem cid cs) cid === Just (stepComponent c)
```

## 7.4 Separation.hs

```
module Separation where

import System

class System cs c cid => Separation cs c cid where
    -- This generalizes the interaction matrix, so that each pair of
    -- processes can have their own predicate determining what is valid
    -- communication between them.
    interactionMatrix :: cs -> cid -> cid -> (cs -> cid -> cid -> Bool)

property FirstSeparation = All ps. All x. All y.
        select (stepSystem ps x) y =/= select ps y ==>
       ((interactionMatrix ps x y) ps x y === True)
property SecondSeparation =
  All ps. All x. All y.
    select (stepSystem ps x) y =/= select ps y ==>
      (Exists f.
        select (stepSystem ps x) y === f (select ps x) (select ps y))
-- NOTE: Second separation axiom can be stated without higher order.
--  property forall ps1 ps2.select (steps ps1 x) y /=
--                          select (steps ps2 x) y ==>
--                             (select ps1 x /= select ps2 x ||
--                              select ps1 y /= select ps2 y)
```

## 7.5 Maybe2.hs

```
module Maybe2
    (
    foldrMaybe,
    composeMaybe
    ) where

foldrMaybe :: (a -> b -> Maybe b) -> b -> [a] -> Maybe b
foldrMaybe f z [] = Just z
foldrMaybe f z (a:as) =
    case foldrMaybe f z as of
      Nothing -> Nothing
      Just b -> f a b
```

```
composeMaybe :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
composeMaybe f g =
    \x -> case f x of
            Nothing -> Nothing
           Just y -> g y
```

## 7.6 GraphInductive.hs

```
module GraphInductive (
    RecursiveGraph (..),
    RecursiveContext (..),
    RecursiveNode (..),
    RecursiveEdge (..),
    listRecursiveNodes,
    listRecursiveEdges,
    NodeName,
    nodeName,
    nodeLabel,
    nullNodeName,
    matchNodeName,
    match,
    mapGraph,
    findEdge,
    flatten,
    deepen,
    insEdgeTo,
    insNode,
    insNodes
    ) where

import List
import Maybe2

-----------------------------------------------------------------------
-- Recursive graph, defined inductively
-----------------------------------------------------------------------
type NodeComponent = Integer
type NodeName = [NodeComponent]

-- Get the longest common prefix of two node names
commonPrefix :: NodeName -> NodeName -> NodeName
commonPrefix xs [] = []
commonPrefix [] ys = []
commonPrefix (x:xs) (y:ys) | x == y = x:(commonPrefix xs ys)
commonPrefix (x:xs) (y:ys) | x /= y = commonPrefix xs ys

-- Test if a node is a local node
isLocal :: NodeName -> Bool
isLocal nn = length nn == 1

nullNodeName = [] :: NodeName

data RecursiveNode a b =
     SimpleNode    NodeName a |
     RecursiveNode NodeName (RecursiveGraph a b) a
     deriving (Eq)

instance (Show a, Show b) => Show (RecursiveNode a b) where
    show (SimpleNode nn a) = decorate ["(", ",", ")"] [show nn, show a]
    show (RecursiveNode nn sg a) =
       decorate ["(", ",", ",{{", "}})"] [show nn, show a, show sg]
```

```
nodeLabel :: RecursiveNode a b -> -- Node to access
             a                    -- Returned node label
nodeLabel (SimpleNode nn a) = a
nodeLabel (RecursiveNode nn subgraph a) = a

nodeName :: RecursiveNode a b -> -- Node to access
            NodeName             -- Returned node name
nodeName (SimpleNode nn a) = nn
nodeName (RecursiveNode nn subgraph a) = nn

-- Make a node flatter
nodeUp :: RecursiveNode a b -> -- The node to deepen
          NodeName ->          -- Additional node components
         RecursiveNode a b
nodeUp (SimpleNode nn a) up = SimpleNode (up ++ nn) a
nodeUp (RecursiveNode nn subgraph a) up = RecursiveNode (up ++ nn) subgraph a

-- Determine if a node is recursive
isRecursive :: RecursiveNode a b -> -- Node to query
               Bool                 -- True if recursive
isRecursive (SimpleNode _nn _a) = False
isRecursive (RecursiveNode _nn _sg _a) = True

-- Print some information with punctuation as decoration
decorate :: [String] -> [String] -> String
decorate punctuation xs | length punctuation == length xs =
    concat (zipWith (++) xs punctuation)
decorate punctuation xs | length punctuation == length xs + 1 =
    concat (zipWith (++) punctuation (xs ++ [""]))
decorate _ _ = error "decorate.punctuation length"

------------------------------------------------------------------------
-- Recursive Edges
------------------------------------------------------------------------
data RecursiveEdge b =
    RecursiveEdge
    {
    reSource    :: NodeName,
    reUplink    :: Int,
    reDownlink  :: NodeName,
    reSink      :: NodeName,
    reEdgeLabel :: b
    } deriving (Eq)

instance (Show b) => Show (RecursiveEdge b) where
    show (RecursiveEdge src up down sink label) =
        if up == 0
       then if null down
            then decorate ["<", ":", "-->", ">"]
                          [show label, show src, show sink]
            else decorate ["<", ",", "-->", ">"]
                          [show label, show src, show down, show sink]
       else if null down
             then decorate ["<", ",", "-->", ">"]
                          [show label, show src, show up, show sink]
            else decorate ["<", ",", ",", "-->", ">"]
                          [show label, show src, show up, show down,show sink]

------------------------------------------------------------------------
-- A recursive context has a list of predecessor edges (added one
-- at a time), a list of successor edges (added one at a time),
-- and a node (added only once)
```

```
-----------------------------------------------------------------------
data RecursiveContext a b =
    RecursiveContext
    {
    rcPreds  :: [RecursiveEdge b],
    rcNode   :: RecursiveNode a b,
    rcSuccs  :: [RecursiveEdge b]
    } deriving (Eq)

instance (Show a, Show b) => Show (RecursiveContext a b) where
    show (RecursiveContext preds node succs) =
      let outsuccs = concat (intersperse "," (map show succs))
          outpreds = concat (intersperse "," (map show preds))
      in if null preds
         then if null succs
              then decorate ["(-|", "|-)"] [show node]
              else "(-|" ++ show node ++ "|" ++ outsuccs ++ "-)"
         else if null succs
              then "(-" ++ outpreds ++ "|" ++ show node ++ "|-)"
              else "(-" ++ outpreds ++ "|" ++ show node ++
                   "|" ++ outsuccs ++ "-)"

-- Get the list of edges in the context
contextEdges :: RecursiveContext a b -> -- Context to listify
                [RecursiveEdge b]       -- Resulting list of edges
contextEdges (RecursiveContext preds node succs) = preds ++ succs

-----------------------------------------------------------------------
-- Finally, the recursive graph data type
-----------------------------------------------------------------------
data RecursiveGraph a b =
    EmptyRecursiveGraph |
    RecursiveGraph (RecursiveGraph a b) (RecursiveContext a b)
    deriving (Eq)

instance (Show a, Show b) => Show (RecursiveGraph a b) where
    show (EmptyRecursiveGraph) = "{G}"
    show (RecursiveGraph g cont) = show g ++ " &C " ++ show cont

-- A single step decomposition of a graph
type Decomp a b = (Maybe (RecursiveContext a b), RecursiveGraph a b)

-- An infix operator to extend a recursive graph
infixr &
c & g = RecursiveGraph g c

-----------------------------------------------------------------------
-- Operators on graphs, stolen and modified from Martin Erwig
-----------------------------------------------------------------------
-- Insert a node in its proper place in the hierarchy
insNode :: RecursiveNode a b ->  -- Node to add
           RecursiveGraph a b -> -- Graph to be augmented
          Maybe (RecursiveGraph a b) -- Augmented graph
insNode n g = insNodeName (nodeLabel n) (nodeName n) g

-- This is an insert of a list of top level nodes, per Martin Erwig
insNodes :: [RecursiveNode a b] -> -- List of nodes to add
            RecursiveGraph a b ->  -- Graph to be augmented
            Maybe (RecursiveGraph a b) -- Augmented graph
insNodes [] g = Just g
insNodes (n:ns) g =
    let mg = insNode n g
```

```
    in case mg of
         Nothing -> Nothing
        Just g -> insNodes ns g

-- This insert places a node identified by a node name in its
-- proper place in the graph
insNodeName :: a ->                  -- Label of new node
               NodeName ->           -- Node to insert
               RecursiveGraph a b -> -- Graph to be augmented
               Maybe (RecursiveGraph a b) -- Augmented graph (maybe)
insNodeName a nn EmptyRecursiveGraph =
    Just (RecursiveGraph EmptyRecursiveGraph
                         (RecursiveContext [] (SimpleNode nn a) []))
insNodeName a [] g = error "insNodeName.[]"
insNodeName a [n] g =
    Just (RecursiveGraph g (RecursiveContext [] (SimpleNode [n] a) []))
insNodeName a nn@(n:ns)
             rg@(RecursiveGraph g rc@(RecursiveContext preds node succs)) =
    case node of
      (SimpleNode nn' a') ->
         if (nn == nn')
        then error "insNodeName.conflict"
        else let mg' = insNodeName a nn g
             in case mg' of
                  Nothing -> Nothing
                 Just g' -> Just (RecursiveGraph g' rc)
      (RecursiveNode nn' subgraph a') ->
         if isPrefixOf nn' nn
        then let mg' = insNodeName a (drop (length nn') nn) subgraph
             in case mg' of
                  Nothing -> Nothing
                 Just g' ->
                   Just (RecursiveGraph
                             g
                             (RecursiveContext
                                preds
                                (RecursiveNode nn' g' a')
                                succs))
         else let mg' = insNodeName a nn g
              in case mg' of
                   Nothing -> Nothing
                  Just g' -> Just (RecursiveGraph g' rc)

insNodeNames :: (Show a, Show b) =>
                [(a, NodeName)] ->         -- List of new nodes
                RecursiveGraph a b ->      -- Graph to be augmented
                Maybe (RecursiveGraph a b) -- Augmented graph (maybe)
insNodeNames [] g = Just g
insNodeNames ((a, nn):anns) g =
    let mg' = insNodeName a nn g
    in case mg' of
         Nothing -> Nothing
        Just g' -> insNodeNames anns g'

-----------------------------------------------------------------------
-- Insert to edges in the recursive graph, given the sink of the
-- to edge
-----------------------------------------------------------------------
insEdgeTo :: (Show a, Show b) =>
             RecursiveEdge b ->       -- Edge to insert
             RecursiveGraph a b ->      -- Graph into which edge is inserted
             Maybe (RecursiveGraph a b) -- Resulting graph
```

```
insEdgeTo ret EmptyRecursiveGraph =
    error ("insEdgeTo.Empty: " ++ show ret ++ "\n")
insEdgeTo ret@(RecursiveEdge source up down sink b)
          rg@(RecursiveGraph g rc@(RecursiveContext preds node succs)) =
    let simplerec node nn = nodeName node == nn
       recrec node nn = isPrefixOf (nodeName node) nn
       newret = convertToDown ret
       simplemod g (RecursiveContext preds node succ) =
          RecursiveGraph g (RecursiveContext (newret:preds) node succs)
        recmod g newsubgraph (RecursiveContext preds node succ) =
         let newnode = RecursiveNode
                         (nodeName node) newsubgraph (nodeLabel node)
          in RecursiveGraph g (RecursiveContext preds newnode succs)
    in match rg sink sink simplerec simplemod recrec recmod

insEdgesTo :: (Show a, Show b) =>
              [RecursiveEdge b] ->   -- Edges to insert
              RecursiveGraph a b ->  -- Graph into which edges are inserted
              Maybe (RecursiveGraph a b) -- Resulting graph
insEdgesTo edges g = foldrMaybe insEdgeTo g edges

-----------------------------------------------------------------------
-- Insert from edges in the recursive graph, given the sink of the
-- from edge
-----------------------------------------------------------------------
insEdgeFrom :: (Show a, Show b) =>
               RecursiveEdge b ->         -- Edge to insert
               RecursiveGraph a b ->      -- Graph into which edge is inserted
               Maybe (RecursiveGraph a b) -- Resulting graph
insEdgeFrom ref EmptyRecursiveGraph =
    error ("insEdgeFrom.Empty: " ++ show ref ++ "\n")
insEdgeFrom ref@(RecursiveEdge source up down sink b)
            rg@(RecursiveGraph g rc@(RecursiveContext preds node succs)) =
    let simplerec node nn = nodeName node == nn
       recrec node nn = isPrefixOf (nodeName node) nn
        newref = convertFromDown ref
       simplemod g (RecursiveContext preds node succ) =
          RecursiveGraph g (RecursiveContext preds node (newref:succs))
        recmod g newsubgraph (RecursiveContext preds node succ) =
         let newnode = RecursiveNode
                         (nodeName node) newsubgraph (nodeLabel node)
          in RecursiveGraph g (RecursiveContext preds newnode succs)
    in match rg source source simplerec simplemod recrec recmod
--                          ((RecursiveEdgeFrom up down sink b):succs)))

insEdgesFrom :: (Show a, Show b) =>
                [RecursiveEdge b] ->   -- Edges to insert
                RecursiveGraph a b ->  -- Graph into which edges are inserted
                Maybe (RecursiveGraph a b) -- Resulting graph
insEdgesFrom edges g = foldrMaybe insEdgeFrom g edges

-----------------------------------------------------------------------
-- Insert an entire context into its proper place in the graph
-- (not just an append)
-----------------------------------------------------------------------
insContext :: (Show a, Show b) =>
              RecursiveContext a b -> -- Context to insert
              RecursiveGraph a b ->   -- Graph to be augmented
             Maybe (RecursiveGraph a b) -- Resulting graph
insContext (RecursiveContext preds node succs) g =
    let mg1 = insNode node g
    in case mg1 of
```

```
         Nothing -> Nothing
        Just g1 ->
          let mg2 = insEdgesTo preds g1
          in case mg2 of
               Nothing -> Nothing
              Just g2 -> insEdgesFrom succs g2

-----------------------------------------------------------------------
-- Graph matching, and other decomposition operators
-----------------------------------------------------------------------
-- Decompose a graph, taking out the context that introcudes the node
-- with the specified name and label, be it a simple or a recursive node
matchNodeName :: NodeName ->           -- node name to look for
                 RecursiveGraph a b -> -- Graph to search
                 Decomp a b            -- Decomposition of the graph
matchNodeName nn EmptyRecursiveGraph = (Nothing, EmptyRecursiveGraph)
matchNodeName nn rg@(RecursiveGraph g rc@(RecursiveContext preds node succs)) =
    if nodeName node == nn
    then (Just rc, g)
    else case node of
           SimpleNode nn' a ->
            let (mcont, g') = matchNodeName nn g
            in case mcont of
                 Nothing -> (Nothing, rg)
                Just cont -> (Just cont, RecursiveGraph g' rc)
          RecursiveNode nn' subgraph a ->
            if null (tail nn)
            then (Nothing, rg)
            else let (mcont', g') = matchNodeName (tail nn) subgraph
                 in case mcont' of
                     Nothing ->
                       let (mcont'', g'') = matchNodeName nn g
                       in case mcont'' of
                            Nothing -> (Nothing, rg)
                            Just cont'' ->
                               (Just cont'', RecursiveGraph g'' rc)
                     Just cont' ->
                       (Just cont',
                        RecursiveGraph g (RecursiveContext
                                            preds
                                          (RecursiveNode nn' g' a)
                                          succs))

-- Define an operator that will identify a context (possibly
-- deeply buried within the graph, and mdofify it in place,
-- according to the modifier specified.
match :: RecursiveGraph a b ->  -- Graph to search and modify
         NodeName ->            -- Node name to look for
        NodeName ->           -- Node name preserved during recursion
        -- Simple node recognizer
        (RecursiveNode a b ->
         NodeName ->
         Bool) ->
        -- Simple node modifier:
        (RecursiveGraph a b ->
         RecursiveContext a b ->
         RecursiveGraph a b) ->
        -- Recursive node recognizer
        (RecursiveNode a b ->
         NodeName ->
         Bool) ->
        -- Recursive node modifier:
```

```
        (RecursiveGraph a b ->
         RecursiveGraph a b ->
         RecursiveContext a b ->
         RecursiveGraph a b) ->
        Maybe (RecursiveGraph a b)
match EmptyRecursiveGraph nnlocal nnglobal simplerec simplemod recrec recmod =
    error "match.empty"
match rg@(RecursiveGraph g rc@(RecursiveContext preds node succs))
      nnlocal nnglobal simplerec simplemod recrec recmod =
    case node of
      (SimpleNode nn a) ->
         if simplerec node nnlocal
        then Just (simplemod g rc)
         else let mg = match g nnlocal
                           nnglobal simplerec
                         simplemod recrec recmod
              in case mg of
                   Nothing -> Nothing
                   Just g -> Just (RecursiveGraph g rc)
      (RecursiveNode nn subgraph a) ->
        if recrec node nnlocal
        then let mnewsubgraph =
                    match subgraph (drop (length nn) nnlocal) nnglobal
                         simplerec simplemod recrec recmod
             in case mnewsubgraph of
                  Nothing -> Nothing
                Just newsubgraph -> Just (recmod g newsubgraph rc)
         else let mg = match g nnlocal
                           nnglobal simplerec
                         simplemod recrec recmod
              in case mg of
                   Nothing -> Nothing
                 Just g -> Just (RecursiveGraph g rc)

-----------------------------------------------------------------------
-- Analyzers for recursive graphs
-----------------------------------------------------------------------

findEdge :: RecursiveGraph a b -> -- Graph to search
            NodeName ->           -- Source of edge to search for
           NodeName ->           -- Sink of edge to search for
           Bool
findEdge g source sink =
    let (mcont, g) = matchNodeName source g
       findSink :: [RecursiveEdge b] -> Bool
       findSink [] = False
       findSink (e:es) = (reSink e == sink) || (findSink es)
       findSource :: [RecursiveEdge b] -> Bool
       findSource [] = False
       findSource (e:es) = (reSource e == source) || (findSource es)
    in case mcont of
         Nothing -> False
        Just cont ->
          let (mcont', g') = matchNodeName sink g
           in case mcont' of
               Nothing -> False
              Just cont' -> findSink (rcSuccs cont) ||
                            findSource (rcPreds cont')

-----------------------------------------------------------------------
-- Graph algorithms specifically for a recursive graph
-----------------------------------------------------------------------
```

```
-- Produce the list of top level nodes (simple or recursive)
-- that make up a graph
listRecursiveNodes :: RecursiveGraph a b -> -- Recursive graph to listify
                      [RecursiveNode a b]   -- list of nodes in the graph
listRecursiveNodes EmptyRecursiveGraph = []
listRecursiveNodes (RecursiveGraph g cont) =
    (rcNode cont):(listRecursiveNodes g)

-- Produce the list of top level edges in the graph
listRecursiveEdges :: RecursiveGraph a b -> -- Recursive graph to listify
                      [RecursiveEdge b]     -- list of edges in the graph
listRecursiveEdges EmptyRecursiveGraph = []
listRecursiveEdges
    (RecursiveGraph g cont@(RecursiveContext preds node succs)) =
    listRecursiveEdges g ++
      case node of
        (SimpleNode nn a) -> contextEdges cont
        (RecursiveNode nn sg a) -> contextEdges cont ++ (listRecursiveEdges sg)

-- Flatten a graph into a single level graph, but with the
-- recursive information stored up in the node names and edge names
flatten :: NodeName ->           -- Node name at next higher level
           RecursiveGraph a b -> -- Graph to flatten
          RecursiveGraph a b    -- Flattened graph
flatten upnn EmptyRecursiveGraph = EmptyRecursiveGraph
flatten upnn (RecursiveGraph g (RecursiveContext preds node succs)) =
    let newpreds = map (convertToUp upnn) preds
       newsuccs = map (convertFromUp upnn) succs
       newnode  = nodeUp node upnn
    in case node of
         (SimpleNode nn a) ->
            RecursiveGraph (flatten upnn g)
                           (RecursiveContext newpreds newnode newsuccs)
         (RecursiveNode nn subgraph a) ->
            -- A crucial property is that the empty recursive node
            -- should be inserted before any of its subnodes or edges
           let emptynode = (RecursiveContext
                                newpreds
                                (RecursiveNode nn EmptyRecursiveGraph a)
                                newsuccs)
            in merge (RecursiveGraph
                       (flatten upnn g)
                       emptynode)
                     (flatten nn subgraph)

-- Deepen a flattened graph, restoring its recursive structure.
-- Assume that nodes are sorted, with prefix always preceding a
-- node with a name that is an extension of the prefix
-- NEED flatten to establish this, or need function to sort the
-- flattened graph
deepen :: (Show a, Show b) =>
          RecursiveGraph a b ->      -- Graph to deepend
          Maybe (RecursiveGraph a b) -- Resulting deepened graph
deepen EmptyRecursiveGraph = Just EmptyRecursiveGraph
deepen (RecursiveGraph g rc@(RecursiveContext preds node succs)) =
    let mdeeper = deepen g
    in case mdeeper of
         Nothing -> Nothing
        Just deeper ->
           case node of
             (SimpleNode nn a) ->
                insContext (RecursiveContext preds node succs) deeper
```

```
             (RecursiveNode nn subgraph a) ->
               let msg = deepen subgraph
              in case msg of
                   Nothing -> Nothing
                   Just sg ->
                         Just (RecursiveGraph
                                 deeper
                                 (RecursiveContext
                                    preds
                                    (RecursiveNode nn sg a)
                                    succs))

-- Convert a node with ups and downs in a recursive graph to a node
-- with complex source and sink but no ups and downs for a flattened
-- graph
convertToDown :: RecursiveEdge b -> -- Edge to convert
                 RecursiveEdge b    -- Converted list of edges
convertToDown ret@(RecursiveEdge source ups downs sink b) =
    let cp = commonPrefix source sink
       ups = length source - length cp - 1
    in if ups == 0
       then if null cp
            then RecursiveEdge [last source] 0 (init sink) [last sink] b
           else RecursiveEdge [last source] 0 [] [last sink] b
       else RecursiveEdge source ups (cp ++ (init sink)) [last sink] b

convertFromDown:: RecursiveEdge b -> -- Edge to convert
                  RecursiveEdge b    -- Converted list of edges
convertFromDown ref@(RecursiveEdge source ups downs sink b) =
    let cp = commonPrefix source sink
       ups = length source - length cp - 1
    in if ups == 0
       then if null cp
            then RecursiveEdge [last source] 0 (init sink) [last sink] b
           else RecursiveEdge [last source] 0 [] [last sink] b
       else RecursiveEdge [last source] ups (init sink) [last sink] b

-- Convert a node with ups and downs in a recursive graph to a node
-- with complex source and sink but no ups and downs for a flattened
-- graph
convertToUp :: NodeName ->        -- Context in which to convert up
               RecursiveEdge b -> -- Edge to convert
               RecursiveEdge b    -- Converted list of edges
convertToUp nn (RecursiveEdge source ups downs sink b) =
    if isLocal source
    then if null downs
         then RecursiveEdge (nn ++ source) 0 [] (nn ++ sink) b
        else RecursiveEdge source 0 [] (downs ++ sink) b
    else RecursiveEdge source 0 [] (downs ++ sink) b

convertFromUp :: NodeName ->        -- Context in which to convert up
                 RecursiveEdge b -> -- Edge to convert
                 RecursiveEdge b    -- Converted list of edges
convertFromUp nn (RecursiveEdge source up downs sink b) =
    if isLocal sink
    then if null downs
         then RecursiveEdge (nn ++ source) 0 [] (nn ++ sink) b
        else RecursiveEdge source 0 [] (downs ++ sink) b
    else RecursiveEdge (nn ++ source) 0 [] sink b

-- Merge two graphs, assuming that sll the nodes of one can be
-- merged into the other.
```

```
merge :: RecursiveGraph a b -> -- First graph to merge
         RecursiveGraph a b -> -- Second graph to merge
        RecursiveGraph a b    -- The merged graph
merge g EmptyRecursiveGraph = g
merge g (RecursiveGraph h c) = RecursiveGraph (merge g h) c

-----------------------------------------------------------------------
-- maps and folds
-----------------------------------------------------------------------

mapGraph :: (a -> a) ->
            RecursiveGraph a b ->
           RecursiveGraph a b
mapGraph f EmptyRecursiveGraph = EmptyRecursiveGraph
mapGraph f (RecursiveGraph g (RecursiveContext preds node succs)) =
    case node of
      (SimpleNode nn a) ->
        RecursiveGraph
         (mapGraph f g)
         (RecursiveContext preds (SimpleNode nn (f a)) succs)
      (RecursiveNode nn sg a) ->
        RecursiveGraph
         (mapGraph f g)
         (RecursiveContext
             preds
             (RecursiveNode nn (mapGraph f sg) (f a))
             succs)
```

## 7.7 GraphFlat.hs

```
module GraphFlat
    (
    module GraphInductive,
    FlatGraph (..),
    flat2Recursive
    ) where

import GraphInductive

-----------------------------------------------------------------------
-- Some utilities that make it easier to test the recursive graph
-- data structure. In particular, it is nice to input and output
-- the graph in a traditional list of vertices and edges format.
-----------------------------------------------------------------------

data FlatGraph a b =
    FlatGraph
    {
    fgNodes :: [(NodeName, a)],
    fgEdges :: [(NodeName, NodeName, b)]
    }

thd :: (a, b, c) -> c
thd (a, b, c) = c

instance (Show a, Show b) => Show (FlatGraph a b) where
    show (FlatGraph nodes edges) =
        let showedge :: (Show b) => b -> String -> String
          showedge b s = if null s
                         then show b
                       else show b ++ ", " ++ s
           shownode :: (Show b) => b -> String -> String
```

```
          shownode a s = if null s
                         then show a
                       else show a ++ ", " ++ s
       in "{ " ++ "[" ++ foldr shownode "" (map snd nodes) ++ "]" ++ " // " ++
                  "[" ++ foldr showedge "" (map thd edges) ++ "]" ++ " }"

flat2Recursive ::      -- Convert a flat graph to a recusive graph
    (Show a, Show b, Eq b) =>
    FlatGraph a b ->   -- The flat graph to convert
    RecursiveGraph a b -- The resulting recursive graph
flat2Recursive (FlatGraph nodes edges) =
    let maddnodes =
         insNodes (map (uncurry SimpleNode) nodes) EmptyRecursiveGraph
       makeEdgeTo :: (Show a, Show b, Eq b) =>
                     (NodeName, NodeName, b) ->
                     RecursiveGraph a b ->
                    Maybe (RecursiveGraph a b)
        makeEdgeTo (src, snk, b) g =
         insEdgeTo (RecursiveEdge src 0 [] snk b) g
    in case maddnodes of
         Nothing -> error "flat2Recursive.addnodes"
        Just addnodes ->
          case foldrMaybe makeEdgeTo addnodes edges of
            Nothing -> error "flat2Recursive.addeges"
            Just g -> g

foldrMaybe :: (a -> b -> Maybe b) -> b -> [a] -> Maybe b
foldrMaybe f z [] = Just z
foldrMaybe f z (a:as) =
    case foldrMaybe f z as of
      Nothing -> Nothing
      Just b -> f a b
```

## 7.8 GraphSystem.hs

```
module GraphSystem where

import GraphFlat
import List
import System
import Separation

-----------------------------------------------------------------------
--  System Instances
-----------------------------------------------------------------------
instance Containers (RecursiveGraph a b) (RecursiveNode a b) NodeName where
    select g nn =
      let (mcont, g') = matchNodeName nn g
      in fmap rcNode mcont
    -- The node name nn should be the same as the name of the node
    addElem g nn node =
      let mg' = insNode node g
      in case mg' of
           Nothing -> g
           Just g' -> g'
    deleteElem g nn =
      let (mcont, g') = matchNodeName nn g
      in case mcont of
           Nothing -> g
          Just cont -> g'
    emptyCont = EmptyRecursiveGraph
    elements = listRecursiveNodes
    idelements g = zip (map nodeName nodes) nodes
```

```
      where nodes = listRecursiveNodes g

instance (Component a) => Component (RecursiveNode a b) where
    stepComponent (SimpleNode nn a) = SimpleNode nn (stepComponent a)
    stepComponent (RecursiveNode nn sg a) =
        RecursiveNode nn sg (stepComponent a)
    stopComponent (SimpleNode nn a) = SimpleNode nn (stopComponent a)
    stopComponent (RecursiveNode nn sg a) =
        RecursiveNode nn sg (stopComponent a)
    startComponent () = SimpleNode nullNodeName (startComponent ())

instance (Component a) =>
         System (RecursiveGraph a b) (RecursiveNode a b) NodeName where
    startSystem = EmptyRecursiveGraph
    stepSystem nn g =
      let simplerec node nn = nodeName node == nn
          simplemod g (RecursiveContext preds node succs) =
            RecursiveGraph
              g
              (RecursiveContext
                preds
               (SimpleNode (nodeName node) (stepComponent (nodeLabel node)))
               succs)
          recrec node nn = isPrefixOf (nodeName node) nn
          recmod g sg (RecursiveContext preds node succs) =
            RecursiveGraph
              g
              (RecursiveContext
                preds
               (RecursiveNode
                 (nodeName node)
                 sg
                 (stepComponent (nodeLabel node)))
               succs)
          mg' = match g nn nn simplerec simplemod recrec recmod
      in case mg' of
           Nothing -> g
          Just g' -> g'
    stopSystem g = mapGraph stopComponent g

-----------------------------------------------------------------------
--  System Instances
-----------------------------------------------------------------------
instance (Component a) =>
         Separation (RecursiveGraph a b) (RecursiveNode a b) NodeName where
    interactionMatrix g nn1 nn2 =
      if findEdge g nn1 nn2
      then \g' nn1' nn2' -> True -- Can model communication protocol here
      else \g' nn1' nn2' -> False
```

## 7.9 KrenzSystem.hs

```
module KrenzSystem
  (
  KrenzSystem,
  KrenzFilterProperty
  ) where

import GraphInductive
import GraphSystem
import Container
import System
import Separation
```

```
data KrenzFilterProperty b =
    KrenzFilterProperty
    {
    kfpOther  :: b,   -- Other edge freight
    kfpFilter :: Prop -- Filter Property
    }

type KrenzSystem a b = RecursiveGraph a (KrenzFilterProperty b)

instance Containers (KrenzSystem a b)
                    (RecursiveNode a (KrenzFilterProperty b)) NodeName

instance (Component a) => Component (RecursiveNode a (KrenzFilterProperty b))

instance (Component a) =>
         System (KrenzSystem a b)
               (RecursiveNode a (KrenzFilterProperty b)) NodeName

instance (Component a) =>
         Separation (KrenzSystem a b)
                   (RecursiveNode a (KrenzFilterProperty b)) NodeName
```

## 7.10 KrenzAssuranceGraph.hs

```
module KrenzAssuranceSystem where

import KrenzSystem
import GraphInductive
import System
import Separation

-- The other freigth is also Prop
type KrenzAssuranceSystem a = KrenzSystem a Prop

instance Containers (KrenzAssuranceSystem a)
                    (RecursiveNode a (KrenzFilterProperty Prop)) NodeName

instance (Component a) =>
         Component (RecursiveNode a (KrenzFilterProperty Prop))

instance (Component a) =>
         System (KrenzAssuranceSystem a)
               (RecursiveNode a (KrenzFilterProperty Prop)) NodeName

instance (Component a) =>
         Separation (KrenzAssuranceSystem a)
                   (RecursiveNode a (KrenzFilterProperty Prop)) NodeName
```

## 7.11 Category.hs

```
module Category where

import Prelude hiding (product, Functor)
import Monad hiding (Functor)
import EdisonPrelude
import qualified Collection as C
import Maybe

-----------------------------------------------------------------
--  The categories defined here use the underlying set concept from
--  The edision "set" has constructors "empty" and "insert". This
--  means that the sets modelled are constructive sets, not general
--  sets. The category class below thus uses a constructive set of
```

```
--  objects and a constructive set of arrows. This permits
--  the definition of constructors for the categories, and puts
--  in the domain of "constructive category theory". No need anymore
--  for distinctions such as "locally small".
--
--  The definitions and properties are based on the thesis of
--  Yellamraju Venkata Srinivas, titled "Pattern Matching: A
--  Sheaf Theoretic Approach", published in 1991. I have also
--  used the references "Categories for the working Mathematician"
--  by Saunders MacLane, and "Categories" by T. S. Blyth, and
--  "Toposes, Triples, and Theories" by Michael Barr and
--  Charles Wells
-----------------------------------------------------------------

class (C.Set s o, C.Set s a) => Category c s o a where
    -- Analyzers
    dom          :: c s o a -> a -> o
    cod          :: c s o a -> a -> o
    catId        :: c s o a -> o -> a
    compose      :: c s o a -> a -> a -> Maybe a
    -- Constructors for a category
    emptyCat     :: c s o a
    mkCat        :: s o ->      -- Set of objects
                    s a ->      -- Set of arrows
                 (a -> o) -> -- Dom
                 (a -> o) -> -- Cod
                 c s o a
    -- Destructors for a category
    objects      :: c s o a -> s o
    arrows       :: c s o a -> s a
    -- build the hom sets right into the definition of a category
    morphisms    :: c s o a -> o -> o -> s a
    morphisms c src snk =
        C.filter (\a -> dom c a == src && cod c a == snk) (arrows c)

compose' :: (Category c s o a) => c s o a -> a -> a -> a
compose' c a1 a2 = fromJust (compose c a1 a2)
composable :: (Category c s o a) => c s o a -> a -> a -> Bool
composable c f g = dom c f == cod c g

-- Composition is associative
property Assoc c = All f g h.
    (compose' c f (compose' c g h)) === (compose' c (compose' c f g) h)
-- Two arrows are composable iff the compose function does not
-- return Nothing
property Composition c = All f g.
    lift (((compose c f g) /= Nothing) == composable c f g)
-- The identify laws, left and right
property Identity c = All f.
    compose' c (catId c (dom c f)) f === f /\
    compose' c f (catId c (dom c f)) === f
-- The domain and codomain of the identity arrow on an object are the
-- object itself
property IdArrow c = All o.
    dom c (catId c o) === o /\
    cod c (catId c o) === o

-----------------------------------------------------------------
-- The opposite of a category
-----------------------------------------------------------------

-- The domain and codomain functions have been interchanged, thus
```

```
-- the opposite of a category is the same category with all the
-- arrows reversed.
opposite :: (Category c s o a) => c s o a -> c s o a
opposite c =
  let os = objects c
      as = arrows c
      domop = dom c
      codop = cod c
  in mkCat os as codop domop

-----------------------------------------------------------------
-- Product of two categories
-----------------------------------------------------------------

instance (C.CollX s x, C.CollX s y) => C.CollX s (x,y)
instance (C.Coll s x, C.Coll s y) => C.Coll s (x,y)
instance (C.SetX s x, C.SetX s y) => C.SetX s (x,y)
instance (C.Set s x, C.Set s y) => C.Set s (x,y)

makepairs :: (C.Set s x, C.Set s y) => s x -> s y -> s (x, y)
makepairs sx sy = C.fromList [(x,y) | x <- C.toList sx, y <- C.toList sy]

product ::
  (Category c s o a, Category c s o' a', Category c s (o, o') (a, a')) =>
  c s o a -> c s o' a' -> c s (o, o') (a, a')
product c c' = mkCat pos pas pdom pcod
  where pos = makepairs (objects c) (objects c')
        pas = makepairs (arrows c) (arrows c')
        pdom (a, a') = (dom c a, dom c' a')
        pcod (a, a') = (cod c a, cod c' a')

-- The product of two Id arrows is an Id arrow
property ProductId c c' = All o o'.
    catId (product c c') (o, o') === (catId c o, catId c o')

-----------------------------------------------------------------
-- Functors
-----------------------------------------------------------------

class (Category c s o a, Category c s o' a') => Functor f c s o a o' a' where
    -- Destructors
    objectmap :: f c s o a o' a' -> o -> o'
    arrowmap  :: f c s o a o' a' -> a -> a'
    -- Constructor
    mkFunctor :: (o -> o') -> (a -> a') -> f c s o a o' a'

property FunctorId c f c' = All o a.
  catId c' (objectmap f o) === (arrowmap f (catId c o))
property FunctorArrow c f c' = All a.
  dom c (arrowmap f a) === objectmap f (dom c a)
property FunctorComposable c f c' =
    All h k. lift (composable c h k) ==>
             lift (composable c (arrowmap f h) (arrowmap f k))
property FunctorCompose c f c' = All h k.
    compose' c (arrowmap f h) (arrowmap f k) === arrowmap f (compose' c h k)

-----------------------------------------------------------------
-- The Hom Set functor
-----------------------------------------------------------------

data (C.Set s a) => HomSet s o a = MkHomSet
  {
```

```
  homdom :: o,
  homcod :: o,
  homset :: s a
  } deriving (Eq)

instance (Eq (HomSet s o a), C.Set s a) => C.CollX s (HomSet s o a)
instance (Eq (HomSet s o a), C.Set s a) => C.Coll s (HomSet s o a)
instance (Eq (HomSet s o a), C.Set s a) => C.SetX s (HomSet s o a)
instance (Eq (HomSet s o a), C.Set s a) => C.Set s (HomSet s o a)
instance (Eq (HomArrow s o a), C.Set s a) => C.CollX s (HomArrow s o a)
instance (Eq (HomArrow s o a), C.Set s a) => C.Coll s (HomArrow s o a)
instance (Eq (HomArrow s o a), C.Set s a) => C.SetX s (HomArrow s o a)
instance (Eq (HomArrow s o a), C.Set s a) => C.Set s (HomArrow s o a)

-----------------------------------------------------------------
-- The hom category, a subcategory of Set, produced from the
-- objects and arrows of another category.
-----------------------------------------------------------------

data (C.Set s a) => HomArrow s o a = MkHomArrow
  {
  homardom :: HomSet s o a,
  homarcod :: HomSet s o a,
  homarrul :: s a -> s a
  }

homsetof :: (Category c s o a) => c s o a -> o -> o -> HomSet s o a
homsetof c src snk = MkHomSet src snk (morphisms c src snk)

homarrowof :: (Category c s o a) => c s o a -> (a, a) -> HomArrow s o a
homarrowof c (f, g) =
  MkHomArrow (homsetof c (dom c f) (dom c g))
             (homsetof c (cod c f) (cod c g))
            (\x -> (C.fromList
                     (map (\e -> compose' c f (compose' c e g))
                    (C.toList x))))

homarid :: (Category c s o a) => c s o a -> HomSet s o a -> HomArrow s o a
homarid c homset = MkHomArrow homset homset id

homcompose :: (Eq (s a),Category c s o a) =>
              c s o a -> HomArrow s o a -> HomArrow s o a ->
             Maybe (HomArrow s o a)
homcompose c (MkHomArrow src1 snk1 r1) (MkHomArrow src2 snk2 r2) =
  if (snk1 == src2)
  then Just (MkHomArrow src1 snk2 (r2 . r1))
  else Nothing

homop :: (C.Set s a) => HomArrow s o a -> HomArrow s o a
homop (MkHomArrow src snk r) = MkHomArrow snk src r

instance (Eq (HomSet s o a), Eq (HomArrow s o a), Category c s o a) =>
         Category c s (HomSet s o a) (HomArrow s o a)

-- Produce the category of hom sets and arrows
sets :: (Eq (s a), Eq (HomArrow s o a), Category c s o a) =>
        c s o a -> c s (HomSet s o a) (HomArrow s o a)
sets c = let
             pos = makepairs (objects c) (objects c)
             pas = makepairs (arrows c) (arrows c)
             setobs = C.fromList (map (genhomset c) (C.toList pos))
            genhomset :: (Category c s o a) =>
```

```
                    c s o a -> (o, o) -> HomSet s o a
            genhomset c (o1, o2) = MkHomSet o1 o2 (morphisms c o1 o2)
            setars = C.fromList (map (genhomarr c) (C.toList pas))
            genhomarr :: (Category c s o a) =>
                    c s o a -> (a, a) -> HomArrow s o a
            genhomarr c (f, g) =
              MkHomArrow (genhomset c (dom c f, dom c g))
                         (genhomset c (cod c f, cod c g))
                      (\x -> C.fromList
                               (map (\a -> (compose' c f (compose' c a g)))
                              (C.toList x)))
         in mkCat setobs setars homardom homarcod

-----------------------------------------------------------------
-- The hom set functor.
-----------------------------------------------------------------

homsetfunctor ::
  (Category c s o a, Category c s (o, o) (a, a),
   Functor f c s (o, o) (a, a) (HomSet s o a) (HomArrow s o a)) =>
  c s o a -> f c s (o, o) (a, a) (HomSet s o a) (HomArrow s o a)
homsetfunctor c =
  let cop = opposite c
      copxc = product cop c
  in mkFunctor (uncurry (homsetof c)) (homarrowof c)

-----------------------------------------------------------------
-- The contravariant Hom Functor, which is the hom set functor on
-- the first argument only. The second argument is fixed at some
-- object of the category
-----------------------------------------------------------------

contravarianthomobjs :: (Category c s o a) => c s o a -> o -> o -> HomSet s o a
contravarianthomobjs c o' o = MkHomSet o o' (morphisms c o o')

contravarianthomarrows :: (Category c s o a) =>
                          c s o a -> o -> a -> HomArrow s o a
contravarianthomarrows c o' a = (curry (homarrowof c)) (catId c o') a

contravarianthom ::
  (Category c s o a, Category c s (o, o) (a, a),
   Functor f c s o a (HomSet s o a) (HomArrow s o a)) =>
  c s o a -> o -> f c s o a (HomSet s o a) (HomArrow s o a)
contravarianthom c o' =
  let cop = opposite c
  in mkFunctor (contravarianthomobjs cop o') (contravarianthomarrows cop o')

-----------------------------------------------------------------
-- Seives.
-- A sieve is a collection of arrows, with common
-- codomain, closed under right composition.
-----------------------------------------------------------------

-- Given a category, and an object (the common codomain), make a sieve
-- out of a set of arrows (the set is currently represented by a list)
class (Category c s o a) => Sieve c s o a where
  isSieve :: c s o a -> s a -> o -> Bool
  allSieves :: c s o a -> o -> s (s a)

-- Now specify the Sieve properties using the Programatica properties
property CommonCodomain c e = All f g o.
  lift (isSieve c e o) ==> (lift (C.member e f) /\ lift (C.member e f)) ==>
```

```
  (cod c f === cod c g)
property RightComposition c e = All f g o.
  lift (isSieve c e o) /\ lift (C.member e f) /\ lift (composable c f g) ==>
  lift (C.member e (compose' c f g))
-- Any sieve on an object is an element of (Sieves o a)
property AllSieves c = All o as.
  lift (isSieve c as o) ==> lift (C.member (allSieves c o) as)

-----------------------------------------------------------------
-- Sieve as a subfunctor of the contravariant Hom Functor
-- "Subfunctor" is a bit of a misnomer here. As far as I can tell,
-- there is no formal concept of subfunctor. What is meant is
-- that the value of the sieve functor, at any object o, is a
-- subset of the value of the contravariant hom functor on o
-----------------------------------------------------------------

sievehomobjs :: (Sieve c s o a) => c s o a -> s a -> o -> o -> HomSet s o a
sievehomobjs c s o' o = MkHomSet o o' (C.intersect (morphisms c o o') s)

sievehomarrows :: (Category c s o a) => c s o a -> o -> a -> HomArrow s o a
sievehomarrows = contravarianthomarrows

sievefunctor ::
  (Sieve c s o a, Functor f c s o a (HomSet s o a) (HomArrow s o a)) =>
  c s o a -> s a -> o -> f c s o a (HomSet s o a) (HomArrow s o a)
sievefunctor c s o =
  let cop = opposite c
  in mkFunctor (sievehomobjs cop s o) (sievehomarrows cop o)

-----------------------------------------------------------------
-- Grothendieck topology
-- The properties here are somewhat difficult, so it is necessary
-- to refer to the thesis of Srinivas (or to some other reference
-- on sheaf theory, such as MacLane and Moerdick)
-----------------------------------------------------------------

class (Sieve c s o a) => GrothendieckTopology c s o a where
  j :: c s o a -> o -> s a

property JYieldsSieves c j = All o. lift (isSieve c (j c o) o)

-- The maximal sieve contains all arrows with codomain o
maximalSieve :: (Category c s o a) => c s o a -> o -> s a
maximalSieve c o = C.filter (\f -> cod c f == o) (arrows c)

-- The maximal sieve is a cover
property IdentityCover j = All gt o.
  lift (C.member (j gt o) (maximalSieve gt o))

-- Stability of covers under a change of base
property StabilityUnderChangeOfBase j gt = All r o.
  lift (C.member (j gt o) r) ==>
  (All f. lift (C.member (j gt o) ((star gt f) r)))

star :: (GrothendieckTopology c s o a) => c s o a -> a -> s a -> s a
star c f r | isSieve c r (cod c f) =
  C.filter (\g -> C.member r (compose' c f g)) (maximalSieve c (dom c f))

-- Stability of covers under refinement
property StabilityUnderRefinement gt j = All r s o f.
  lift (C.member (j gt o) r) /\
  lift (not (null (allSieves gt o))) /\
```

```
  (All f s. lift (C.member r f)) /\
            lift (C.member (allSieves gt o) s) ==>
            lift (C.member (j gt o) ((star gt f) s))
```

## 7.12 GraphCategory.hs

```
module GraphCategory
    (
    RecursiveGraphHomomorphism (..)
    )
    where

import Prelude hiding (product, Functor)
import GraphInductive
import Maybe2
import Category
import EdisonPrelude
import qualified Collection as C

--------------------------------------------------------------------------
-- Define a recursive graph homomorphism
--------------------------------------------------------------------------

-- The homomorphism has a source, a sink, and a function from the
-- source to the sink
data RecursiveGraphHomomorphism a b =
    RecursiveGraphHomomorphism
    {
    -- The source and sink of the homomorphism
    rghSource :: RecursiveGraph a b,
    rghSink   :: RecursiveGraph a b,
    -- The node and edges in the sink are (possibly) mapped to
    -- the nodes and edges in the source
    rghnodemap :: RecursiveNode a b -> Maybe (RecursiveNode a b),
    rghedgemap :: RecursiveEdge b -> Maybe (RecursiveEdge b)
    }

-- Compose two recursive graph homomorphisms
rghcompose :: (Eq a, Eq b) =>
              RecursiveGraphHomomorphism a b ->
              RecursiveGraphHomomorphism a b ->
              Maybe (RecursiveGraphHomomorphism a b)
rghcompose f g =
     if rghSink g == rghSource g
     then Just (RecursiveGraphHomomorphism
                (rghSource g)
                (rghSink g)
                (composeMaybe (rghnodemap f) (rghnodemap g))
                (composeMaybe (rghedgemap f) (rghedgemap g)))
     else Nothing

-- The identify morphism is a function that leaves the recursive
-- graph unchanged. Each node is mapped to the singleton list of nodes
-- having that one node as a member, and similarly for edges.
rghid :: RecursiveGraph a b -> RecursiveGraphHomomorphism a b
rghid rg =
    RecursiveGraphHomomorphism
      rg
      rg
      (\n -> Just n)
      (\e -> Just e)

--------------------------------------------------------------------------
```

```
-- Now make a class for graph category
-----------------------------------------------------------------------

-- First, an abstract data type to serve as an instance. Note that the
-- parameters "o" for objects, and "a" for arrows are ignored in this
-- declaration.
data (C.Set s (RecursiveGraph n e),
      C.Set s (RecursiveGraphHomomorphism n e)) => GraphCategory n e s o a =
    GraphCategory
    {
    gcObjects :: s (RecursiveGraph n e),
    gcArrows  :: s (RecursiveGraphHomomorphism n e)
    }

-- Next, a class (API) like interface for the graph category
class (C.Set s o, C.Set s a) =>
     GraphCategoryC c n e s o a | n e -> o, o -> a where
    gccObjects        :: c n e s o a -> s o
    gccArrows         :: c n e s o a -> s a
    gccEmptyGraph     :: c n e s o a -> o
    gccRecursiveGraph :: c n e s o a -> o -> RecursiveContext a b -> o
    gccEmptyCat       :: c n e s o a
    gccMkGraphCat     :: s o -> s a -> c n e s o a

-- The abstract data type is an instance of the API class
instance (C.Set s (RecursiveGraph n e),
          C.Set s (RecursiveGraphHomomorphism n e)) =>
         GraphCategoryC
           GraphCategory
             n e s
             (RecursiveGraph n e)
             (RecursiveGraphHomomorphism n e)

-- If it is an instance of the graph category, then a category can be
-- made out of it.
instance (Eq n, Eq e,
          C.Set s (RecursiveGraph n e),
          C.Set s (RecursiveGraphHomomorphism n e),
         GraphCategoryC c n e s o a) =>
     Category (c n e) s (RecursiveGraph n e) (RecursiveGraphHomomorphism n e)
    where
     dom c rgh     = rghSource rgh
     cod c rgh     = rghSink rgh
     catId c rg    = rghid rg
     compose c f g = rghcompose f g
     objects c     = gccObjects c
     arrows c      = gccArrows c
     emptyCat      = gccEmptyCat
     mkCat os as dom cod = gccMkGraphCat os as
```

# 8 Hol Source

This section contains the Hol source code for the current version of the Haskell specifications that have been translated to Hol. So far, only the inductive definition of the recursive graph have been translated into Hol, however, this is the fundamental data structure of the specification.

## 8.1 GraphInductive.sml

```
(* HOL Source for GraphInductive *)
```

```
load "bossLib";
open bossLib;
load "integerTheory";
open integerTheory;
load "listTheory";
open listTheory;
load "stringTheory";
open stringTheory;

(* This one is good *)
val x = Hol_datatype
    `RecursiveNode = SimpleNode of int list => 'a |
                     RecursiveNode of int list => RecursiveGraph => 'a;
     RecursiveEdge =
       <|source:   int list;
         uplink:   int list;
         downlink: int list;
         sink:     int list;
         edgeLabel: 'b
        |>;
     RecursiveAdjacency = RecursiveAdjacency of RecursiveEdge list;
     RecursiveContext =
       <|preds:   RecursiveEdge list;
         newnode: RecursiveNode;
         succs:   RecursiveEdge list
        |>;
     RecursiveGraph = EmptyRecursiveGraph |
                      RecursiveGraph of RecursiveGraph => RecursiveContext;
     Decomp =
       <|flag: bool;
         component: RecursiveContext;
         subgraph:  RecursiveGraph
        |>`
     handle e => Raise e;

TypeBase.axiom_of (valOf (TypeBase.read "RecursiveGraph")) handle e => Raise e;
map type_of [``RecursiveGraph``,
             ``RecursiveContext``,
             ``RecursiveAdjacency``,
             ``RecursiveEdge``,
             ``RecursiveNode``,
             ``Decomp``
            ];

``x: (bool, int) RecursiveNode`` handle e => Raise e;
``RecursiveNode [1] EmptyRecursiveGraph "label"`` handle e => Raise e;
type_of it;

Define `insNode n g =
           RecursiveGraph g
                         <|preds:=[];
                           newnode:=n;
                           succs:=[]|>`;
type_of ``insNode``;
```

# 9 To do list

The following is a list of tasks to complete on the specifications presented in this section.

- **Completed by version 3.0:** ~~**Dynamic graphs:** The ability to add and delete nodes from the recursive graph. This will be a very straightforward extension of~~

~~the current recursive graph class. Once added to the recursive graph class, the new capabilities should carry through to all the instances, such as Krenz system.~~

- **Completed by version 2.0:** ~~**Grothendieck topology on the graph:** A topological view of the system, permitting the formulation (and automated answering) of what if questions. For example: Is the system still an instance of the specified Krenz information flow policy when a particular node (and edges to and from the node) are added?~~
- **Completed by version 3.0:** ~~**Testing:** The functions in the specification will be tested (and corrected). In particular, the flatten and deepen functions will be tested.~~
- **Added version 1.0:** Node freight of different types: Add the ability to have node freight (and node properties) of different types in different subgraphs of the recursive graph. Version 3 update: This appears to be a straightforward exercise in the use of existential types in Haskell. During earlier versions of the report, this looked like a more difficult task.
- **Added version 1.0:** Edge freight of different types: Add the ability to have edge freight (and edge properties) of different types in different subgraphs of the recursive graph. Version 3 update: This appears to be a straightforward exercise in the use of existential types in Haskell. During earlier versions of the report, this looked like a more difficult task.
- **Added version 1.0:** Graphical output: Add the ability to make a graphical output of a Krenz system or a Krenz assurance system. This will be done if a suitable graphics package can be interfaced to Haskell. This item is of low priority. Version 3 update: There is a nice graph drawing package called Da Vinci, which has an interface to Haskell. This will probably be used for the Krenz system.
- **Added version 3.0:** It should be possible to define a category as a graph with additional properties. This will be investigated, and done if possible. This will result in an inductive definition of a category. This may be a result worth publishing.
- **Added version 3.0:** Having seen a simpler definition, it may be possible to do another definition of recursive graph that is not inductive, but still simple, having clear constructors and destructors, but permitting more a elegant and efficient definition of the flatten and deepen primitives.

# 10 References

1. *FGL / Haskell – A Functional Graph Library (User Guide)*, Martin Erwig, http://www.cs.orst.edu/~erwig/papers/abstracts.html#AMAST98
2. *Sheaves in Geometry and Logic*, Saunders Mac Lane and Ieke Moerdijk, Springer Verlag, Berlin, 1992.
3. *Elementary Categories, Elementary Toposes*, Colin McLarty, Oxford Science Publications, 1995.
4. *Categories for Types*, Roy L. Crole, Cambridge University Press, 1993.

5. *Categories,* T. S. Blyth, Longman Group Limited, 1986.
6. *Categories for the Working Mathematician,* Saunders Mac Lane, Springer Verlag, New York, 1971.
7. *The Haskell School of Expression, Learning Functional Programming Through Multimedia,* Paul Hudak, Cambridge University Press, 2000.
8. *Pattern Matching: A Sheaf-Theoretic Approach,* Yallamraju Venkata Srinivas, Phd Thesis, 1991
9. *Top-down Synthesis of Divide and Conquer Algorithms,* Doug R. Smith, *Artificial Intelligence 27* (1985), pages 43-96.
10. *Toposes, Triples, and Theories,* M. Barr and C. Wells, Number 278 in Comprehensive studies in Mathematics, Springer-Verlag, 1985

# 11 Acronyms

ADT..... Abstract Data Type
CC..... Common Criteria
FGL..... Functional Graph Library
HOL ..... Higher Order Logic
ID ..... Identifier
I/O ..... Input / Output
KAG ..... Krenz Assurance Graph
KMP ..... Knuth Morris Pratt algorithm
MASK..... Mathematically Analyzed Separation Kernel
NT ..... New Technology
PhD..... Philosophy Doctorate