Calculate with Concepts – An Introduction

This paper introduces the modelling technique dubbed 'Calculate with Concepts' (CC). This technique was developed in the area of process architecture, but can be used in other areas as well, such as in software, business, service, or security architecture. The technique helps in producing a model, including formal specifications (restrictions) for architectures of any kind.

1 Introduction

So what are the issues in modelling? To answer this, let us look at modelling from a distance. First, you have an idea about what to model. The idea is usually pretty simple, so it is probably going to not so hard can to make a model for that – say a week.

Next, you go to a room with whiteboards, and you start making the model. You are gazing at the impeccable white of the board, and wonder where you will start, and how you will continue. You are perhaps familiar with OO methodologies, but they are so huge – and you won't get it done in – say a week.

Nevertheless, you persist in your search for the right model, and you may find it. You know it's the right model – your feel it, you stomach tells you. And it *is* simple and easy, as you had expected. You can explain it to others, and all of you can see how it is relevant, and useful.

If you have been lucky, one week (or less) has passed. If you're not so lucky – and you are not to blame, after all, it is a search and you can't be sure when you find what you're looking for. Still, people whom you showed this simple model might ask why it took you so long to come up with this simple thing…

So how does CC help here? For one thing, CC is not the magic wand that finds the model you're looking for. Rather, it helps you on your journey by keeping you focussed so that you don't stray too far onto the wrong paths. Here are some major features that help accomplish this:

- CC is a technique that helps you produce formal models. This means that they have a mathematical basis, i.e. set theory and logic.
- the problem owner is not forced to talk in another language (as is done in OO-modelling) than what he is used to. There will always be a one to one relation between the natural language sentences used by the problem owner, and the mathematical underpinning of that model.

So basically, the modeller maps natural language sentences onto a mathematical model. Within this model, set theory and logic help the modeller to reason with this model, make it more precise, consistent, etc. The results are translated back to natural language, where the problem owner is invited to check the results, and provide feedback in case the model can be enhanced.

Whenever the model needs to take multiple problem owners into account, there is the problem that such problem owners don't necessarily have the same problems, yet refer to the same system (or business). If the problems cannot be addressed from a single point of view (e.g. there are problems with finance handling, the ICT infrastructure, etc.), then the CC technique allows a sub-model (discussion) to be made for each frame of reference. Then, the sub-models are 'glued' onto one another, and the interaction between these models is presented to the different problem owners in his/her own problem specific language so that they can provide feedback to enhance the sub-models.

In a time where the trend is to make large systems larger and larger, and where organisations are focussing more and more on specific services of such systems, the need arises to oversee all that. Businesses must now oversee systems and services of which parts are outsourced. This cannot be done easily with traditional ways (such as in an OO model), as issues like communication, performance, fault tolerance, load balancing, task scheduling, security, quality and the like remain a business responsibility, while the mechanisms for controlling them are no longer of a technical nature. Instead, contracts and lawyers now ensure the (service) levels of quality, security, etc. This requires alternative ways to look at systems, and that is what CC can help you obtain.

2 Modelling And The CC Technique

Here we describe how to work with the Calculate with Concepts (CC) technique so as to create models, and work with them. It does *not* explain the (mathematical) backgrounds of the CC technique. Questions such as 'why does this work', or 'why are things done this particular way' are outside the scope of this document.

Within CC, a model has two representations, one representation is in natural language sentences, and the other uses mathematical set theory. For our purposes here, we'll only use the set theory representation when that is really necessary.

A CC-model consists of:

- **Concepts**. In natural language, concepts are words that the problem-owners agree on, as (1) having some meaning that (2) is relevant for the problem at hand. Note that we don't require that the problem-owners agree as to what this meaning, or the definition of the concept, is. Definitions/meanings in the real world are irrelevant to the model.
  In set theory, a concept is a set bearing the same name as the concept in the natural language representation.
- **Relations**. In natural language, a relation is a sentence that (1) 'connects' two (and no more) concepts, and (2) the problem-owners agree on as having some meaning that (3) is relevant for the problem at hand. Again, we don't require that the problem-owners agree on what that meaning is.
  In set theory, a relation is a subset of the cross product of the two sets that are 'connected' in the natural language sentence. The name of this set can be chosen at will, but often is a verb from which the natural language sentence can be easily remembered.
- **Restrictions**. In natural language, a restriction is a sentence (in the present perfect tense) that (1) 'connects' multiple relations, and (2) the problem owners agree on, as having some meaning that (3) is relevant for the problem at hand. Again, for the modeller this meaning is irrelevant.
  In set theory, a restriction is a logic statement about relations between relations. These statements of course should not be conflicting, and this can be verified by the rules of (predicate) logic.

Basically, modelling with the CC-technique thus consists of talking with the problem-owners, and getting them to choose, and agree on, a set of sentences that, to them, describe the part of the world that they want to have modelled. From such sentences, concepts and relations are identified. The concepts are then represented as nodes in a graph, and relations are shown as paths between such nodes (see Figure 1 for an example of such a graph).

In doing so, any discussion about 'definitions' or 'meaning' can be cut short, thus saving valuable time. Checking the relevancy of concepts and relations is another means of structuring the modelling process, rendering it more efficient.

Next, we need to find the restrictions. This is done by a process called 'cycle-chasing', which consists of the following steps:

- Choose two concepts in the graph (they may be the same)
- Select two (different) paths between these two concepts, say P1 and P2.
- As both P1 and P2 are (possibly composite) relations, i.e. subsets of the same set (namely the cross-product set of the two chosen concept sets), you can wonder whether P1 $\subseteq$ P2, or if perhaps P2 $\subseteq$ P1.

Recall that there is a one-to-one correspondence between the set theoretical representation of the model and natural language sentences that problem owners have agreed to as being relevant. Hence, the propositions 'P1 $\subseteq$ P2' and 'P2 $\subseteq$ P1' can be translated into their natural language equivalents, that the problem-owners may or may not recognise as (1) having some meaning that (2) is relevant within the scope of the purpose the model is to serve.
Propositions for which there is such agreement are called 'restrictions', and are included in the model. Propositions for which there is no such agreement are left out. As with concepts and relations, the modeller may cut short discussions about meaning or definitions, and challenge the relevancy of proposed restrictions.

All this does not seem too revolutionary, but that is not the case, since everything you do to the model has consequences. For instance, introducing a new concept is only done if there is at least one relation with another concept within the model, and by definition this must somehow be meaningful to all involved. Also, removing a concept from the model implies that all relations and restrictions in which that concept occurs are removed, thus degrading the model (even though this sometimes is necessary).

So, the process of playing around with concepts, relations and restrictions, even if the number is very limited, cannot be done just by hand waving. Everything you do has consequences, and you'll be asked to account for them.

There is another thing. Models are often needed for different stakeholders that have different interests, and require the model to do different things. If you make one monolithic model, it is not easy to come up with one that is agreeable to all persons. Also, not everybody wants to know about *all* parts of the model, but only those parts (s)he is interested in.

Introducing the notion of 'discussions' accommodates for this. A discussion is a set of sentences that a group of problem-owners can agree on as being relevant for their purposes. Therefore, a discussion can be modelled in a graph exactly as described before. Whenever a model is required with different interests at stake, each interest can be modelled separately, using the language (sentences) of the people involved.

Next, these different interest groups (each having their own discussion, modelled by say models M1 and M2) must agree on which part of their models overlap, i.e. which relation in model M1 maps onto which relation in model M2. If an agreement on this overlap can be established, then the associated graphs can be 'strung together' using such relations as the 'glue'. Then again a cycle-chasing session can be started to find the restrictions that involve both discussions.

Having described the basic stuff of the CC-techniques, let's now sum up the modelling difficulties that it addresses:

1. Differences in use of the (natural) language by different stakeholders, often causing virtually endless discussions, can largely be avoided. Discarding the notion of 'definitions' altogether does this, and consequently speeds up the modelling process.
2. There is a 1-1 mapping between natural language sentences and the mathematical representations used in the model. This means that everything you do can be *formally verified* (by the math), but you can also talk about it with non-mathematicians using the language representation.
3. Restrictions, and the cycle-chasing processing of obtaining them, are novel. To our knowledge there are no methods or techniques besides CC that can actually find them[1]. The result is that the model can be remembered (by means of a simple graph) relatively easily, and yet include a wealth of semantics.
4. Everything you do to a model has immediate consequences. Adding concepts, relations, or restrictions means that you need to agree (within the scope of the discussion) that there is relevant meaning to what is added. Removing them implies that their relevancy no longer exists.
5. Since the model can be represented using set theory and logic, which are both well-known mathematical disciplines, this representation can also be used within reasoning processes, such as (dis)proving whether or not (existing, or new) relations or restrictions must hold. This is outside the scope of this document.
6. Coming up with some model is easy – coming up with a *good* model is an altogether different thing. In our experience the CC-technique is quite valuable as it continued to mercilessly point out flaws that we had in the various versions of our model.

Even though there are many issues this technique addresses, it is not *the* magic potion that solves all of our modelling problems. The journey that leads you to discover a *good* model is still untrod, and you're boldly going where no one has gone before, charting the area as it presents it to you. This technique helps you to minimise wasting time, as it aims at keeping you focussed on what you want to achieve, and keeping track of consistency within the models.

2.1 Syntax And Semantics

Figure 1 shows the toplevel discussion of the model we present in this document. We will use this figure/discussion to explain syntax and semantics.

Within Figure 1 you can see:

- **Named white boxes**, representing the **concepts** within the model. Below the name of the concept, you'll find examples of instances of the concept (mathematically speaking they are elements in the concept set). For example, the top left box in the figure represents the infrastructure components (e.g. 'router box R2', 'workstation WS24', and 'firewall box FW1') of such a system.
- **Named grey boxes** reference concepts of other discussions.
- **Named lines connecting boxes**, representing the **relations** within the model. The name associated with a line names the (mathematical) relation. Whenever a line is thick, this means that it acts as 'glue' between this discussion and another discussion. The 'crows foot' at each end of a line indicates the cardinalities of the relation; this is ignored here.

As the figure only presents part of the model – and in particular some mathematical representations – the sections following the figure contain the natural language representations of the model's concepts and relations. There are two such sections:

1. The section containing a table describing the relations. In this table, the name of the relation is in the left column. The right column holds the mathematical definition (bold), and the natural-language equivalent thereof (normal).

[1] In fact, the director of KISS (a company specialised in modelling) has acknowledged this.

2. The section containing a table describing the restrictions. In this table, the (short) mathematical notation is in bold, and the equivalent natural-language form follows in normal font. The relevancy of the restriction is then explained, e.g. by mentioning a situation that you don't want to exist in the system, and that the restriction prevents to exist. An optional paragraph can be added to provide further backgrounds or detail.

The mathematical notations used are as follows:

- **run :: CA × I** says that there is a relation between concepts CA an I, and the name of that relation is 'run'. The set represented by 'run' contains elements (x, y), with $x \in$ CA and $y \in$ I, which means that 'run' is a subset of CA $\times$ I (i.e. all possible combinations of x and y).
  This notation is used when describing relations.
- **[run]** represents the *meaning* of the relation 'run'. In our example, it is the set consisting of tuples (x, y), with $x \in$ CA and $y \in$ I for which it is true that 'content/application component x runs on infrastructure component y'.
  This notation is used when describing restrictions.
- **[flp run]** represents the inverse of [run], i.e. the set tuples (y, x), with $x \in$ CA and $y \in$ I for which it is true that 'content/application component x runs on infrastructure component y'.
  This notation is used when describing restrictions.
- **[within_ca, flp within_i]** is a relation between three concepts (in this case: CA, SP, and I). It is the set consisting of tuples (x, y), with $x \in$ CA and $y \in$ I for which it is true that 'there is at least one security perimeter $z \in$ SP such that content/application component x is within security perimeter z AND infrastructure component y is within (the same) security perimeter z'.
  This notation is used when describing restrictions.
- **[run] [within_ca, flp within_i]** represents a restriction, indicating that the set (x, y) represented by **[run]** is a subset of the set represented by **[within_ca, flp within_i]**.
  The associated natural language representation is as follows:
  'For all Content/Application components x and all infrastructure components y for which it is true that content/application component x runs on infrastructure component y, there must be at least one security perimeter z such that content/application component x is within security perimeter z AND infrastructure component y is within the same security perimeter z'.
  This notation is used when describing restrictions.

3 An Example Model

Modelling is about making choices, e.g. as to which discussions you have, what the concepts, relations and restrictions shall be, etc). Where do you want to go today? Which parts of the model (discussions) to you think will be the most relevant to the problems at hand? We are prone to make mistakes here, and we also need a bit of luck. We haven't had time to explore the problems to the extent we would have liked, and the models presented in this chapter reflect what we have discovered so far:

1. The 'toplevel' discussion is about security. After all, it is nice to know what you're talking about. It introduces notions like a security perimeter and directives, and is aimed to direct all other security related discussions.

2. The 'security' discussion models a method for defining directives, and keeping them up-to-date given business objectives and real-world threats. This model resulted from our thoughts on how the security constraints as mentioned in the toplevel discussion could actually be implemented. It has answers on questions such as: where do the directives come from? Who decides this? Etc. It remains to be seen whether the real world system this is to be applied to, can be appropriately mapped onto this model. However, as this model is a first modelling attempt, we expect it to be developed further, ant to be made consistent with security standards such as BS7799.

3. The 'infrastructure' discussion models a generic systems service architecture, including networks, (distributed) services and applications consisting of components that communicate with one another. It is a first attempt to model services, and show how such architectures can inherit security specifications. Also, this model could provide specifications that can be inherited by more specific service discussions.

In the next sections, these three discussions are described, as well as how they are 'glued' together. This will also show what the consequences are of requiring that the infrastructure comply with the security requirements imposed by the 'toplevel' discussion. Also, we will show how security issues restrict the systems operation in the case of distributed applications whose parts may be in different security perimeters. This part, although promising, has not yet been fully worked out.

The models as we present them here are not completely worked out. They are what we came up with in a limited amount of time. Further refinements, in particular the cycle chasing process, can be carried out.

Also, given that security is 'everywhere', there may be other discussions that are useful to model and subsequently glue into the current model. In particular, the client-reception-server model is a candidate for this. We are not all too worried about this, as the model itself lends itself excellently to being extended either now or in the future.

3.1 Discussion 1: 'Toplevel'

This model is designed to talk about security at a high abstraction level. All subsidiary models will, when glued to this model, inherit the restrictions of this model.

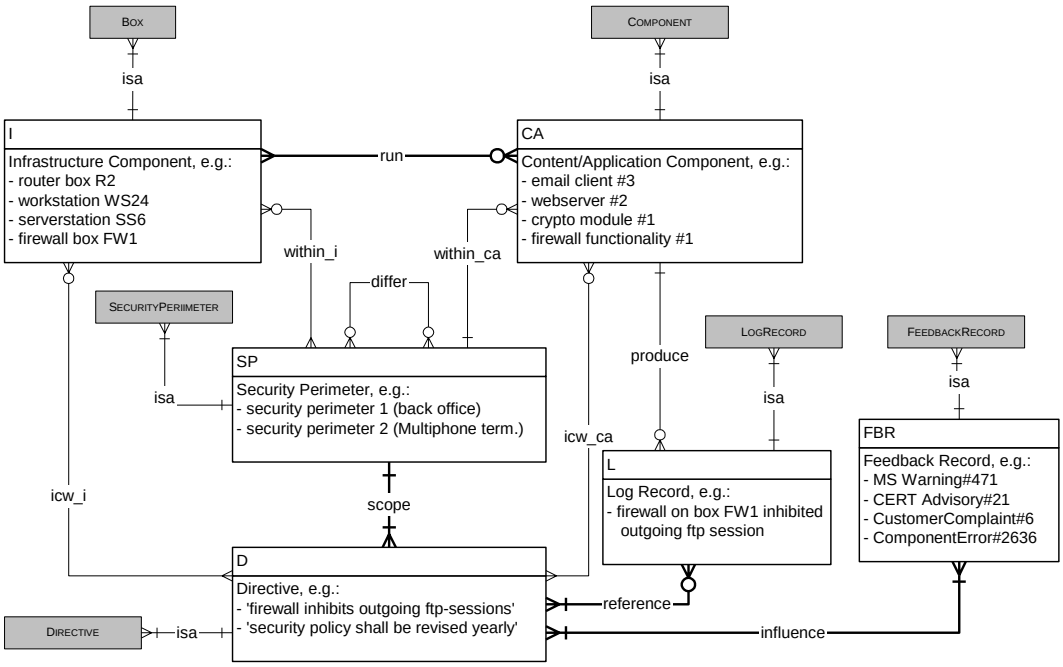

Figure 1: 'Toplevel' discussion

Note that the relations shown within the above figure have crow's feet. Although this has its uses within the CC-technique, this is not relevant for this document, and therefore should be ignored.

3.1.1 Concepts

This section describes the concepts within the discussion. It must be clear from the below what real-life elements are covered by the component, and which are not. The mathematical equivalent is that we want to have a set with well-defined elements, so that the relations on the set could be true for all elements.

CA **Content/Application components**, e.g.: email client #3, webserver #2, crypto DLL #1, firewall application #2.

We have chosen for CA-components to provide all functionality within the system. CA components must run on an I-component.

D **Directives**, e.g. 'firewall #1 shall inhibit all outgoing ftp-sessions', or 'security policy for system X shall be revised on a yearly basis'.

Directives are the statements that control all (security)parts of the system.

FBR **FeedBack Records**, such as Microsoft Warning #471, CERT advisory #21, Customer complaint#6, CA-component error #2636.

Feedback records provide the means of closing the feedback loop that stable systems require. Any information that may be of interest to the security of the system is (or should be) represented in a FBR-element.

I **Infrastructure components**, e.g.: router box R2, workstation WS24, serverstation SS6, firewall box FW1.

In this model, we have chosen that I contains 'hardware boxes', that don't do anything useful as such. In order for such a box to do something useful, it needs some software – i.e. a CA-component – to run on it. This software provides the functionality

L **Log records**, e.g. ‘firewall running on box #1 has inhibited outgoing ftp session’.

In this model, we have chosen that L contains logging sentences as produced by CA components. This provides a means for CA components to produce feedback about their own actions, or the environment as they perceive it.

SP **Security Perimeters**, such as: back-office security perimeter, terminal security perimeter (one perimeter for each individual terminal).

A security perimeter defines the scope within which directives are valid. The model only talks about stuff within a security perimeter. Everything outside a security perimeter is outside the control of the modelled system, and hence could be hostile.

3.1.2 Relations

This section describes the relations, i.e. it explicitly defines the relation in terms of a cross-product of sets. It also defines the natural language sentence that corresponds with the relation, as well as an indication to the relevancy of each relation.

differ **differ :: SP × SP**

‘Security perimeter *sp1* is not the same as security perimeter *sp2*.’

This relation is relevant where issues involving different security perimeters are to be addressed. An example would be the transfer of data or applications between the backend system and a terminal.

icw_ca **icw_ca :: CA × D**

Content/Application component *ca* is consistent with directive *d*.

If there are directives governing content or application components, but they are not adhered to, there obviously is no security, hence this relation is relevant.

icw_i **icw_i :: I × D**

Infrastructure component *i* is consistent with directive *d*.

If there are directives governing content or application components, but they are not adhered to, there obviously is no security, hence this relation is relevant.

influence **influence :: FBR × D**

Feedback record *fbr* influences directive *d*.

This shows that directives don’t just drop out of the blue, but rather are based on general security knowledge represented by feedback records. Strictly speaking, this relation belongs in another discussion, but we included it here as we have not modelled this other discussion.

produce **produce :: CA × L**

Content/Application component *ca* produces a log *l*

If an application never produces a log, it is difficult to tell whether or not it complies with directives.

run **run :: CA × I**

Content/Application component *ca* runs on infrastructure component *i*.

This binds software and hardware, which is relevant in case we need to talk about physical security.

scope **scope :: SP × D**

Security perimeter *sp* defines the scope within which directive *d* operates.

This is relevant as we don't like directives that apply to multiple security perimeters. This is a question of keeping oversight within the context of each security perimeter.

within_ca **within_ca :: CA × SP**

Content/Application component *ca* is within security perimeter *sp*.

We only want to deal with content/application (i.e. data/software) components that are within a security perimeter that we control. All other content is outside the scope of this model.

within_i **within_i :: I × SP**

Infrastructure component *i* is within security perimeter *sp*.

We only want to deal with infrastructure components (i.e. hardware boxes) that are within a security perimeter that we control. All other content is outside the scope of this model.

3.1.3 Restrictions

This section lists the restrictions that we have identified for this discussion. Each restiction is represented by a notation reflecting the mathematical model, and also by the corresponding natural language sentence. In cases, comments are added that are meant to illustrate what the restriction is about, showing the relevancy.

1. **[run] [within_ca, flp within_i]**

   'If a Content/Application component *ca* has run on an Infrastructure component *i*, then both *ca* and *i* must have lied within the same security perimeter *sp*'.

   As a security perimeter defines the scope of (a set of) directives, this restriction ensures that the set of directives within one security perimeter covers any hardware/software interaction within that security perimeter.

2. **[run] [icw_ca, flp scope, flp within_i]**

   'If a Content/Application component *ca* has run on an Infrastructure component *i*, then there must exist a directive *d* and a security perimeter *sp* such that that *ca* is consistent with *d*, the scope of which is defined by *sp*, and *i* must have lied within the same security perimeter *sp*'. Note that because restriction 1 is also true, the CA component *ca* must also have lied within the same security perimeter *sp*.

3. **[reference] [flp produce, icw_ca]**

   'If a log record *l* references a directive *d*, this *l* must have been produced by a Content/Application component *ca* that is consistent with *d*.

   This restriction deals with the sanity of the system. We should not allow software, that does not comply with applicable directives, to produce logs that reference

directives (note that software complies with ANY directive that does not apply to that software).

4. **[icw_ca] [produce, reference]**

   'If a Content/Application component *ca* is consistent with directive *d*, then *ca* must have produced at least one log record *l* that references *d*.'

   What this says is that software, in order to be consistent with directives governing its functionality, must have produced log records showing such consistency. In other words: every directive that is applicable to a software component must be accounted for in the components test suite.

5. **[icw_i] [flp run, produce, reference]**

   'If an Infrastructure component *i* is consistent with directive *d*, then there must be at least one Content/Application component *ca* and at least one log record *l* with the properties that *ca* has produced *l*, and *l* has referenced *d*.'

   What this says is that there is a feedback loop for hardware directives, too: all hardware, in order to be consistent with a directive governing that hardware, must be shown to be consistent. But since hardware cannot do this in our model, it must be done by software running on that hardware.

6. **[icw_i] [within_i, scope]**

   'If an Infrastructure component *i* is consistent with directive *d*, then there is a security perimeter *sp* such that *i* is within *sp*, and *sp* defines the scope within which *d* operates.

   This tells us that directives are pretty specific, since there cannot be directives *d* that could possibly apply to an *i* that is not within *d*'s scope (i.e. *d*'s *sp*).

   Note that this restriction is not something that you would have in real-life, as the amount of directives would be enormous. This does not pose a problem when it can be shown that there is a proper mapping between the real-world system and this model.

7. **[icw_ca] [within_ca, scope]**

   'If a CA-component *ca* is consistent with directive *d*, then there is a security perimeter *sp* such that *ca* is within *sp*, and *sp* defines the scope within which *d* operates.

   This tells us that directives are pretty specific, since there cannot be directives *d* that could possibly apply to a *ca* that is not within *d*'s scope (i.e. *d*'s *sp*).

   Note that this restriction is not something that you would have in real-life, as the amount of directives would be enormous. This does not pose a problem when it can be shown that there is a proper mapping between the real-world system and this model.

3.2 Discussion 2: 'Security'

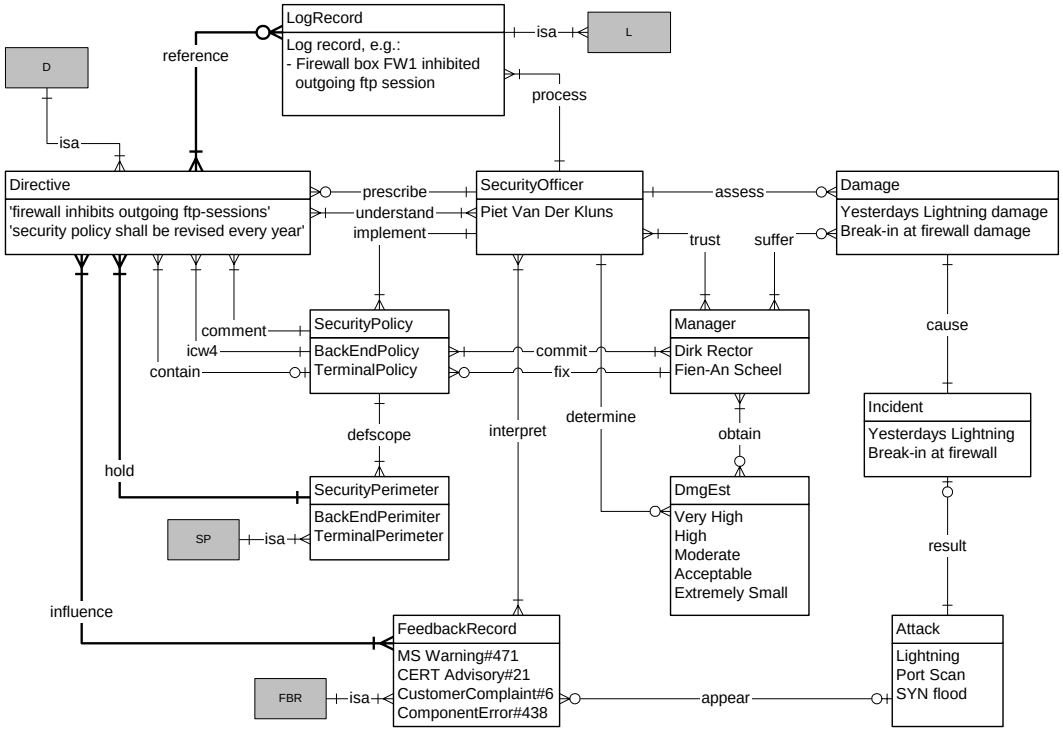

Figure 2: 'Security' discussion

Note that the relations shown within the above figure have crow's feet. Although this has its uses within the CC-technique, this is not relevant for this document, and therefore should be ignored.

3.2.1 Concepts

Attack — **Attack**, e.g. yesterdays lightning, the port scan of last week Friday, etc.

This set contains events that took place at a given time, and have the property that they could result in damage.

Damage — **Damage**, e.g. the damage resulting from yesterdays lightning.

Damage can be both monetary damage or loss of image.

Directive — **Directive**, e.g. 'firewall on box#3 must inhibit outgoing ftp-sessions', 'security policy governing this directive must be revised every year', etc.

The set Directive contains text-instances that say something about how things work within a security perimeter. It is an instruction (restriction perhaps) for any hardware or software component, person, process, etc. within the scope of the security perimeter to which the directive belongs.

DmgEst — **Damage Estimate**, e.g. 'Very High', or 'Extremely Small'

This is a measure by which damages can be predicted

Feedback-Record — **Feedback Record**, e.g. Microsoft warning$471, CERT Advisory#21, Customer complaint#6, component error#438

Feedback records are meant to be evaluated by security officers, who will use the information contained herein to maintain the directives for which they are responsible.

Incident	**Incident**, e.g. ‘Break in at firewall#3 of last week Friday’. An attack that results in a damage is called an incident.
LogRecord	**Log Record**, e.g. ‘Firewall box FW1 inhibited outgoing ftp session’ Log records contain information that provide feedback that can be used to maintain directives.
Manager	**Manager**, e.g. Dirk Rector, John Doe, Fien-An Scheel (she’s Dutch). Managers are responsible for the business, even if they delegate security (and other) issues to other people.
Security–Officer	**Security Officer**, e.g. Pete Vanderkluns, Jane Doe, etc. Security Officers are executives that managers delegate the security responsibility to.
Security–Perimeter	**Security Perimeter**, e.g. Back office security perimeter, or security perimeter for terminal#7 Security Perimeters define the scope within which a security policy, and directives, apply.
Security–Policy	**Security Policy**, Back office security policy, Termianl security policy, Company security policy A security policy is a policy document (i.e. fixed and endorsed by management) that governs the way security issues are to be treated.

3.2.2 Relations

prescribe	**prescribe :: SecurityOfficer × Directive** ‘Security Officer *so* prescribes directive *d*.‘ *So*meone must be responsible for directives. We have modelled that for each directive, a single per*so*n is responsible – this is the per*so*n that has prescribed that directive.
understand	**understand :: SecurityOfficer x Directive** ‘Security officer *so* understands directive *d*.’ If a security officer is incompetent, there is no point in discussing security.
icw	**icw :: SecurityPolicy x Directive** 'Security policy *sp* is consistent with directive *d*.' There shall be directives governing a security policy. An example of such a directive could be: ‘any directive in a security policy shall have a security officer assigned that is responsible for that directive’.
fix	**fix :: Manager x SecurityPolicy** 'Manager *m* fixes security policy *sp*.' Management must decide that a given document contains the security policy, and that the organisation must adhere to this policy. This also holds for changes in such a document.
commit	**commit :: Manager x SecurityPolicy**

'Manager *m* commits itself to security policy *sp*.'

Whenever a security policy is fixed, management must commit itself to that policy, or else it won't have any effect in the organisation.

implement **implement :: SecurityOfficer x SecurityPolicy**

'Security officer *so* implements security policy *sp*.'

Security officers are responsible for the execution of the security policy.

contain **contain :: SecurityPolicy x Directive**

'Security policy *sp* contains directive *d*.'

Examples of security policy directives are:
- 'this security policy must be evaluated on a yearly basis'
- 'security officer must provide an overview of incidents to management on a quaterly basis'
- 'all directives within this policy are the responsibility of security officer X'

trust **trust :: Manager x SecurityOfficer**

'Manager *m* trusts that security officer *so* is competent.'

Since management is responsible for the business, and security is (also) about whom/what can be trusted, it is imperative that management trusts all security officers.

assess **assess :: SecurityOfficer x Damage**

'Security officer *so* assesses damage *d*.'

Assessment of damage is part of the feedback loop to make security better.

suffer **suffer :: Manager x Damage**

'Manager *m* suffers damage *d*.'

From the business perspective, damages, either monetary or image damages, are suffered by the management.

influence **influence :: FeedbackRecord x Directive**

'Feedback record *fbr* influences directive *d*.'

Not having feedback records influence directives is a pretty stubborn attitude to have in an organisation.

appear **appear :: Attack x FeedbackRecord**

'Attack *a* appears in feedback record *fbr*.'

For all known attacks, there's a description around somewhere. Such descriptions are considered feedback records.

result **result :: Attack x Incident**

'Attack *a* results in incident *i*.'

Attacks may, or may not, result in damage. Attacks that do are called incidents.

cause **cause :: Incident x Damage**

'Incident *i* causes damage *d*.'

Attacks may, or may not, result in damage. Attacks that do are called

incidents.

process **process :: SecurityOfficer x LogRecord**

'Security officer *so* processes log record *l*.'

System logs must be inspected, and acted upon.

reference **reference :: LogRecord x Directive**

'Log record *l* references directive *d*.'

If log records do not refer to directives, either implicitly or explicitly, they have no meaning within this model.

interpret **interpret :: SecurityOfficer x FeedbackRecord**

'Security officer *so* interprets feedback record *fbr*.'

Security officers do this to keep up to date with current and new security developments, and perhaps change directives to accommodate for new types of attacks.

comment **comment :: SecurityPolicy x Directive**

'Security policy *sp* comments on directive *d*.'

Security policies talk about directives so as to assure that you can't just have any sentence become a directive.

obtain **obtain :: Manager x DmgEst**

'Manager *m* obtains damage estimate *de*.'

Managers don't do the damage estimate themselves. They leave that to security officers.

determine **determine :: SecurityOfficer x DmgEst**

'Security officer *so* determines damage estimate *de*.'

Managers don't do the damage estimate themselves. They leave that to security officers.

hold **hold :: SecurityPerimeter x Directive**

'Security perimeter *p* holds directive *d*.'

Each directive is part of (belongs to) a security perimeter.

defscope **defscope :: SecurityPolicy x SecurityPerimeter**

'Security policy *sp* defines the validity scope for security perimeter *p*.'

The scope of a security perimeter is defined the security policy governing the directives in the scope of that security perimeter.

3.2.3 Restrictions

This section lists the restrictions that we have identified for this discussion. Each restriction is represented by a notation reflecting the mathematical model, and also by the corresponding natural language sentence. In some cases, comments are added that are meant to illustrate what the restriction is about, showing the relevancy.

1. **[Prescribe] [Understand]**

   For all s in SecurityOfficer, d in Directive:
   s is prescriber(d) ==> s has understood d.

   This restriction demands that security officers should only prescribe what they understand.

2. **[Icw] [flp Implement, Prescribe]**

   For all s in SecurityPolicy, d in Directive:
   d is consistent with s ==> implementor(s) = prescriber(d).

   This restriction says that directives that are consistent with a security policy are to be prescribed by the implementor of the security policy. Hence, for every security policy, and all directives contained therein, there is only one security officer.

3. **[Fix] [Commit]**

   For all m in Manager, s in SecurityPolicy:
   m has fixed s ==> m has committed to s.

   Managers commit themselves to security policies by fixing them.

4. **[Fix, Contain] [Trust, Prescribe]**

   For all m in Manager, d in Directive:
   m has fixed security policy containing d ==> m has trusted prescriber(d).

   Managers that fix a security policy must trust the prescriber of (i.e. security officer responsible for) directives contained in this policy. Note that in combination with restriction 2, this implies that managers that fix a security policy must have assigned a security officer to manage this policy, and they must trust this officer.

5. **[Trust, Implement] [Commit]**

   For all m in Manager, s in SecurityPolicy:
   m has trusted implementor(s) ==> m has committed to s.

   This is another way of stating that managers must trust security officers that are responsible for security policies that these managers have fixed.

6. **[Contain] [Icw]**

   For all s in SecurityPolicy, d in Directive:
   s contains d ==> d is consistent with s.

   Directives contained in the security policy must be consistent with (the directives in) the security policy.

7. **[Suffer] [Trust, Assess]**

   For all m in Manager, d in Damage: Exists s in SecurityOfficer:
   m has suffered d ==> m has trusted s AND s has assessed d.

   Damages suffered by a manager must be assessed by a security officer trusted by that manager.

8. **[Assess] [Prescribe, flp Influence, flp Appear, Result, Cause]**

   For all s in SecurityOfficer, d in Damage: Exists d' in Directive, f in FeedbackRecord, a in Attack:
   s has assessed d ==> s is prescriber(d') AND f has influenced d' AND a that has resulted in i which in turn caused d, has appeared in f.

Security officers assessing a damage must be prescriber of directives that are influenced by feedback records concerning attacks that cause these damages.

9. **[Influence] [flp Interpret, Prescribe]**

For all f in FeedbackRecord, d in Directive:
f has influenced d ==> prescriber(d) has interpreted f.

If some feedback record has influenced a directive, then the security officer that has prescribed this directive must have interpreted the feedback record.

10. **[Process] [Understand, flp Reference]**

For all s in SecurityOfficer, l in LogRecord: Exists d in Directive:
l has been processed by s ==> s has understood d AND l has referenced d.

Security officers that understand the directives log records refer to must process such log records.

11. **[Prescribe, flp Contain] [Implement]**

For all s in SecurityOfficer, s' in SecurityPolicy, d in Directive:
s has prescribed d AND s' is the security policy containing d ==> s is the implementor of s'.

12. **[flp Contain, Comment, flp Prescribe] [flp Understand]**

For all d, d' in Directive, s in SecurityOfficer: Exists s’ in SecurityPolicy
d is contained in s’ that comments on d’ AND s prescriber of d' ==> s has understood d

Security officers prescribing directives must understand the security policy directives that comment on directives (in general).

13. **[Obtain] [Trust, Determine]**

For all m in Manager, d in DmgEst: Exists s in SecurityOfficer:
m has obtained d ==> m has trusted s that has determined(d).

14. **Hold == [flp DefScope, Contain]**

For all s in SecurityPerimeter, d in Directive:
s holds d <=> security policy containing d has defined scope of s.

15. **Influence == [flp Interpret, Prescribe]**

For all f in FeedbackRecord, d in Directive:
f influen**c**e d <=> prescriber of(d) has interpreted f

3.2.4 Gluing

Figure 1 and Figure 2 show some relations with thicker lines that the other relations. Such relations are considered ‘glued’ to one another. This means that relations, whose concepts correspond with each other, are considered to be ‘the same’. Mathematically speaking, gluing two relations means that we model one of these relations as a subset of the relation it is glued to. This is a modelling decision, as this cannot be proved mathematically. The consequences are that the relation that is a subset of the other relation ‘inherits’ the properties of the relation it is a subset of (as do the concepts). In the figures, the grey boxes and the ‘isa’ relation connecting these boxes show this.

In the ‘Security’ model (Figure 2), the relations ‘reference’, ‘hold’, and ‘influence’ are glued to the ‘Toplevel’ model (Figure 1) relations ‘reference’, ‘scope’, and ‘influence’ respectively.

The consequences of this gluing are that all Directives, FeedbackRecords, LogRecords, and SecurityPerimeters as defined in the ‘Security’ model must also satisfy the relations

and restrictions in the ‘Toplevel’ model applying for the concepts D, FBR, L, and SP respectively.

As the Security discussion is a worked out toplevel discussion, this does not come as a big surprise.

3.3 Discussion 3: 'Infrastructure'

As this discussion does not seem to be of primary importance, we have taken the liberty to document it pretty much straight from our tools. While already saying quite a lot about generic infrastructures and services running on such infrastructures, we must stress that restrictions have not been exhaustively searched for, so in this sense this model is incomplete.

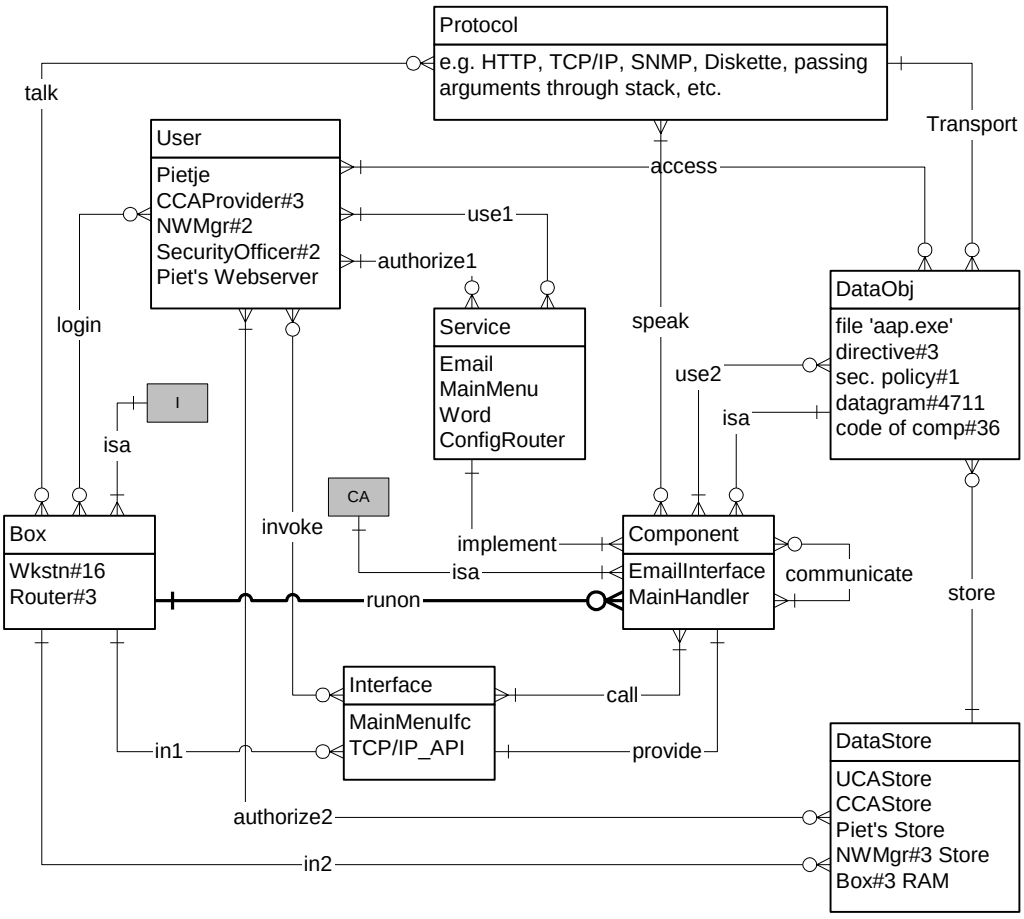

Figure 3: 'Infrastructure' discussion

Note that the relations shown within the above figure have crow's feet. Although this has its uses within the CC-technique, this is not relevant for this document, and therefore should be ignored.

3.3.1 Concepts

Box — **Boxes**, i.e. e.g.: router box R2, workstation WS24, serverstation SS6, firewall box FW1.

In this model, we have chosen that I contains 'hardware boxes', that don't do anything useful as such. In order for such a box to do something useful, it needs some software – i.e. a CA-component – to run on it. This software provides the functionality

DataObj — **Data Object**, e.g. the file 'aap.exe', directive #3, the backoffice security policy, a datagram, the code of a component, etc.

In this model, anything that can be stored and transported is a data object.

Component	**(Software) Components**, e.g: email client #3, webserver #2, crypto DLL #1, firewall application #2. We have chosen for CA-components to provide all functionality within the system. CA components must run on an I-component.
DataStore	**Data Store**, e.g. uncontrolled-content store, controlled content/application store, store of user Piet, network manager#3 store, RAM in Box#3, etc.
Interface	**Interface**, e.g. the main menu user interface on box#3, the TCP/IP API in box#5, etc.
Protocol	**Protocol**, e.g. HTTP, TCP/IP, SNMP, COM/DCOM, but also the method for passing arguments from one component to another through the stack, or passing data by means of floppy disk, etc. Essentially, any method by which data objects are transferred from one component to another is a protocol.
Service	**Service**, e.g. Email, the main menu, MS Word, a router configuration program, etc. Services are what the user wants; they are implemented by a collection of components that may run on a variety of boxes.
User	**User**, e.g. Pete Vanderkluns, Content provider#3, Network manager#5, Security Officer#5, but also services that may use other services (e.g. John's webserver may host pages that call other services such as email) To some extent, a user can be compared to a component as it invokes (user)interfaces. The difference is that a user does not necessarily implement a service, whereas a component does.

3.3.2 Relations

access	**access :: User × DataObj** User *u* accesses data object *d*.
authorize1	**authorize1 :: User × Service** User *u* is authorized to use service *s*.
authorize2	**authorize2 :: User × DataStore** User *u* is authorized to store data in data store *s*.
call	**call :: Component × Interface** Component *c* calls interface *i*.
communicate	**communicate :: Component × Component** Component *c* communicates with (other) component *c'*.
implement	**implement :: Component × Service** Component *c* implements (a part of) service *s*.
in1	**in1 :: Interface × Box** Interface *i* resides in box *b*.

in2	**in2 :: DataStore × Box** Data store *s* resides in box *b*.
invoke	**invoke :: User × Interface** User *u* invokes interface *i*.
login	**login :: User × Box** User *u* is logged in onto box *b*.
provide	**provide :: Component × Interface** Component *c* provides interface *i*.
runon	**runon :: Component × Box** Component *c* runs on box *b*.
speak	**speak :: Component × Protocol** Component *c* speaks protocol *p*.
store	**store :: DataStore × DataObj** Data store *s* stores data object *d*.
talk	**talk :: Box × Protocol** Box *b* talks using protocol *p*.
transport	**transport :: Protocol × DataObj** Protocol *p* transports data object *d*.
use1	**use1 :: User × Service** User *u* uses service *s*.
use2	**use2 :: Component × DataObj** Component *c* uses data object *d*.

3.3.3 Restrictions

1. **[invoke] [authorize1, flp implement, provide]**

   For all u in User, i in Interface: Exists c in Component:
   u has_invoked i ==> u has_been_authorized_to_use
   service_implemented_through(c) AND i = interface_provided_by(c)

2. [invoke, flp provide, implement] [authorize1]

   For all u in User, s in Service, c in Component:
   u has_invoked interface_provided_by(c) AND s =
   service_implemented_through(c) ==> u has_been_authorized_to_use s

3. **[invoke] [login, flp in1]**

   For all u in User, i in Interface:
   u has_invoked i ==> u has_been_logged_in_onto box_containing_interface(i)

4. **[Invoke, In1] [Login]**

   For all u in User, b in Box, i in Interface:

u has_invoked i AND b = box_containing_interface(i) ==> u has_been_logged_in_onto b

5. [Invoke, flp Provide, Call, flp Provide, Implement] [Authorize1]

   For all u in User, s in Service, c, c' in Component:
   u has_invoked interface_provided_by(c) AND c has_called interface_provided_by(c') AND s = service_implemented_through(c') ==> u has_been_authorized_to_use s

6. **[Invoke, flp Provide, Use2, flp Store] [Authorize2]**

   For all u in User, d in DataStore, c in Component, d' in DataObj:
   u has_invoked interface_provided_by(c) AND c has_used d' AND d has_stored d' ==> u has_been_authorized_to_use d

7. **[Use1] [Authorize1]**

   For all u in User, s in Service:
   u has_used s ==> u has_been_authorized_to_use s

8. [Authorize1, flp Implement, Use2, flp Store] [Authorize2]

   For all u in User, d in DataStore, c in Component, d' in DataObj:
   u has_been_authorized_to_use service_implemented_through(c) AND c has_used d' AND d has_stored d' ==> u has_been_authorized_to_use d

9. **[Access] [Authorize2, Store]**

   For all u in User, d in DataObj:
   u has_accessed d ==> u has_been_authorized_to_use has_stored(d)

10. **[Access] [Authorize1, flp Implement, Use2]**

    For all u in User, d in DataObj: Exists c in Component:
    u has_accessed d ==> u has_been_authorized_to_use service_implemented_through(c) AND c has_used d

11. **[Use2, flp Transport] [Speak]**

    For all c in Component, p in Protocol, d in DataObj:
    c has_used d AND p protocol_that_transports d ==> c has_spoken p

12. **[Call, flp Provide] [Speak, flp Speak]**

    For all c, c' in Component: Exists p in Protocol:
    c has_called interface_provided_by(c') ==> c has_spoken p AND c' has_spoken p

13. **[Communicate] [Speak, flp Speak]**

    For all c, c' in Component: Exists p in Protocol:
    c has_communicated_with c' ==> c has_spoken p AND c' has_spoken p

14. **[Call, flp Provide] [Communicate]**

    For all c, c' in Component:
    c has_called interface_provided_by(c') ==> c has_communicated_with c'

15. **[Provide, flp Call] [Communicate]**

    For all c, c' in Component:
    c' has_called interface_provided_by(c) ==> c has_communicated_with c'

16. **[Call, flp Provide, Communicate] [Communicate]**

    For all c, c', c'' in Component:
    c has_called interface_provided_by(c'') AND c'' has_communicated_with c' ==> c has_communicated_with c'

17. **[Talk, Transport] [flp RunOn, Use2]**

    For all b in Box, d in DataObj: Exists c in Component:
    b has_talked protocol_that_transports(d) ==> b = box_that_runs(c) AND c has_used d

18. **[Talk, Transport] [flp RunOn, Communicate, Use2]**

    For all b in Box, d in DataObj: Exists c, c' in Component:
    b has_talked protocol_that_transports(d) ==> b = box_that_runs(c) AND c has_communicated_with c' AND c' has_used d

19. [Talk, Transport] [flp RunOn, Communicate, Communicate, Use2]

    For all b in Box, d in DataObj: Exists c, c', c'' in Component:
    b has_talked protocol_that_transports(d) ==> b = box_that_runs(c) AND c has_communicated_with c' AND c' has_communicated_with c'' AND c'' has_used d

3.3.4 Gluing

Figure 1 and Figure 3 show some relations with thicker lines that the other relations. Such relations are considered ‘glued’ to one another. This means that relations, whose concepts correspond with each other, are considered to be ‘the same’. Mathematically speaking, gluing two relations means that we model one of these relations as a subset of the relation it is glued to. This is a modelling decision, as this cannot be proved mathematically. The consequences are that the relation that is a subset of the other relation ‘inherits’ the properties of the relation it is a subset of (as do the concepts). In the figures, the grey boxes and the ‘isa’ relation connecting these boxes show this.

In the ‘Infrastructure’ model (Figure 3), the relation ‘runon’ is glued to the ‘Toplevel’ model (Figure 1) relation ‘run’.

The consequences of this gluing are that all Boxes and Components as defined in the ‘Infrastructure’ model must also satisfy the relations and restrictions in the ‘Toplevel’ model applying for the concepts I and CA respectively.

As the Infrastructure discussion has been set up without any presupposed link to the Toplevel discussion, we are in for the following ‘surprises’ (which are simply the restrictions of the Toplevel discussion applied to the concepts of the Infrastructure discussion, as can be easily verified):

1. If a component *c* has run on a box *b*, then both *c* and *b* must be within the same security perimeter *sp*.
2. If a component *c* has run on a box *b***,** then there must be a directive *d* and a security perimeter *sp* such that *c* is consistent with *d*, the scope of which is defined by *sp*, and *b* must have lied within the same security perimeter *sp*’. Note that because restriction 1 is also true, *c* must also have lied within the same *sp*.
3. If a log record *l* references a directive *d*, this *l* must have been produced by a component *c* that is consistent with *d*.

4. If a component *c* is consistent with directive *d*, then *c* must have produced at least one log record *l* that references *d*.

5. If box *b* is consistent with directive *d*, then there must be at least one component *c* and at least one log record *l* with the properties that *c* has produced *l*, and *l* has referenced *d*.

6. If box *b* is consistent with directive *d*, then there is a security perimeter *sp* such that *b* is within *sp*, and *sp* defines the scope within which *d* operates.

7. If a component *c* is consistent with directive *d*, then there is a security perimeter *sp* such that *c* is within *sp*, and *sp* defines the scope within which *d* operates.

These restrictions, or specifications if you will, are all for free once the modelling decision has been made to glue the (infrastructure) relation ‘runon’ onto the (toplevel) relation ‘run’. The resulting restrictions 3 and 4 show that this is not trivial, as they specify that (software) components must produce log records – this was previously unspecified, and there must be log records referencing (appropriate) directives before they can be considered to comply with directives. This is not bad for a simple inheritance!

Apart from inheritance, we can do some more cycle chasing, focussing on the glued relations and concepts. This could result in additional restrictions, for example:

8. **[infraCommunicate] [within_ca, flp within_ca]**

   If components *c* and *c’* have communicated with one another, then *c* must be within the same security perimeter as *c’*.

   Please note that we don’t advise this restrictions per se; this depends on the actual situation at hand. It is only given to illustrate the cycle chasing process between two discussions that have been glued.

There are other restrictions that we would like to impose, but for which the model(s) up to this point are not suitable. For example, we currently have no way of talking about ‘secure communications’ between two components, communications between components that are in different security perimeters, etc. For such and other discussions, we need to do more work on the models.

4 Working With The Model

Working with the model means: getting an answer for the (security) questions or concerns that you may have. Using the model must fulfil the purposes that the model is supposed to serve.

Roughly speaking, here's how it works.

1. Identify a question or concern in 'customer language'
2. Map the question or concern onto the model. This means that you must rephrase it in terms of the natural language sentences that are within the model. Here you change the frame of reference from that of the customer to that of the model.

If this succeeds, go to step 3. If it fails, then the model may not be adequate enough for the concern, and either one or more discussions might need to be revised, or another discussion could be added in such a way that the model addresses the concern.

3. Within the model, find the restrictions that apply to the concern in question. As these restrictions are invariants, they are always true within the model.
4. Next, go back to the 'customer' frame of reference, and find out whether or not these restrictions are true for the real-world system. As you have pinpointed where in the system you need to look, you may now decide what, if at all, to do about it.

The following sections will provide some examples of how this works.

4.1 Content Stealing.

Step 1: state the concern. The concern we would like to deal with here is that a customer that uses a terminal steals content, i.e. a user that legitimately obtains content (e.g. views it on the terminal screen), but then manages to create a copy of this content, say on a floppy, a server on the internet, or what else.

Step 2: map the concern onto the model. We do this by looking at the 'Infrastructure' discussion, as this discussion addresses things like 'terminals' (which are considered instances of 'Box' here), 'content' (obviously 'DataObj'), and users (id). 'Legitimately' has to do with authorisation, copying has to do with the relations 'access' and 'store'. Floppy disk, a storage space on the Internet etc. are all 'store's.

Step 3: find applicable restrictions. Since 'copying' is a form of [access; store] we check section 3.3.3 and find the following restrictions:

- [infraInvoke, flp infraProvide, infraUse2, flp infraStore] [infraAuthorize2]
- [infraUse1] [infraAuthorize1]
- [Authorize1, flp Implement, Use2, flp Store] [Authorize2]
- [Access] [Authorize2, Store]
- [Access] [Authorize1, flp Implement, Use2]

Together, these restrictions state that users:

- must be authorised for any service they use.
- must be authorised for any store used by that service.

Step 4: How do these restrictions (not) apply to the real-world system? If a user can steal content, then he is either using a service that he should not be able to access (so there is a service-authorisation problem), or he is using a store for which he should not be authorised. If we're unhappy with the current situation, then this might lead to a

discussion focussing on store authorisation specifications, service authorisation specifications, and/or how such authorisations might be combined.

4.2 Back Office Spoofing.

The current models cannot address back office spoofing as restrictions (currently) do not allow components that are part of a single service to be in separate security perimeters, and there is also no language dealing with communications to components that are outside any security perimeter. Discussions dealing with this will therefore have to be modelled, and glued into the existing models.

4.3 Providing Content/Applications

Step 1: Application providers can provide either content – this content may include applications (scripts, applets, etc). The worry we have here is how such content enters the security perimeter of our system.

Step 2, Step3: Mapping this onto our model, we would like to see such content end up as CA-elements in the Toplevel discussion. As restrictions apply for such CA-elements, and it is not self-evident that such restrictions are met, there must be a provision within the system that ensures that provided content cannot enter the security perimeter before it is guaranteed to comply with applicable directives. This means for example that it must be shown to produce logfiles that reference the applicable directives (as a means of showing that it is consistent with such directives).

Also, if we would like to map the provided content onto (infrastructure discussion) components, it must be verified that:
- whoever stores this content/application must be authorised to use the service that allows such data to be transferred into the system;
- this person must also be authorised for the store in which the content/application is going to be stored;
- this component (1) calls interfaces from the given interfaces-set, (2) speaks protocols from the protocol-set, (3) implements a service from the services set, and (4) is able to properly check for service and storage authorisations.

Step 4: Here we put the question how the real world system complies with the statements of the earlier steps. You need to have a real world system at hand to talk about this.

Note that the current models do not allow components to be in separate security perimeters, and there is also no language dealing with communications to components that are outside any security perimeter. This inhibits our dealing with the process itself of transferring content/application into the system. Discussions dealing with this can still be modelled, and glued into the existing model.

4.4 What Security Perimeters Can Do to Each Other

Step1: Suppose we have a system consisting of a Client system in one security perimeter, and a Server system in another security perimeter. This is the case if you have a back office (server system), and terminals that you can actually control (e.g. satellite terminals in case you're in that business).

The basic actions of terminals that we're worried about here involve the sending of malicious service requests to the back office, causing some damage being done. We are also concerned that a terminal could connect to other service providers, perhaps even to the internet in an uncontrolled manner, leaving the internals of the terminal wide open to attack.

Step2/Step3: As the back office is within one security perimeter, the situation at hand resembles the sending of content/application into the back office; the statements made by our models about this situation are described in section 4.3. But now these statements hold not only for the back office, but for the terminals as well.

Step 4: From the above, it follows that neither the back office, nor the terminal, should off-hand accept that the communication partner is ‘the expected one’ (i.e. the terminal for the back office, and vice versa). Proper authentication and authorisation mechanisms should be in place both at the terminal and the back office side.