Department of Computer Science
George Mason University
Technical Report Series

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/
703-993-1530

Security Policy Cognizant Module Composition

Paul Seymer, Angelos Stavrou, Duminda Wijesekera, and Sushil Jajodia
{pseymer|astavrou|dwijesek|jajodia}@gmu.edu

Technical Report GMU-CS-TR-2010-1

Abstract

Component-based software development and deployment is based on developing individual software modules that are composed on an as needed basis. Such modules expose the computations they provide and their dependencies on providing these computations - that results in a well known *requires-provides* specifications for modules. This paper provides a framework to combine modules that specify their requires-provides interfaces in a policy dependent way. Our framework specify policies as combinations of Constraint Logic Programming (CLP) based rules and our policies can cover multiple *aspects* associated of compositions, such as security and quality of service. We apply our framework to specify Quality of Protection (QoP) and Quality of Service (QoP) policies. An example shows the applicability of our policy language to a teleconferencing application with multiple security and resource usage policies.

1 Introduction

In component-based software development, modules are developed independently and composed on an *as needed* basis [30]. In order to facilitate such compositions (either statically or dynamically), the composer needs to be aware of what each module provides as a service and what it requires in order to provide that service - resulting in the well known *requires-provides* specification of modules. Making these *requires-provides* interfaces policy dependent is the objective of this paper. In order to do so, we propose a *Constraint Logic Programming* based rule language to specify the *requires-provides policies*. Our sample policies shown in this paper addresses two *aspects* of software composition; viz., *Quality of Protection (QoP)* and *Quality of Service (QoS)*, specifying security and performance policies. For QoP, we show two kinds policies; viz., access to individual modules and information flown between them as a consequent to composition. For QoS, we show how to integrate local resource control policies of individual modules and communication resource policies between them. Figure 1 shows our overall architecture.

As an example, consider multimedia conferencing, where the conference coordinator residing at one site wants to use two other conferees with differing computing and communicating resources that are subjected to different access control and communication policies. For example, the coordinator may reside at a site that allows video, audio, and HAIPE encryption, where one participant with a land-line may only be allowed to use DES encryption with audio and video capabilities and the other participant joining by mobile telephone may only have the audio stream, but no video or encryption capabilities. Consequently, for the modules to be correctly composed, the conference should deliver both audio and video encrypted using DES to the second participant and only an audio stream to the third participant.

The advantage of this framework are many. Firstly, it separates independent aspects of composition (e.g. QoP and QoS) from structural composability criteria. Secondly, it further provides a framework to separates different sub-aspects within an aspects. Thirdly, it provides a basis to reason about all aspects and sub-aspects uniformly. Lastly, it adds policies to aspect-oriented composition.

The rest of the paper is written as follows. Section 2 presents the formal language of the rule-based composition framework that is sufficiently expressible to accommodate many notions of module composition mostly reflecting existing definitions. Section 3 presents the sub-language for security policies. Section 4 presents the

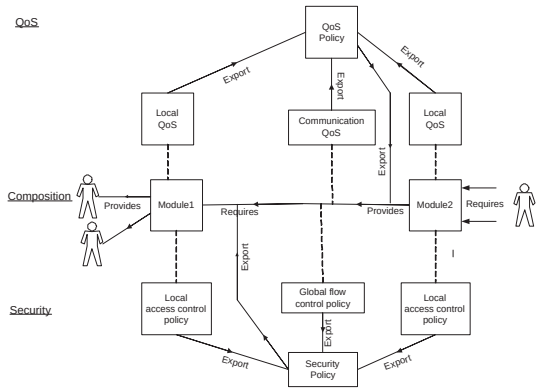

Figure 1: Architecture of the Policy Framework

sub-language for QoS policies. Section 5 presents the semantics and models for module composition syntax. Section 6 presents an example composition of a teleconference. Section 7 presents related work and Section 8 presents our conclusions.

2 A Rule-based Language to Specify Module Composition

We propose using a *set constraint* based logic programming language to specify policies. The version of set theory we use is CLP($\mathcal{SET}$), the *hereditarily finite* set theory developed by Dovier et al. [15, 16, 17], where the hereditarily finiteness refers to the fact that all sets are constructed out of a finite universe (of so called *urelements*) by applying a finite collection of composition operators. The set of operators we use are $\{=, \neq, \in, \notin, \cup_3, / \cup_3, \|, \not\|\}$, where $\cup_3$ is the ternary predicate $X \cup_3 Y = Z$, An analogous explanation applies for $\cap_3$. Similarly $X \parallel Y$ holds iff $X \cap Y = \emptyset$. Our constraints are conjunctions and disjunctions of these constraint predicates, instantiated with terms belonging to a chosen set of *sorts*. We use five sorts for Modules (N), Locations (L), Interfaces (I), QoP policies (QoP) and QoS options (QoS), respectively refereed to as $Ker_N, Ker_L, Ker_I, Ker_{QoP}$ and Ker_{QoS}. Each sort has its own constants and function symbols. We use five constant symbols to denote *undefined values* $\perp_N, \perp_L, \perp_I, \perp_{QoP}$ and $\perp_{QoS}$ to model partial functions.

In addition, we use sets created with the $\emptyset$, such as $\{\emptyset, \{\emptyset\}\}$ to model the structure of any well-founded finitely branching tree. As will be seen shortly, we use such nested sets to limit the recursive backtracking through rule chains. We also use nested sets to code integers. For example $\{\{\ldots\{\emptyset\}\ldots\}\}$ where the empty set $\emptyset$ is embedded in `n` braces is used to represent the integer `n`.

The rest of the paper uses names starting with an upper case letter for variables and names starting with a lower case letter for constants, and sometimes use a name with a *hat above* (for example $\widehat{ReqM}$) to describe a term with variables and constants, when it is too long the write out the details within a rule. We now describe the other parts of our language.

`module(Name,Location,Req,Prov,Depth)`: is a 5-ary predicate where the `Name` is a constant or variable from Set_N representing a name. Similarly, `Location` is a constant or a variable from Set_L, representing a location (or a set of locations). `Req` is a constant or a variable representing an ordered triple of the type $Set_I \times Set_{QoP} \times Sec_{QoS}$ used to model a set with elements of the form (interface, Set of QoP and QoS options) representing the collection of *requires* interfaces with their options and `Depth` (sometimes denoted as 'Z') is a set term used to encode the recursive depth. Similarly, `Prov` is a constant or variable from the same type as that of `Req`, representing the interfaces and the options of the *provides* interfaces of the component. In this version where we do not use recursive compositions, we use the value of $\emptyset$ for the `Depth` attribute.

`cModule(Name,Location,Req,Prov,Depth)`:] is a 5-ary predicate with the same type of predicates as the `module` predicate, representing a composed module. In this version where we do not use recursive compositions, we use the value of $\emptyset$ for the `Depth` attribute.

`composable2(Name1,Location1,Req1,Prov1, Name2,Location2,Req2,Prov2,Depth)`: is an 9-ary predicate where `(Name1,Location1,Req1,Prov1)`, `(Name2,Location2,Req2,Prov2)` are the two components, and the *provides* interfaces of the second are connected to the *requires* interfaces of the first and `Depth` is a set term used to encode the recursive depth. We also use another predicate `composable3(Name1,Location1, Req1,Prov1,Name2,Location2,Req2,Prov2, Name3,Location3,Req3,Prov3,Depth)` that composes three modules where the *provides* interfaces of the second and the third modules are connected to the *requires* interfaces of the first.

`structOK2(Name1,Location1,Req1,Prov1, Name2,Location2,Req2,Prov2,Depth)`: is an 9-ary predicate where `(Name1,Location1,Req1,Prov1)`, `(Name2,Location2,Req2,Prov2)` belong to two components and `Depth` is a set term used to encode the recursive depth. This predicate specifies the structural compatibility requirements between the *provider* and the

requester modules. Similarly, we use a 12-ary predicate `structOK3(Name1,Location1,Req1,Prov1, Name2,Location2,Req2,Prov2,Name3,, Location3,Req3,Prov3)` to model the structural integrity of a module composed from three sub-modules.

`qopOK2(Name1,Location1,Req1,Prov1, Name2,Location2,Req2,Prov2,Depth)`: is an 9-ary predicate where each of `(Name1,Location1,Req1,Prov1)` and `(Name2,Location2,Req2,Prov2)` belong to two modules. The predicate `qopOK2`is used to specify the security policy for module composition. Similarly, we use a 12-ary `qopOK3()` predicate for composing three modules.

`qosOK2(Name1,Location1,Req1,Prov1, Name2,Location2,Req2,Prov2, Depth)`: is an 9-ary predicate where `(Name1,Location1,Req1,Prov1)`, `(Name2,Location2,Req2,Prov2)` are parameters belonging to two modules and `Depth` is a set term used to encode the recursive depth. The predicate `qosOK2` is used to specify QoS policies of the composition. We also use a 12-ary predicate `qosOK3()` to specify QoS preferences for three modules.

`typeOK2(Name1,Location1,Req1,Prov1, Name2,Location2,Req2,Prov2,Depth)`: is an 9-ary predicate where all interfaces in `(Name1,Location1,Req1,Prov1)` match the respective interfaces of `(Name2,Location2, Req2,Prov2)` in type and `Depth` is a set term used to encode the recursive depth. Similarly, we use a 12-ary predicate `typeOK3()` to specify type compatibility of three modules.

2.1 Specifying Module Composition Policies

In this section we specify rules about module composition. Composition rules must satisfy some structural properties such as the requirement that one module provide the interfaces needed by the other module. First, we specify the structural composition rules.

$composable2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z) \leftarrow$
$structOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_s)$,
$qopOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_p)$,
$qosOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_q)$
$Z_s, Z_p, Z_q \in Z$.

This rule defines structural compatibility using `structOK2()`, security policy specified using the predicate `qopOK2()` and QoS policy specified using the predicate `qosOK2()`. A similar definition can be given for `composable3()`.

2.2 Structural Composition Policies

Structural composability can be specified using the set constraint language and other user defined predicates as given in the following two examples:
$structOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z) \leftarrow$
$cModule(N1, L1, Re1, Pr1, S_1)$,
$cModule(N2, L2, Re2, Pr2, S_2)$,
$typeOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, S_{type})$,
$Pr1 \cap Pr2 = \emptyset, Re1 \cap Re2 = \emptyset, Re1 \subseteq Pr2$,
$S_1, S_2, S_{type} \in S$.

This example says that two modules $N1$ and $N2$ are structurally compassable iff they do not share any *provides* interfaces or *requires* interfaces and the *requires* interfaces of the first module $N1$ are provided by the second module $N2$. By adding an extra clause $Re1 \subseteq Pr2$ to the last line of the policy, we can allow partial compositions where we the second module $N2$ can provides more interfaces than those required by $N1$. Also by reversing the constraint $Re1 \subseteq Pr2$ to $Re1 \supseteq Pr2$, we can accept *partial* compositions where the second module is unable to provide all required interfaces. By doing so, we can partially compose $N1$ and $N2$ and then compose this intermediate result with another module $N3$ to obtain a different definition of composition showing the flexibility of our design. The condition $S_1, S_2, S_{type} \in S$ is used to ensure the termination of queries.

3 QoP Policies

As mentioned, our sample QoP policies either control access to modules or information that flow between them. We modify *Attribute-based Access Control* [54] of Wang et al. for the former and *FlexFlow* [12] of Chen et al. for the latter. These policies have their own languages and are evaluated separately. Their evaluations are then imported into the module composition framework by using two kinds of predicates.

Flow Control: `secureFlow2(Sub1,Role1, Org1,N1,L1,Req1,Prov1,Sub2,Role2, Org2,N2,L2,Req2,Prov2,Z)`, say that information is allowed to flow between modules `N1` and `N2` when they are composed. A similar predicate `secureFlow3(Sub1,Role1,Org1,N1, L1,Req1,Prov1,Sub2,Role2,Org2,N2,L2, Req2,Prov2, Sub3,Role3,Org3,N3,L3, Req3,Prov3,Z)` is used to exercise flow control between the modules `N1`, `N2` and `N3`. The last variable `Z` is used to ensure that recursion terminates.

Access Control: `do(Sub,Role,Org,Loc,Mod, Int,Z)` says that a subject `Sub` playing role `Role` belonging to organization `Org` at location `Loc` can execute

interfaces `Int` belonging to module `Mod` and the last variable `Z` is used to ensure that recursion terminates.

Using the above two predicates exported from the global flow control policy specification module and the two local policy specification modules at locations $L1$ and $L2$ are related to $qopOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z)$ by the following rule.

$qopOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z) \leftarrow$
$secureFlow2(Sub1, Role1, Org1, N1, L1, Re1, Pr1,$
$Sub2, Role2, Org2, N2, L2, Re2, Pr2, Z_f),$
$do_{L1}(Sub1, Role1, Org1, L1, N1, Pr1 \cup Re1, Z_1)),$
$do_{L2}(Sub2, Role2, Org2, L2, N2, Pr2 \cup Re2, Z_2)),$
$Z_f, Z_1, Z_2 \in Z.$

As described, in the above rule, the requester-provider policy is considered secure if the flow is permitted by flow control predicate $secureFlow2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_f)$, and the two access control predicates $do_{L1}(Sub1, Role1, Org1, L1, N1, Pr1 \cup Re1, Z_1)$ and $do_{L2}(Sub2, Role2, Org2, L2, N2, Pr2 \cup Re2, Z_2)$ allow subject $Sub1$ to execute $N1$'s interfaces and subject $Sub2$ to execute $N2$'s interfaces respectively.

3.1 The Flow Control Sub-language

In order specify flow control policies, we add three more sorts, Sub for subjects and and $Role$ for roles and Org for organizations, and use the following predicates:

Predicates in Stratum 1: We use two binary predicates, `owner(s,m,∅)` with $m \in Names$ and $s \in Sub$ where `owner(s,m,∅)` is true if module m is owned by subject s. Similarly, `playRole(s,r)` where $s \in Sub$ and $r \in Role$ says that subject s performs the role (i.e. job function or has a military rank) r. We also use a third predicate `organization(s,org,∅)`, saying that the subject `s` works for the organization `org`.

Predicates in Stratum 2: We use two 14-ary predicates and two 21-ary predicates `canFlow2(Sub1,Role1,Org1,N1,L1,Req1, Prov1,Sub2,Role2,Org2,L2,N2,Req2, Prov2,Z)`, and `cannnotFlow2(Sub1,Role1, Org1, N1,L1,Req1,Prov1,Sub2,Role2, Org2,N2,L2,Req2,Prov2,Z)`. Similar definitions exists for predicates `canFlow3()` and `cannotFlow3()` used for ternary compositions.

Predicates in Stratum 3: Consists of two predicates `canFlow2*(Sub1,Role1,Org1, N1,L1,Req1,Prov1,Sub2,Role2,Org2,L2, N2,Req2,Prov2,Z)`, and `cannnotFlow2*( Sub1,Role1,Org1,N1,L1,Req1,Prov1,Sub2, Role2,Org2,N2,L2,Req2,Prov2,Z)`, that are used to recursively specify flow control permissions and prohibitions.

Predicates in Stratum 4: Consists of two predicate `secureFlow2(Sub1,Role1,Org1,N1,L1, Req1,Prov1, Sub2,Role2,Org2,N2,L2, Req2,Prov2,Z)` and `secureFlow3(Sub1, Role1,Org1,N1,L1,Req1,Prov1,Sub2, Role2,Org2,N2,L2,Req2,Prov2,Sub3, Role3,Org3,N3,L3,Req3,Prov3,Z)` that evaluates the final decision to allow information flow.

A flow control policy is a finite collection of rules constructed according to the following constraints:

Rules in Stratum 1: Rules specify basic facts such as the roles played by modules and modules belonging to organizations etc. Hence they should have a head predicate belonging to Stratum 1 and an empty body, and therefore should be of the following form, $owner(s, m, \emptyset) \leftarrow$, $playRole(s, r, \emptyset) \leftarrow$ and $organization(s, org, \emptyset) \leftarrow$.

Rules in Stratum 2: The head predicate must be one of `canFlow2()`, `cannotFlow2()`, `canFlow3()` or `cannotFlow3()` and the bodies may contain predicates from stratum 1 and constraints from `CLP(Set)`. An example rule is as follows:

$cannotFlow2(Sub1, Role1, Org1, N1, L1, Req1, Prov1,$
$Sub2, Role2, Org2, N2, L2, Req2, Prov2, Z) \leftarrow$
$organization(L1, O_1, Z_1),$
$organization(L2, O_2, Z_1),$
$securityLevel(O_1, S_1, Z_2),$
$securityLevel(O_2, S_2, Z_3),$
$S_2 < S_1, Z_1, Z_2, Z_3 \in Z.$

Rules in Stratum 3: The head predicate of a rule must have one of `canFlow2*()`, `cannotFlow2*()`, `canFlow3*()`, `cannotFlow3*()` and the bodies may contain predicates from strata 1 or 2, constrains from `CLP(Set)`, or stratum 3 predicates that appear positively, as shown in the following example.

$canFlow2*(Sub1, Role1, Org1, N1, L1, Req1,$
$Prov1, Sub3, Role3, Org3, N3, L3, Req3, Prov3, Z) \leftarrow$
$canFlow2(Sub1, Role1, Org1, N1, L1, Req1,$
$Prov1, Sub2, Role2, Org2, N2, L2, Req2, Prov2, Z_1),$
$canFlow2*(Sub2, Role2, Org2, N2, L2, Req2,$
$Prov2, Sub3, Role3, Org3, N3, L3, Req3, Prov3, Z_2)$
$Z_1, Z_2 \in Z.$

Rules in Stratum 4: This stratum contains the following rule:

$secureFlow2(Sub1, Role1, Org1, N1, L1, Req1,$
$Prov1, Sub2, Role2, Org2, N2, L2, Req2, Prov2, Z) \leftarrow$
$canFlow2*(Sub1, Role1, Org1, N1, L1, Req1$
$Prov1, Sub2, Role2, Org2, N2, L2, Req2, Prov2, Z_1),$
$\neg cannotFlow2*(Sub1, Role1, Org1, N1, L1, Req1,$
$Prov1, Sub2, Role2, Org2, N2, L2, Req2, Prov2, Z_2)$
$Z_1, Z_2 \in Z.$

FlexFlow [12] shows that this type of a rule-base is locally stratified, and consequently returns an answer for every query. As already stated, we export predicates `secureFlow2()` and `secureFlow3()` from the flow-control sub-language to our security policy

specification language. Therefore, some variables need to be shared between the two sub-policy frameworks. Section 6 shows how to use the flow control sub language in enforcing security policies for module composition.

3.2 The Access Control Sub-language

Discretionary access control polices that are stated in terms of `(Subject,Object,Access-method)` needs to be replaced with sextuple `(Subject,Role,Organization,Module Name, Location, Interfaces)` in order to specify module composition policies. We do so by using the following predicates, appropriately adopted from [54].

Predicates in Stratum 1: Has predicates to represent basic facts such as ownerships, subject-role assignments, object and subject hierarchies etc.

Predicates in Stratum 2: `cando(Subject, Role,Organization,Module Name,Location, Interfaces,Z)` and `cannotdo(Subject, Role,Organization, Module Name, Location,Interfaces,Z)` state which module options are permitted or prohibited from executing. Stated attributes represents Subjects, Roles played by the subjects, organizations that roles and subjects belong to, module name and interfaces with their security and QoS options.

Predicates in Stratum 3: `cando*(Subject, Role,Organization,Module Name,Location, Interfaces,Z)` and `cannotdo*(Subject, Role,Organization,Module Name, Location,Interfaces)` are two predicates used to recursively extend the definitions of `cando()` and `cannotdo()`.

Predicates in Stratum 4: `do(Subject,Role, Organization,Module Name,Location, Interfaces,Z)` expresses the final decision about a module being able to execute with specified options.

Our access control policies are constructed using reserved predicates and possibly other application specific predicates using the following stratification:

Rules in Stratum 1: Specify basic relationships and application specific facts written as predicate instances (i.e. rules with empty bodies). Some examples are $owner(Subject, Module, \emptyset) \leftarrow$, $role(Subject, Role, \emptyset) \leftarrow$, and $organization(Subject, Org, emptyset) \leftarrow$.

Rules in Stratum 2: Rules using `cando` and `cannotdo` heads must be of the form $cando(X, Y, Z, W, U, V, Z) \leftarrow B$ or $cannotdo(X, Y, Z, W, U, V, Z) \leftarrow B$ where the body B may have predicates from lower strata and constraint expressions. These are used to state basic facts about granting/denying access to services.

Rules in Stratum 3: Rules with `cando*` or `cannotdo*` heads can have have `cando*`, `cannotdo*` predicates in their bodies only positively, but may have `cando` and other non-reserved predicates and constraint terms.

Rules in Stratum 4: $do(X, Y, Z, W, U, V, Z) \leftarrow cando * (X, Y, Z, W, U, V, Z_+), \neg cannotdo * (X, Y, Z, W, U, V, Z_-)$, $Z_+, Z_- \in Z$ is the only rule at this stratum.

Any finite collection of rules conforming to constraints (1) through (4) is said to be an access control policy. Wang et al. [54] provides a fixed point semantics for similar access control policies. But their use in the module composition framework is limited to being an exported predicate, and therefore taken as true, iff the instance is exported.

4 QoS Policies

In QoS policies, we represent total resources requirements (of a module and inter-module communication) as a multiset (i.e. a bag) where every resource unit of a given type is represented by an element in the bag. For example, if a mobile phone has 2 CPU threads and 3 units of buffer space (say in Kilo bytes) available for applications, total resources available are represented as $\{|\ cpu, cpu, buf, buf, buf\ |\}$, where the symbols in between the braces $\{|$ and $|\}$ are the (repeated) elements of the resource *bag*. We decide if a module to be used has sufficient resources to execute iff its resources can be packed inside the resource bag offered by the hosting platform. For example, to verify that the hardware platform of a mobile phone with resources $\{|\ cpu, cpu, buf, buf, buf\ |\}$ can accommodate an application with an estimated resource requirement (multiset) U, we need to check if U is a (multi) subset of $\{|\ cpu, cpu, buf, buf, buf\ |\}$. With this representation, we propose a three level stratified logic programming language with multi-set constraints as follows:

Predicates in Stratum 1: This stratum specifies service dependencies. The predicate $needs(S, T)$ says that the (subsidiary) set of services T are required to provide the set of (primary) services S. $allNeeds(S, T)$ says that the set of services T consists of all subsidiary services required to provide the set of primary services S.

Predicates in Stratum 2: This stratum computes resource requirements for a given set of services with all of its QoP and QoS options. In order to express the resources needed to communicate, two predicates $comRes(N1, L1, Pr1,\ \ Re1, N2, L2, Re2, Pr2, R, Z)$

and $comRes*(N1, L1, Re1, Pr1, N2, L2, Pr2, R, Z)$ are used. The first one says that R resources are required to communicate between modules $N1$ and $N2$. The second predicate $comRes*(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, R, Z)$ is used to recursively compute resource needs of dependent services. Finally, the predicate $allComRes(N1, L1, Re1, Pr1, N2, L2, Pr2, Re2, R, Z)$ says that the total amount of resources available to be consumed by all software modules is R.

Similarly, to compute the resource needs for local platforms, we use two predicate $localRes(N1, L1, Pr1, R, Z)$ and $localRes*(N1, L1, Re1, Pr1, R, Z)$ where $(N1, L1, Re1, Pr1)$ requires resources R. $localRes(N1, L1, Re1, Pr1, R)$ say that R resources are needed to service $(N1, L1, Re1, Pr1)$, and $localRes*(N1, L1, Pr1, Re1, R, Z)$ is used in recursive rules that compute the resources needed to execute all dependent services. $allLocalRes(N1, L1, Re1, Pr1, R, Z)$ says that the total resources requirement to service the module $(N1, L1, Re1, Pr1, Z)$ is R.

Predicates in Stratum 3: This level computes resource availability on local platforms and communication links. In order to do so we use two predicates $localLimit(L1, R1)$ and $commLimit(L1, L2, R2)$ which respectively says that module $L1$ has $R1$ amount of resources to execute and the total amount of resources required to communicate between locations $L1$ and $L2$ is $R2$.

Predicates in Stratum 4: This level renders the final decision to allow a module to execute on a host, and connectable modules to connect to each other. In order to indicate so, we use two predicates, $comResOK(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, R, Z)$ and $localResOK(N1, L1, Re1, Pr1, R, Z)$, saying that with R resources, the module $N1$ can execute on its proposed host and modules $N1$ and $N2$ can be connected to each other, respectively.

Predicates listed above can be used in rules of the following kinds.

Rules in Stratum 1: Rules at this stratum are of the forms $P(X1, \ldots, Xn, Z) \leftarrow$, $needs(X, Y, \emptyset) \leftarrow$ and $allNeeds(X, Y, Z) \leftarrow Body$, where $P(X1, \ldots, Xn, Z)$ is any application dependent predicate that encodes facts. $needs(X, Y)$ say that service X depends on services Y, and consequently, in order to have X, resources must also be provided for Y. The Body of the last rule may contain application dependent predicates, `depends`, positive occurrences of `allNeeds` multiset constraints and $Z_1, \ldots, Z_n \in Z$ where $Z_1, \ldots, Z_n$ are the set variables that occur as the last parameter in predicates used in the body of the rule.

Rules in Stratum 2: Rules at this stratum recursively define `comRes*()` and `localRes*()` using positive occupancies of themselves, `comRes()`, `localRes()` and (multi)set constraints respectively. Consequently, B may have any predicate from stratum 1, but bodies B and D should not have `comRes()` and `localRes()` appearing negatively. `allComRes()` and `allLocalRes()` collect all resources needed to communicate between sites $L1$ and $L2$ and at site $L1$ respectively. Bodies C and E may have any other predicates belonging to lower strata. In the following four rules, $Z_1, \ldots, Z_n$ are the set variables that occur as the last parameter in predicates used in the bodies of the rule B, C, D and E respectively.

$comRes*(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, R, Z)$
$\leftarrow B, Z_1, \ldots, Z_n \in Z.$
$allComRes(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, R, Z)$
$\leftarrow comRes*(N1, L1, Re1, Pr1, N2, L2, Pr2, Re2, R, Z), C, Z_1, , \ldots, Z_n \in Z.$
$localRes*(N1, L1, Re1, Pr1, R, Z)$
$\leftarrow D, Z_1, \ldots, Z_n \in Z.$
$allLocalRes(N1, L1, Re1, Pr1, R, Z)) \leftarrow$
$localRes*(N1, L1, Re1, Pr1, R, Z), E, Z_1, \ldots, Z_n \in Z.$

Rules in Stratum 3: Rules in this stratum may have $localLimit(L1, R, Z)$ or $commLimit(L1, L2, R, Z)$ as heads and (set and multi-set) constraints, but their bodies may not have predicates from stratum 2. They are used to specify resource limitations on the communication channels and local sites.

Rules in Stratum 4: Rules at this stratum has $comResOK(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z)$ or $localResOK(N1, L1, Re1, Pr1, Z)$ heads and bodies consisting of predicates from lower strata with the usual $Z_1, \ldots, Z_n \in Z$. These predicates say that QoS requirements are satisfied for communication and local computations respectively.

We use these predicates to compute the total resource utility of a module and to specify the policy of deciding if the modules are to be composed. For example, one could use an optimistic policy of composing modules if they consume about 110% of the total available resources (such as in air-line seat reservations). Conversely, a pessimistic policy would not commit more than 60% of the total available resources. More complex policies can be composed based on better resource estimates. The predicates at Stratum 4 are related to exported QoS policies using a rule of the following kind, where `qosOK2` or `qosOK3()` are defined using `localResOK()` and `comResOK()`.

$qoSOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z)$
$\leftarrow localResOK(N1, L1, Re1, Pr1, Z_1),$
$localResOK(N2, L2, Re2, Pr2, Z_2),$
$comResOK3(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_c),$
$Z_1, Z_2, Z_c \in Z.$

This rule says that the QoS requirements are satisfied iff the local computing requirements and communication requirements are satisfied. A similar definition can be given for `qosOK3()`.

5 Semantics

This section describes models of our module composition syntax and their policies. Taken as a constraint logic program, our syntax has a three valued Kripke-Kleene model [36, 22] where every predicate instance evaluates to one of three truth values *true, false* or *undefined*. We will shortly show that every query (a request) will evaluate to either *true* or *false*, and therefore has only two truth values - ensuring that every module composition request is either granted or denied. Because we allow nested negative predicates, we need to interpret *negation*. We can either use negation as failure or *constructive* negation [10, 11] as proposed by Fages [20, 19]. This is because the third alternative namely using constructive negation as proposed by Stuckey [49, 50] requires that the constraint domain be *admissibly closed*. But Dovier shows that set constraints as we use them are not admissibly closed, and proposes an alternative formulation to handle nested negations [18]. Conversely, at the cost of requiring some uniformity in computing negated subgoals of a computation tree, Fages's formulation does not require the constraint domain to be admissible closed [20, 19]. Formalities follow. We first repeat some standard definitions as they appear in [23] in order to clarify notation.

Definition 1 (P^*, T_P and $\Phi_P \uparrow$ operators) *Suppose that P is a logic program, and let P^* be all ground instances of atoms in P. We take $A\leftarrow$ as A and any ground atom A not in the head of any rule as $A\leftarrow$`false`. We now define two and three valued truth lattices to be $\mathbf{2} = \langle\{T, F\}, <_{\mathbf{2}}\rangle$ and $\mathbf{3} = \langle\{T, F, \bot\}, <_{\mathbf{3}}\rangle$ respectively, where T, F and $\bot$ are taken to mean* `true`, `false` *and* `unknown` *truth values. Partial orderings $<_{\mathbf{2}}$ and $<_{\mathbf{3}}$ satisfy the conditions $F <_{\mathbf{2}} T$ and $\bot <_{\mathbf{3}} T, \bot <_{\mathbf{3}} F$ respectively. A mapping V from instantiated clauses of P to $\mathbf{2}$ and $\mathbf{3}$ is said to be respectively a two-valued or a three-valued valuation of P. Given a valuation V, the two and three valued immediate consequence operators $T_P(V)$ and $\Phi_P(V)$ are defined as follows.*
$\mathbf{T_P(V)} : T_P(V) = W$ *is defined as:*

- $W(H) = T$ *if there is a ground clause* $H\leftarrow B_1, \ldots, B_n$ *in* P^* *such that* $V(B_i) = T$ *for* $i \leq n$.
- $W(H) = F$ *otherwise.*

$\mathbf{\Phi_P(V)} : \Phi_P(V) = W$ *is defined as:*

- $W(H) = T$ *if there is a ground clause* $H\leftarrow B_1, \ldots, B_n$ *in* P^* *such that* $V(B_i) = T$ *for* $i \leq n$.
- $W(H) = F$ *if for every ground clause* $H\leftarrow B_1, \ldots, B_n$ *in* P^* *such that* $V(B_i) = F$ *for some* $i \leq n$.
- $W(H) = \bot$ *otherwise.*

In evaluating Φ, negation is interpreted as $\neg T = F, \neg F = T$ and $\neg\bot = \bot$. Now we define bottom-up semantic operators for both T_P and Φ_P, where Ψ stand for either of them in the following.

- $\Psi^0 \uparrow (P) = V_{false}$*, where* V_{false} *assigns* F *(false) to all instantiated atom.*
- $\Psi^{\alpha+1} \uparrow (P) = \Psi(\Psi^\alpha \uparrow (P))$ *for every successor ordinal* α.
- $\Psi^\alpha \uparrow (P) = \bigvee_{\beta<\alpha}(\Psi^\beta \uparrow (P))$ *for every limit ordinal* α.

For Horn clauses (i.e. those without negative non-constraint predicates in the body) $T(P)$ has a least fixed point, that is considered the model theoretic semantics of P [23] that is $T_\omega(P)$. But for three-valued semantics, the least fixed point may not be obtained at ordinal ω. But following standard practice we take $\Phi_\omega(P)$ as the *meaning* (i.e. semantics) of our combined rule base (i.e. composition rules + QoP policies + QoS Policies) $\mathcal{P}$ as formalized in definition 2.

Definition 2 (bottom-up semantics) *Let $\mathcal{P}$ be a policy and Φ be the three-valued immediate consequence operator stated in definition 1. Then we say that $\bigcup_{i\in\omega} \Phi^i(\mathcal{P})$ is the model of $\mathcal{P}$.*

Definition 2 says that we obtain a model of $\mathcal{P}$ by evaluating the Φ operator ω many times. As promised, we now show that $\bigcup_{i\in\omega} \Phi^i(\mathcal{P})$ only takes two truth values. In order to do so, we consider a version of the standard operational semantics for constraint logic programs. Thereafter by defining a *rank* for a formula so that the rank decreases as one proceeds from the root towards the leaves of a top down computation tree, we show that every computation terminates. The property we use here is the well-foundedness of the membership predicate built into some of our predicates (namely, those that would interleave recursion and negation). In order to do so, we now repeat (a version of) operational semantics proposed for constraint logic programs [33, 35].

Definition 3 (operational semantics) *A state is a pair (A, C) of multisets of predicates A and constraints C. Let $\mathcal{P}$ be an ABAC policy and (A, C) (A', C') are states. We say that:*

- $(A \cup \{p(\vec{s})\}, C) \rightarrow_1 (A \cup B, C \cup C" \cup \{\vec{s} = \vec{t}\})$ *is a one-step derivation provided $p(\vec{t}) \leftarrow B, C"$ is a rule in $\mathcal{P}$ and $p(\vec{s}) \leftarrow B, C"$ is a renamed apart instance of $p(\vec{s}) \leftarrow B, C"$.*
- *We say that (A, C) fails if $A \neq \emptyset$ and there is no predicate $p \in A$ where $p(\vec{t}) \leftarrow B, C"$ is a rule in $\mathcal{P}$.*
- *We say that (A, C) is successful if $(A, C) \rightarrow_* (\emptyset, C')$ for some constraint set C' satisfiable by an assignment σ of variables to values, where $\rightarrow_*$ is the reflexive transitive closure of $\rightarrow_1$.*
- *A query (A, C) is said to flounder if it neither successful nor fails.*

The third clause of definition 3 usually reads as (A, C) is said to be successful if $(A, C) \rightarrow_* (\emptyset, C')$ for some consistent constraint set C'. But Dovier et al. shows that in the computable set theory we use, a set of constraints C' is consistent iff it is satisfiable by some assignment of variables to values [15, 16, 17]. Coincidentally, the operational semantics given by definition 3 and the fixed point semantics given by definition 2 coincide [33, 35]. We now proceed to show that our policies do not flounder.

Definition 4 (rank) *We say that the rank of a well-founded set is the maximum nesting of braces in it, formally defined $rank(s)$ to be max$\{1+rank(u), rank(v)\}$ when s is of the form $\{u \mid v\}$ and as 0 otherwise. We say that the rank of an instance of a reserved predicate is the rank of its last variable, and the rank of a rule instance is the rank of its head predicate.*

Suppose A is a finite multiset of predicates, and $\{a_1, \ldots, a_n\}$ lists elements of A in the decreasing order of their ranks. That is, they satisfy the condition that $i > j \rightarrow R(a_i) \geq R(a_j)$, where $R(a_j)$ is the rank of a_j. Suppose m is the highest rank in A. That is, m =max$\{R(a) : a \in A\}$, and let $\mathcal{C}(A, i) = |\{R(a_j) : R(a_j) = i\}|$ for every $i \leq m$. That is, $\mathcal{C}(A, i)$ is the number of predicates with rank i. Then define $R(A)$, the rank of the multiset A as the vector $(\mathcal{C}(A, m), \ldots, \mathcal{C}(A, 0))$. We order multiset ranks (that is $(\mathcal{C}(A, m), \ldots, \mathcal{C}(A, 0))$) lexicographically.

Definition 4 defines the ranks for reserved predicates, rules and multisets of predicates. Because the multiset ranks are finite sequences of non-negative integers. We use the fact that finite sequences of non-negative integers are well-ordered in Lemma 1 to show that any application of any of our rules reduces the rank of the *rule state*, and therefore must terminate finitely.

Lemma 1 (miscellaneous properties of ranks) *Suppose $h \leftarrow B$ is a derivation rule where the last attribute is fully instantiated (i.e. variable free). Then $R(h) > R(b)$ for any reserved predicate b in the body B. Furthermore, if $(A \cup \{p(\vec{s})\}, C) \rightarrow_1 (A \cup B, C \cup C" \cup \{\vec{s} = \vec{t}\})$ is a one-step derivation where $p(\vec{t}) \leftarrow B, C"$ is a rule in $\mathcal{P}$ and $p(\vec{s}) \leftarrow B, C"$ is a named apart instance of $p(\vec{s}) \leftarrow B, C"$. Then $R(A \cup \{p(\vec{s})\}) > R((A \cup B)$.*

Proof:

Case 1: Predicates of the Access Control Sub language

To prove the first claim, according to policy definition, the reserved predicates are `cando cando*, cannotdo, cannotdo*` and `do`. We consider each of them now.

`cando, cannotdo`: $cando(-, -, -, \ldots, \{\emptyset\}) \leftarrow B$ where B consists of non-reserved predicates or is empty. This is the only allowed form of `cando` in a rule head. Thus, R($cando(-, -, -, \ldots, \{\emptyset\})) = 1$ and R(b) = 0 for any predicate b in B. A similar argument holds for `cannotdo`.

`cando*` and `cannotdo*`: According to the third rule in the policy definition, if $cando*(-, -, -, \ldots, Z)$ is in the body and $cando*(-, -, -, \ldots, Z')$ is in the body, then $Z = \{Z'|U\}$ for some set U. Hence by definition 4, $R(cando*(-, -, -, \ldots, Z)) \geq 1 + R(cando*(-, -, -, Z'))$.

`do`: The same argument applies for rules with a $cando*(-, -, \ldots, Z)$ head. The only rule with a $do(-, -, \ldots, Z)$ head is $do(X, Y, -, \{Z\}) \leftarrow cando*(-, -, \ldots, Z_1), \neg cannotdo*(-, -, \ldots, Z_2)$. Thus, $R(do(-, -, \ldots, \{Z\})) = 1 + max\{R(cando*(-, -, \ldots, Z_1)), R(cannotdo*(-, -, \ldots, Z_2))\}$.

Case 2: Predicates of the Flow Control Sub Language

This case is similar to that of Case 1, due to the similarity of the rules at the same starts.

Case 3: Predicates of the QoS Policy Sub Language

This case is similar to that of Case 1, due to the similarity of the rules at the same starts.

Case 4: Predicates for Composition

For predicates `module, cModule, typeOK2, typeOK3, structOK2` and `structOK3` are base predicates, where the last attribute is $\emptyset$. Hence they have Rank 0, and therefore do not flounder.

For the predicate `composable2`, the prescribed rule of usage is as follows.

$composable2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z) \leftarrow structOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_s), qopOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_p),$

$qosOK2(N1, L1, Re1, Pr1, N2, L2, Re2, Pr2, Z_q)$, $Z_s, Z_p, Z_q \in Z$.

Hence the rank of any instance of `composable2` is `max {structOK2, qopOK2, qosOK2}+1`. A similar argument holds for `composable3`. ■

Now we use the first claim to justify the second. Suppose that $(A \cup \{p(\vec{s})\}, C) \rightarrow_1 (A \cup B, C \cup C" \cup \{\vec{s} = \vec{t}\})$ is a one-step derivation using the ABAC rule $p(\vec{t}) \leftarrow B, C"$ is a rule in $\mathcal{P}$ and $p(\vec{s}) \leftarrow B, C"$ is a fresh instance of $p(\vec{s}) \leftarrow B, C"$. To prove that $R(A \cup \{p(\vec{s})\}) > R(A \cup B)$, suppose $R(A \cup \{p(\vec{s})\}) = (k_m, \ldots, k_0)$, $R(P) = t < m$, and $R(B) = (t_u, \ldots, t_0)$ where $u < t$. Then $R(A \cup B) = (k_m, \ldots, k_{t+1}, k_t - 1, k_t, \ldots, k_i + t_i, \ldots, k_0)$. In the lexicographical ordering, $(k_m, \ldots, k_0) > (k_m, \ldots, k_{t+1}, k_t - 1, k_t, \ldots, k_i + t_i, \ldots, k_0)$, implying $R(A \cup \{p(\vec{s})\}) > R(A \cup B)$.

■

We now use Lemma 1 to show that composition queries terminate.

Theorem 1 (finite termination of queries) *Every query (A, C) either fails or succeeds, where A is a reserved predicate with a fully instantiated first attribute.*

Proof:

Suppose $(A, C) \rightarrow_1 (A_0, C_0) \rightarrow_1 (A_1, C_1), \ldots$ is an infinite sequence of one-step reductions. Then by lemma 1, $R(A, C) > R(A_0, C_0) > \ldots$ is an infinite descending sequence, contradicting the well-foundedness of the rank function. This is a contradiction, as the lexicographical ordering on integers is well-founded. ■As a corollary, we now obtain that any query always gives a *yes* or *no* answer, implying that following theorem which says that all three valued models have only two truth values *true* and *false*.

Corollary 1 (three valued model has two truth values) *Every three valued model of a composition policy assigns either T or F for reserved predicates where the first attribute is instantiated. In that case, bottom-up semantics and the well-founded constructions assigns the same truth values to the same predicate instances and have the same answer sets.*

Proof: See [7]. ■

Corollary 1 show that every composition request is either honored or rejected. But notice that our model is not a fixed point of the Φ operator , as it is well known that the least fixed point of the Φ operator is not ω - meaning that the fixed point of Φ is not attained in a countable number of iteration. Φ [24, 19]. ([24] gives a simple counter example)

6 An Example

This section shows an example application of our framework to a conference call manager. As shown in the schematic of Figure 6, the *talking phase* of the teleconferencing module is constructed from three existing modules, a `manager` located at the `pentagon`, a `land phone` located at `ftMeade` and a `mobile phone` located in `iraq` . The *provides* interfaces of the `manager` is used by the conferees and the *requires* interfaces are connected to the *provides* interfaces of the `land phone` and `mobile phone` modules. The *requires* interfaces of these modules are used by the conferees at their respective sites.

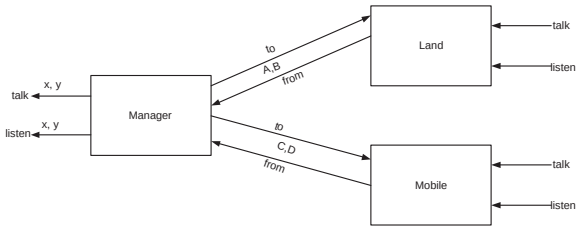

Figure 2: The Composed Teleconferencing Module

The call manager has four interfaces: two dedicated per conferee where the first interface is used to send multimedia streams consisting of some combination of audio, video and textual data and the second interface is used to receive the same data from the other two conferees. Hence we model the conference manager as $module(manager, pentagon, \widehat{ReqM}, \widehat{ProvM})$ where $\widehat{ReqM}$ is {(toLand,A,B),(fromLand,A,B), (toMobile,C,D),(fromMobile,C,D)} and $\widehat{ProvM}$ is {(talkM,X,Y),(listenM,X,Y)}. The `manager` module is specified using the following rule:
$cModule3(manager, L, \{(toLand, A, B),$
$(fromLand, A, B), (toMobile, C, D), \{fomMobile,$
$C, D\}, \{(talkM, X, Y), (listenM, X, Y)\}, \emptyset)$
$\leftarrow \{des\} \subseteq X \subseteq \{des, 3des, aes\},$
$\{audio\} \subseteq Y \subseteq \{audio, video, text\},$
$\{3des\} \subseteq A \subseteq \{des, 3des, aes\},$
$\{des\} \subseteq C\{des, 3des, aes\}, \{audio, text\} \subseteq B,$
$D \subseteq \{audio, video, text\}, X = A \cup C, \ Y = B \cup D.$

The head predicate `cModule3()` of the above rule says that the module `manager`, located at L, has two *provides* interfaces `talkM` and `listenM`, has shared security and QoS parameter sets X and Y. The head predicate also says that this module has four *requires* interfaces: `toLand`, `fromLand`, `toMobile` and `fromMobile`, where the first two are to be connected to a land line with respective QoP and QoS parameter sets A and B. The latter two interfaces can be

connected to a mobile phone with QoP and QoS parameter sets C and D. The design constraints of the `manager` module appears in the body of the rule, and are as follows: The QoS parameters of the *provides* interfaces `talkM` and `listenM` must be the same (because they are given as (talkM,X,Y) and (listenM,X,Y)) and contain `des` and is contained in `des,3des,aes`, specified as $\{des\} \subseteq X \subseteq \{des, 3des, aes\}$ in the fourth line of the rule. Similarly, QoS parameters must contain `audio` and must be chosen from `audio, video` and `text`, stated in the sixth line as $\{audio\} \subseteq Y \subseteq \{audio, video, text\}$. Similarly, the land line must provide at least `3des` encryption and could include `des` or `aes`, as stated in the seventh line as $\{3des\} \subseteq A \subseteq \{des, 3des, aes\}$. Similarly, the land line must provide at least `audio` and `text` and may include `video`, stated as $\{audio, text\} \subseteq B, E \subseteq \{audio, video, text\}$ in the eighth line. The last line of the rule, $X = A \cup C, Y = B \cup D$ says that the *provides* interface of the `manager` module must match all the security and QoS options available to its `requires` interfaces.

The Land Conferee's module is modled as $module(land, ftMeade, ReqLand, ProvLand, \emptyset)$ where $\widehat{ReqLand}$ is {(talkLand,A,B),{listenLand,A,B})} and $\widehat{ProvLand}$ is {(toLand,A,B),(fromLand,A,B)} defined using the following rule. $module(land, L, \{(talkLand, A, B), (listenLand, A, B)\}$ $\{(toLand, A, B), (fromLand, A, B\}, \emptyset \leftarrow$ $\{des\} \subseteq A \subseteq \{des, 3des, aes\}, \{audio\} \subseteq B \subseteq \{audio, video, text\}$

The rule above shows that the `land` conference module, located at L, consists of two *provides* interfaces `toLand` and `fromLand` and two *requires* interfaces `talkLand` and `listenLand`. As the rule says, the security and QoS parameters, respectively given by set variables A and B are configurable subjected to the constraints $\{des\} \subseteq C \subseteq \{des, 3des, aes\}$ and $\{audio\} \subseteq D \subseteq \{audio, video, text\}$ in the second and third lines of the rule.

The Mobile Conferee's module is modeled as $module(mobile, iraq, \widehat{ReqMobile}, \widehat{ProvMobile}, \emptyset)$ where $\widehat{ProvMobile}$ is {(toMobile,C,D), (fromMobile,C,D)} and $\widehat{ReqMobile}$ is {(talkMobile,C,D), (listenMobile,C,D)} defined using the following rule.

$module(mobile, L, \{(toMobile, C, D), (fromMobile,$ $C, D)\}\{(talkMobile, C, D), (listenMobile, C, D)\}, \emptyset) \leftarrow$ $\{des\} \subseteq C \subseteq \{des, 3des, aes\}\{audio\} \subseteq D \subseteq \{audio, text\}$.

Above rule says that the `mobile` component at location L consists of two *provides* interfaces `toMobile` and `fromMobile` and two *requires* interfaces `talkMobile` and `listenMobile`, with QoP and QoS given by set variables C and D are configurable subjected to the constraints $\{des\} \subseteq C \subseteq \{des, 3des, aes\}$ and $\{audio\} \subseteq D \subseteq \{audio, video, text\}$ in the second and third lines of the rule. Now we specify the composition.

$cModule2(telecon, pentagon, \widehat{ReTcon}, \widehat{PrTCon}, Z)$
$\leftarrow composable3((manager, pentagon, \widehat{ReqM}, \widehat{ProvM}),$
$(land, ftMeade, \widehat{ReqLand}, \widehat{ProvLand}),$
$(mobile, iraq, \widehat{ReqMobile}, \widehat{ProvMobile}), \emptyset),$
$\widehat{PrTcon} = \widehat{PrM}, \widehat{ReTCon} = \widehat{ReqLand} \cup \widehat{ReqMobile}$

Above rule with a $cModule2()$ head says that location `pentagon` requires a module `telecon` that provides $\widehat{PrTcon}$ and requires capabilities $\widehat{ReTcon}$. The right hand side of the rule says that the `telecon` module that is constructed from three modules `manager, land` and `mobile`, located at the `pentagon`, `ftMeade` and `iraq` respectively. The *required* and *provided* interfaces of the composed module `telecon` can now be constructed from the last two lines of the rule above as follows:

$$\begin{aligned}
\widehat{PrTcon} &= \widehat{ProvM} \\
&= \{(talkM, X, Y), (listenM, X, Y)\}, \\
\widehat{ReTcon} &= \widehat{ReqLand} \cup \widehat{ReqMobile} \\
&= \{(talkLand, A, B), (listenLand, A, B), \\
&\quad (talkMobile, C, D), (listenMobile, C, D)\})
\end{aligned}$$

Subjected to the following constraints:

$$\begin{aligned}
\{des\} &\subseteq X, A, C \subseteq \{des, 3des, aes\} \\
\{audio\} &\subseteq Y, B, D \subseteq \{audio, video, text\} \\
X &= A \cup C \\
Y &= B \cup D
\end{aligned}$$

We now explain how the composition is written. Firstly, $\widehat{PrM}$ says that the composed module name `telecon` at the `pentagon` must provide the capabilities listed in $\widehat{PrTcon}$, requiring capabilities $\widehat{ReTcon}$. Notice that $\widehat{PrM}$ is a set with two ordered triples $(talkM, X, Y)$ and $(listenM, X, Y)$, which says that it may provide the ability to translate `audio` with possibly more QoS capabilities than `audio` and possibly more QoP capabilities than `des`. The specification says that the conference manager's site must have all QoP and QoS related services available for all mobile and land phone conferees. Notice that the security and QoS capabilities are specified using set variables, such as X, Y etc., that may be instantiated by the caller of the rule, and resolved by the constraint solver `CLP(Set)` during the composition process. Notice also that `composable3()` is specified using the structural rule `struct3()` as defined in Section 2.2.

We now show the QoP and QoS policies. In order to do so, we specify global cross-domain flow control policies that apply to the communication links spanning across all three sites and access control policies that regulate the accesses to the resources at each site. The facts about the modules, their users and the roles played by these users are stored at different locations as stated in Table 1. The table has site specific information for the four sites: the conference `manager`'s site `pentagon`, the `land` phone's site `ftMeade` and the `mobile` phone's site `iraq`. The square at the right hand bottom corner of Table 1 stores information about the teleconference module. There, we have stored multiple owners, a new role name `teleconferencing` and a non-physical location `global`.

We now state the flow control policies. The first rule places some constraints on information that can flow from the two modules `land` (telephone) and `mobile` (telephone) to the (teleconference) `manager` module simultaneously. They are (1) the manager's organization must be either the `pentagon` or `jtChiefs`, (2) the other two modules belongs either of `usArmy`, `usNavy` or the `usAF`, the manager of the teleconference must play the roles of a `commOfficer` (communications officer) or a `commander`. It further stipulates that the users of other modules must play the roles of `commOfficer`, `recon` (reconnaissance) or `signal` (officer) and prohibits the `mobile` module being located in `china`.

$canFlow3(Sub1, Role1, Org1, manager, L1, Req1,$,
$Prov1, Sub2, Role2, Org2, land, L2, Req2, Prov2,$
$Sub3, Role3, Org3, mobile, L3, Req3, Prov3, \emptyset)$
$\leftarrow Org1 \in \{pentagon, jtChiefs\}, Org2, Org3 \in \{usArmy, usNavy, usAF\}, L2 \neq china,$
$Role1 \in \{commOfficer, commander\},$
$Role2, Role3 \in \{commOficer, recon, signal\}.$

$canFlow3*(Sub1, Role1, Org1, N1, L1, Req1, Prov1,$
$Sub2, Role2, Org2, N2, L2, Req2, Prov2, Sub3, Role3,$
$Org3N3, L3, Req3, Prov3, Z) \leftarrow$
$canFlow3(Sub1, Role1, Org1, N1, L1, Req1, Prov1,$
$Sub2, Role2, Org2, N2, L2, Req2, Prov2, Sub3, Role3,$
$Org3, N3, L3, Req3, Prov3, Z_1), Z_1 \in Z$

$safeFlow3(Sub1, Role1, Org1, N1, L1, Req1, Prov1,$
$Sub2, Role2, Org2, N2, L2, Req2, Prov2,$
$Sub3, Role3, Org3N3, L3, Req3, Prov3, Z) \leftarrow$
$canFlow3(Sub1, Role1, Org1, N1, L1, Req1, Prov1,$
$Sub2, Role2, Org2, N2, L2, Req2, Prov2,$
$Sub3, Role3, Org3, N3, L3, Req3, Prov3, Z_1), Z_1 \in Z$

The last two rules are required to complete the policy specification, where `canFlow3*()` allows `canFlow3()` to be defined recursively and `safeFlow3()` defines the final decision after any potential conflict resolution about flows. The important issue here is that `safeFlow3()` and all the predicates used to define `safeFlow3()` must be globally available, including the geographical locations, users and the roles of the participants of the teleconference, which may expose location specific information inadvertently. We now specify the access control polices at the mobile phone, and omit others for the sake of brevity.

$cando_{mobile}(S, R, Org, mobile, L, \{(talkMobile, A, B),$
$(listenMobile, C, D)\}, Z) \leftarrow owner(S, mobile, \emptyset), \emptyset \in Z.$

$cando_{mobile}(S, R, Org, mobile, L, \{(toMobile, A, B),$
$(fromMobile, C, D)\}, Z) \leftarrow des \in C, des \in A,$
$role(S, R, \emptyset), owner(mobile, S, \emptyset),$
$R \in \{recon, commander\},$
$L \notin \{china, Afghanistan\}, \emptyset \in Z.$

$cannotdo_{mobile}(S, R, Org, mobile, L, \{(toMobile, A, B),$
$(fromMobile, C, D)\}, \emptyset) \leftarrow video \in B \cup D,$
$L \in \{china, Afghanistan\}.$
$cando *_{mobile} (S, R, Org, mobile, L, Pr, Re, Z) \leftarrow$
$cando_{mobile}(S, R, Org, mobile, L, Pr, Re, Z_1)$
$Z_1 \in Z.$

$cannotdo *_{mobile} (S, R, Org, mobile, L, Pr, Re, Z) \leftarrow$
$cannotdo_{mobile}(S, R, Org, mobileL, Pr, Re, \emptyset),$
$\emptyset \in Z.$

$do_{mobile}(S, R, Org, mobile, L, Pr, Re, Z) \leftarrow$
$cando *_{mobile} (S, R, Org, mobile, L, Pr, Re, Z_+),$
$\neg cannotdo *_{mobile} (S, R, Org, mobile, L, Pr, Re, Z_-)$
$Z_+, Z_- \in Z.$

The access control policy at the `mobile` module says that the owner of that device may talk and listen to the device in the first rule. The second rule says that the device may connect to a teleconference `manager` through two interfaces `toMobile` and `fromMobile` provided that both streams are able to use `des` encryption, the owner of the mobile device is using it in playing the role of a reconnaissance officer or a commander, and the device is not being operated from `china` or `Afghanistan`. The third rule of the policy explicitly prohibits using `video` from `china` or `Afghanistan`. Fourth and fifth rules are there to resolve authorization conflicts and to ensure that every request is answered affirmatively or negatively (i.e. no queries flounder). Similarly, the other two modules `manager` and `land` telephone can have their own access control policies.

Following the stratification stated in Section 4, we first state service dependencies as rules. They are stated in Table 2 for any site that has a (mobile or land line) telephone. As stated in the table, audio and video individually require cpu_{au}, cpu_v units of CPU resources and buf_{au}, buf_v units of buffer space in order to maintain the *continuity* of the media streams. But if `audio` and `video` are present together, they need an extra cpu_{lip} amount of CPU and $2.(buf_{au} + buf_v)$ buffer (for double-buffering) in order to maintain lip-synchronization at any local site (not for communication). Consequently, the rules of strata 1 and 2 are as follows:

$needs(\{audio, video\}, \{lipSync\}, \emptyset) \leftarrow$
$needs(X, X, , \emptyset) \leftarrow$
$needs*(X, Y, Z) \leftarrow needs(X, Y, Z_1).$
$needs*(X, Y \cup Z, Z_1) \leftarrow needs*(X, Z, Z_2), needs*(Z, Y, Z_3), Z_1, Z_2 \in Z_3.$

At the Manager's site: Pentagon	At the Land Phone's site: Ft. Mead
$owner(ltCook, manager, \emptyset) \leftarrow$ $playRole(ltCook, commOfficer, \emptyset) \leftarrow$ $organization(ltCook, pentagon, \emptyset) \leftarrow$	$owner(cptJones, land, \emptyset) \leftarrow$ $playRole(cptJones, signal, \emptyset) \leftarrow$ $organization(cptJones, usNavy, \emptyset) \leftarrow$
At the Mobile Phone's Site: Iraq	**At the Teleconference's "site"- to be determined**
$owner(sgtJane, mobile, \emptyset) \leftarrow$ $playRole(sgtJane, reconn, \emptyset) \leftarrow$ $organization(sgtJane, usArmy, \emptyset) \leftarrow$	$owner(\{ltCook, cptJones, sgtJane\}, telecon, \emptyset) \leftarrow$ $playRole(participants, \{conferencing\}, \emptyset) \leftarrow$ $organization(participants, global, \emptyset) \leftarrow$

Table 1: Facts Stored in Local Access Control Policy Bases

Service	Needed Services	Resource Requirements	Multi-Set Representation
audio	auContinuity	cpu_a, buf_a	$\{\mid cpu_a, buf_a \mid\}$
video	vContinutiy	cpu_v, buf_v	$\{\mid cpu_v, buf_v \mid\}$
text		cpu_t, buf_t	$\{\mid cpu_t, buf_t \mid\}$
{audio,video}	lipSync	cpu_{lip}, $2.(buf_{au} + buf_v)$	$\{\mid cpu_{lip}, cpu_a, cpu_a, buf_a, buf_a,$ $cpu_v, buf_v, cpu_v, buf_v, \mid\}$

Table 2: Service Dependencies of the Teleconferencing Application at Local sites

$allNeeds(X, Y, Z) \leftarrow needs*(X, Y, Z_1), Z_1 \in Z.$

The first rule state that {`audio,video`} needs {`lipSync`}. The next two rules say that `needs*` is the transitive closure of `needs`, and the last rule collects all dependencies of `X`. The given set of rules imply the conclusion `allNeeds({audio,video},{audio,video,lipSync})`. We now state the rules for computing all resources required at a local site.

$localRes2*(N1, L1, \{(\_, \_, \{X \mid C\}) \mid B\}, Re1, Res,$
$Z) \leftarrow allNeeds(X, Y, Z_1), resourceMap(Y, Res, \emptyset),$
$localRes2*(N1, L1, \{(\_, \_, C) \mid B\}, Re1, Res, Z_2).$
$localRes2*(N1, L1, \{(\_, \_, \emptyset) \mid B\}, Re1, Res)$
$\leftarrow localRes*(N1, L1, B, Re1, Res, Z_3)$
$Z_1, Z_2, Z_3\emptyset \in Z.$
$allLocalRes(N1, L1, Re1, Pr1, Res, Z) \leftarrow$
$localRes*(N1, L1, Re1, Pr1, Res, Z_1), Z_1 \in Z.$

Two rules above show that the CPU and buffer requirements for resources $\{(\_, \_\{X|C\}|B\}$ in the provides interface is recursively computed using the resources required to offer all the services needed for executing X and adding them to the buffer requirement of $\{(\_, \_, C), B\}$. The required resource lookup table is given by the predicate $resourceMap(services, resources)$. Similar recursive rules can be written for the *requires* interface and added to the total resources required for the *provides* interfaces. The last rule computes all the required resources in the predicate `allLocalRes()`. We now model the resource allocation policy for the communication part of the teleconference as follows:

$localResOK(N1, L1, Re1, Pr1, Res, Z) \leftarrow$
$allLocalRes(N1, L1, Re1, Pr1, Res, Z_1),$
$localLimit(L1, MaxRes, \emptyset),$
$Res \sqsubseteq MaxRes, \emptyset, Z_1 \in Z.$

The simple rule above says that if the CPU and Buffer space is below that of the maximum available at the local site $L1$, then the local QoS policy permits the module to be invoked.

7 Related Work

The subject matter addressed in this paper, gluing software modules to construct new modules and applications is an area that has received much attention in the last two decades. It falls within the areas of *component-based software engineering* [30], *architecture description languages* [13] and to some extent, *aspect-oriented software design* [21]. It belongs in the latter because of our concentration in the aspects of QoP and QoS.

The comprehensive survey [13] summarizes the most prominent architecture description languages (ADLs). According to [13], there are about five main ADLs, Rapide [37], UniCon [1], ArTek [29], Wright [3], Meta-H [53] and Darwin [40], of which we review and compare with some. More recently, an XML-based ADL has been introduced by Dashofy et al. [14]. Broy et al. [9] also addresses the central question of *what characterizes a software component?* The authors argue that a component should (1) encapsulate data (hence hides information representation and computational algorithms), (2) implementable in most programming languages, (3) can be hierar-

chically nested, (4) has clearly defined interfaces and (5) can be incorporated in a framework. The framework we propose have all these properties.

Beugnard et al. [8] have identified the following four levels of *contracts* that can be specified in components and their interconnections: (1) Syntactic contracts to specify data type compatibility, (2) Behavioral contracts to specify pre-conditions, post-conditions and invariants, (3) Synchronization contracts to specify dependency constraints between information exchanged within a concurrent context and (4) Quality of Service contracts to specify quantitative properties like maximum response time etc. Our paper extends this list by adding a fifth level of *security* and separate policy from design by having a modular design framework that taken multiple policies, as implied by the plethora of emerging policy research [48, 52].

One of the most cited publications in the area of module composition is Allen and Garlands's *A Formal Basis for Architectural Connection* [4] provides an operational semantics for the `Wright` architecture description language. They say [4] [3] that two modules connecting to each other must express a *role* and a *port* for that purpose. Informally, the *role* describes the purpose of the connection and exposes its interface name so that it can be used by the *port* which formally specify the allowable interactions. They are connected using a *connection* that specify the interaction protocol. The ports and roles are formalized using *Communicating Sequential Processes (CSP)* [31]. The consistency of the specification of two connecting ports with that of their end connectors is addressed as a refinement problem in the *failure divergence model* of CSP and is checkable by the FDR tool [25].

To the extent we know, two other publications have elaborated in the behavioral specification of ports and their connections. The first is the ADL *Darwin* [40] where port behaviors are specified using π-Calculus [41, 45]. The advantage of this work over [4] is its ability to reason about making connections to mobile modules.

Moschoyiannis [42] has formalized component composition using a set-theoretic framework where the (requires and provides) interfaces are given as a collection of methods and the synchronous interactions between the two parties are specified as an abstract partial ordering. The consistency of a connection between two interfaces is verified by checking the consistency of the partial orders expressed by the two interfaces. In a previous work, Shields et al. [46] formalized port behaviors as automatons. Consequently, the former becomes a more abstract versions of the latter. Moschoyiannis [42] continues to advance set theory and automata based formulation of module composition. Our rule-based formulation of module composition using computable set constraints [15, 16, 17] policies were inspired by his set-based formalizations.

There have also been attempts to use λ-calculus [6] based formulations of module composition. Anancona and Zucca [5] have developed the *calculus for module systems (CMS)* by mixing λ-calculus and the *object calculus* of Abadi and Cardelli [2]. In this calculus, a module is chracterized by the values it imports (like requires), exports (like provides) and has as internal variables. Although the definition of module linking is somewhat similar to that of [42], this work defines more operators on modules: taking the *sum*, reducing modules, renaming, assigning modules to module variables, freezing values of variables etc. The authors show that the (module composition) terms in their module calculus has a normal form. In an analogous attempt Lumpe [38] describes λ-forms, a λ-calculus for forms with form variables, bindings of modules to module variables, extensions and restrictions of modules, de-refencing and assigning context for modules. Although this work addresses the *syntactic contacts* in the sense described by Beaugnard et al. [8], they also have a normal form for λ-forms expressions. Goguen et al. [28] also develops an Intuitions [27] based calculus for module composition. Formulated using Category Theory [39], they use aggregation, re-naming, enrichment (i.e. add functionality to a module) information hiding, and parametrization of modules as operators.

Two papers, Gensler et al. [26] and Kim et al. [34] describe rule-based formulations of module compositions. [26] describes a module with respect to its input and output parameters, public interfaces and mandatory properties. The modeler then expresses these properties using custom-made predicates, and specifies Prolog rules to make inferences about properties of modules. But these rules do not provide a general framework to specify policies on propositionally, QoS and security. [34] describes roles and rules about modules in an *adaptable computing model*, inspired by software engineering concerns. The rules sets they have described covers business concerns such as subscriptions, contracts and sales of customer bases etc. Although we were inspired by these attempts, we recognized the need to expand upon their work to develop a comprehensive rule-based framework that specifies policies for module composition, QoP and QoS.

JBCDL [44] introduces the *Jade Bird Component Description Language*, that is a part of a component description language developed for the *Jade Bird Component Library* that allows hierarchical composition of classes of libraries. This system provides templates to describe software modules consisting of parameters, provided and required functions, connections, class members, their implementation code and code dependencies. Although implementable, this work does not explicitly cover QoS and security either. Secondly, parameterized templates made for software modules are difficult to use to express compositional policies.

[32] makes the important observation that exceptions could arise (due to faults) when linked software modules are executed and proposes to specify exception handlers (using an appropriate fault model) to address them during the design phase. We extend this suggestion to incorporate security policies to handle misuses that may arise due to mal-acts (using some mal-actor model).

Templeton and Levitt [51] proposes an abstract syntax to specify attacks using a *requires-provides* syntax. This work has inspired other work that specify security vulnerabilities using pre-conditions and post-conditions. Sidiroglou et al. [47] proposes to construct mediated overlay networks as a compassable service. They argue that many security vulnerabilities can be avoided by using the appropriate security services. Furthermore Makio [43] proposes a *requires-provides* based language to specify negotiation structures that can model complex market place negotiations. Their models specify synchronization details of the ports to specify acceptable business negotiation

strategies.

8 Conclusions

We have constructed a CLP(Set) based policy language that is capable of specifying module composition based on their exposed requires-provides interfaces. As shown, our language is able to specify and enforce policies related to multiple *aspects* of the composition. We have shown how QoP and QoS can be used as sample aspects, and a modular way to decompose and specify policies that govern the composition into sub-aspects within an aspect. Towards this end, we have shown how access and flow control policies can be specified as sub components of security policies.

We have also provided an operational semantics for the entire policy language and shown that our policies are flounder-free. That is the rule execution engine will always return an *yes* and *no* response to every composition request.

References

[1] *The UniCon Architecture Description Language*.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science, Springer-Verlag, 1996.

[3] Robert Allen and David Garlan. Beyond definition/use: Architectural interconnection. In *Proceedings of the Workshop on Interface Definition Languages*, Portland, Oregon, 1994.

[4] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[5] Davide Anancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[6] Henk P. Barendrecht. *The Lambda Calculus – Its Syntax and Semantics*. North-Holland, rev. edn, 1984.

[7] Steve Barker and Peter J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transactions on Information and System Security*, 2004. to appear.

[8] Antoine Beaugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damian Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[9] Mansfred Broy, Anton Dieme, Jurgen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemnts Szyperski. What characterizes a (software) component. *Software-Concepts and Tools*, 19:49–56, 1998.

[10] David Chan. Constructive negation based on the completed databases. In R. A. Kowalski and K. A. Bowen, editors, *Proc. International Conference on Logic Programming (ICLP)*, pages 111–125. The MIT Press, 1988.

[11] David Chan. An extension of constructive negation and its application in coroutining. In E. Lusk and R. Overbeek, editors, *Proc. North-American Conference on Logic Programming*, pages 477–489. The MIT Press, 1989.

[12] Shiping Chen, Duminda Wijesekera, and Sushil Jajodia. Flexflow: A flexible flow control policy specification framework. In *DBSec*, pages 358–371, 2003.

[13] Paul Clements. A survey of architecture description language. In *Eighth International Workshop on Software Specification and Design*, pages 16–28, 1996.

[14] Eric M. Dashofy, Andre van der Hoek, and Richard Taylor. A highly-extensioble, xml-based architecture description language. In *Proceedings of the IEEE/IFIP Working Conference on Software Architecture*, pages 103–112, 2001.

[15] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Transactions of Programming Languages and Systems*, 22(5):861–931, 2000.

[16] Agostino Dovier, Carla Piazza, and Gianfranco Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. Technical Report Quaderno 235, Department of Mathematics, University of Parma, Italy, 2000.

[17] Agostino Dovier, Alberto Policriti, and Gianfranco Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundamenta Informaticae*, 36(2/3):201–235, 1998.

[18] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Constructive negation and constraint logic programming with sets. *New Generation Comput*, 19(3):209–256, May 2001.

[19] Francois Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.

[20] Francois Fages and Roberta Gori. A hierarchy of semantics for normal constraint logic programs. In *Algebraic and Logic Programming*, pages 77–91, 1996.

[21] Robert E. Filman, Tzilla Elrad, Siobhan Clark, and Mehmet Aksit. *Aspect-oriented Software Development*. Addison-Wesley, 2005.

[22] Melvin C. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[23] Melvin C. Fitting. Fixedpoint semantics for logic programming. *Theoretical Computer Science*, 278:25–31, 2002.

[24] Melvin C. Fitting and Marion Ben-Jacob. Stratified, weak stratified, and three-valued semantics. *Fundamenta Informaticae, Special issue on LOGIC PROGRAMMING*, 13(1):19–33, March 1990.

[25] FormalSystems. *The FDR Tool*. available at http://www.fsel.com/fdr2_download.html.

[26] Thomas Gensler and Christian Zeidler. Rule-driven component composition for embedded systems.

[27] Joseph Goguen and Rodney Burstall. Intuitions: Abstract model for specification and programming. *Journal of the Association of Computing Machinery*, 39(1):95–146, january 1992.

[28] Joseph Goguen and Grigoe Rosu. *Composition of Modules with Hidden Information over Inclusive Institutions*, volume LNCS 2635, chapter 11. Springer-Verlag, 2004.

[29] Terry Hays-Roth and Ross Klein. Abstractions for software architectures and tools to support them. Carnegie Mellon University, 1994.

[30] George T. Heineman and William T. Councill. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley professional, 2001.

[31] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[32] Viliam Holub. Enhancing behavior protocols with exceptions.

[33] Joxann Jaffar and Jean-Louise Lassez. Constraint logic programming. *Proceedings of Principles of of Programming Languages*, pages 111–119, 1987.

[34] Jeong Ah Kim, JinYoung Taek, and SunMyung Hwang. Rule-based component development. In *SERA '05: Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications*, pages 70–74, Washington, DC, USA, 2005. IEEE Computer Society.

[35] Dexter C. Kozen. Set constraints and logic programming. *Information and Computation*, 142:2–25, 1998. Article No IC972694.

[36] Kenneth J. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):298–308, December 1987.

[37] David Luckham, John J. Kenney, Larry M. Augustine, James Vera, Dough Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. Technical report, Stanford University, 1993.

[38] Markus Lumpe. *A Lmabda Calculus with Forms*, volume LNCS 3628, chapter 11, pages 83–98. Springer-Verlag, 2005.

[39] Saunders MacLane. *Categories for a Working Mathematician*. Springer-Verlag, 1998.

[40] Richard Magee, Narankar Dulay, Susan Eisenbach, and Jeffery Kramer. Specifying distributed software architecture. In *Proceedings of the Fifth Europeon Software Engineering Conference ESEC'95*, 1995.

[41] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 2001.

[42] Sotris Moschoiannis and Michael Shields. A set-theoretic framework for component composition. *Fundamenta Informaticae*, 59:373–396, 2004.

[43] Juho Mki and Ilka Weber. Component-based specification and composition of market structures.

[44] Wu Qiong, Chang Jichuan, Mei Hong, and Yang Fuqing. Jbcdl: An object-oriented component description language. In *TOOLS '97: Proceedings of the Technology of Object-Oriented Languages and Systems-Tools - 24*, page 198, Washington, DC, USA, 1997. IEEE Computer Society.

[45] David Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[46] Michael W. Shields and Sotris Moschoyiannis. An automata theoretic view of software components. Technical Report SCOMP-TC-02-04, Department of Computing, University of Surrey, 2004.

[47] Stelios Sidiroglou, Angelos Stavrou, and Angelos D. Keromytis. Network security as a composable service. In *Proceedings of the IEEE Sarnoff Symposium.*, January 2007.

[48] John Strassner. *Policy-Based Network Management*. Morgan Kaufmann, 2004.

[49] Peter J. Stuckey. Constructive negation for constraint logic programming. In *Logic in Computer Science*, pages 328–339, 1991.

[50] Peter J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

[51] Steven Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 New Security Paradigms Workshop*, pages 31–38, 2000.

[52] Dinesh Verma. *Architecture and Algorithms*. New Riders, 2000.

[53] S. Vestal. Mode changes ina real-time architecture description language. In *Proceedings of the International Workshop on Configurable Distributed Systems*. Honeywell Technology Center and University of Maryland.

[54] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *FMSE*, pages 45–55, 2004.