QoP and QoS Policy Cognizant Module Composition

Paul Seymer, Angelos Stavrou, Duminda Wijesekera, and Sushil Jajodia
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
{*pseymer, astavrou, dwijesek, jajodia*}*@gmu.edu*

***Abstract*—Component-based software engineering is generally recognized as one of the best methods to develop, deploy, and manage increasingly complex software systems. To enable the dynamic composition of software modules, it is often required to expose their functionality dependencies. This results the a well-known *requires-provides* specifications' model.**

In this paper, we introduce a framework that enables individual software components to specify their requires-provides interfaces in a policy dependent way. Our framework specifies policies as combinations of Constraint Logic Programming (CLP) based rules. Moreover, our policies are flexible and expressive, allowing the enforcement of multiple *aspects* for the requested composition including security and quality of service. We apply our framework to specify Quality of Protection (QoP) and Quality of Service (QoS) policies. We demonstrate the applicability of our policy language using as an example a teleconferencing application with diverse requirements for the specification of security and resource policies.

***Keywords*-Policy-based software composition; Policies for software interfaces; Policies for aspect-oriented software**

I. Introduction

In component-based software development, modules are developed independently and composed on an *as needed* basis [1]. 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 I 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 II frames a case study that will be referenced throughout the paper. Section III presents the formal language of the rule-based composition framework. Section IV specifies the sub-language for security policies and Section V the sub-language for QoS policies. Section VI details the semantics and models for module composition syntax. Section VII uses the theory from the previous sections to complete the case study. Related work and concluding remarks are in Sections VIII and IX respectively.

II. Case Study: Group Teleconferencing

This section shows an example application of our framework to a conference call manager. As shown in the schematic of Figure II, 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.

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

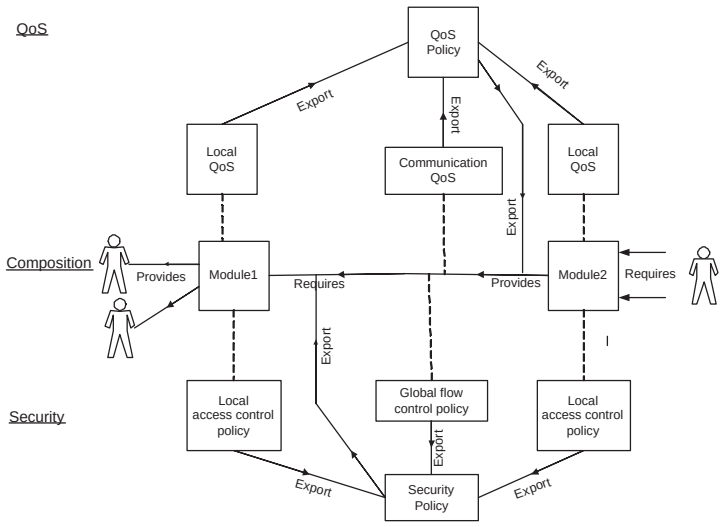

Figure 1. Architecture of the Policy Framework

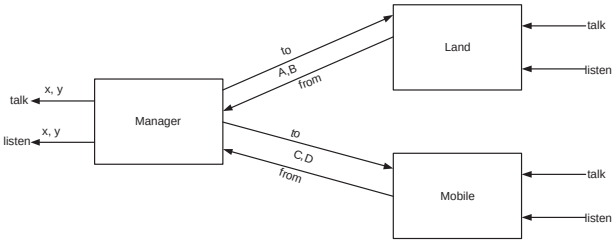

Figure 2. The Composed Teleconferencing Module.

textual data and the second interface is used to receive the same data from the other two conferees.

We will be adding to this case study as we progress through the sections of this paper.

III. A Rule-based Language for 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. [2], [3], [4], 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 $\{=, \neq, \in, \notin, \cup_3, \not\cup_3, \|, \not\|\}$, 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 referred 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**. We 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(`$\mathtt{C}_n$`,Depth)` is a predicate used to represent a module. $\mathtt{C}_n$ is a shorthand notation for a set of 4 constants or variables $\mathtt{N}_n$, $\mathtt{L}_n$, $\mathtt{Pr}_n$, and $\mathtt{Re}_n$ that represent a component. $\mathtt{N}_n$ is a constant or variable from Set_N representing a name. Similarly, $\mathtt{L}_n$ is a constant or a variable from Set_L, representing a location (or a set of locations). $\mathtt{Pr}_n$ is a constant or a variable representing an ordered triple of the type $Set_I \times Set_{QoP} \times Set_{QoS}$ used to model a set with elements of the form (interface, Set of QoP and QoS options) representing the collection of *provides* interfaces with their options. Similarly, $\mathtt{Re}_n$ is a constant or variable from the same type as that of $\mathtt{Re}_n$, representing the interfaces and the options of the *requires* interfaces of the component.

`Depth` is a set term used to encode the recursive depth (Sometimes shown as `Z` or $\mathtt{Z}_n$). In this version where we do not use recursive compositions, we use the value of $\emptyset$ for the `Depth` attribute. `cModule(`$\mathtt{C}_n$`,Depth)` is a

predicate with the same type of predicates as the module predicate, representing a composed module. In order to specify access and flow control policies, we add three more sorts, Sub for subjects and and Role for roles and Org for organizations. We will use F_n as a shorthand for these set of parameters. composable2(C_1,C_2,Z) is an predicate where interfaces of the second module are connected to the *requires* interfaces of the first module.

We also use another predicate composable3(...,C_3) 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(C_1,C_2,Z) is a predicate that specifies the structural compatibility requirements between the *provider* and the *requester* modules. Similarly, we use another predicate structOK3(...,C_3,Z) to model the structural integrity of a three sub-modules composition. qopOK2(C_1,C_2,Z), and qosOK2(C_1,C_2,Z) are predicates used to specify the security policies and QoS policies, respectively, of a module composition. Similarly, qopOK3(...,C_3,Z) and qosOK3(...,C_3,Z) are used for composing three modules. typeOK2(C_1,C_2,Z) is a predicate used to specify type compatibility of two modules. Similarly, we use another predicate typeOK3() to specify compatibility of three modules. Structural composability can be specified using the set constraint language and other user defined predicates as given in the following examples:

$$structOK2(C_1, C_2, Z) \leftarrow cModule(C_1, S_1), \\ cModule(C_2, S_2), typeOK2(C_1, C_2, S_{type}) \\ Pr_1 \cap Pr_2 = \emptyset, Re_1 \cap Re_2 = \emptyset, Re_1 \subseteq Pr_2, \\ S_1, S_2, S_{type} \in S.$$

This example shows that two modules N_1 and N_2 are structurally compatible iff they do not share any *provides* interfaces or *requires* interfaces and the *requires* interfaces of the first module N_1 are provided by the second module N_2. By adding an extra clause $Re_1 \subseteq Pr_2$ to the policy, we can allow partial compositions where the second module N_2 can provide more interfaces than those required by N_1. Also by reversing the constraint $Re_1 \subseteq Pr_2$ to $Re_1 \supseteq Pr_2$, we can accept *partial* compositions where the second module is unable to provide all required interfaces. By doing so, we can partially compose N_1 and N_2 and then compose this intermediate result with another module N_3 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.

Returning to our case study, 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 modeled 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 can now explain how this case study composition is actually 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.

IV. QoP Policies

As mentioned, our sample QoP policies either control access to modules or information that flows between them. We modify *Attribute-based Access Control* [5] of Wang et al. for the former and *FlexFlow* [6] 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 predicates for Flow Control, and Access Control.

`secureFlow2(...)` mandates that information is allowed to flow between modules $\texttt{C}_1$ and $\texttt{C}_2$ when they are composed. A similar predicate `secureFlow3(...,`$\texttt{F}_3$`,`$\texttt{C}_3$`,Z)` is used to exercise flow control between three modules. `do(Sub,Role,Org,`$\texttt{L}_n$`,`$\texttt{N}_n$`,`$Re_n \cup Pr_n$`,Z)` details that a subject `Sub` playing role `Role` belonging to organization `Org` belonging to module $\texttt{N}_n$. Using these two predicates exported from the global flow control policy specification module and the two local policy specification modules at locations L_1 and L_2 are related to `qopOK2(`$\texttt{C}_1$`,`$\texttt{C}_2$`,Z)` by the following rule:

$$\begin{aligned}
qopOK2(C_1, C_2, Z) \leftarrow secureFlow2(F_1, C_1, F_2, C_2, Z_f) \\
do_{L1}(F_1, L_1, C_1, Re_1 \cup Pr_1, Z_1)), \\
do_{L2}(F_2, L_2, C_2, Re_2 \cup Pr_2, Z_2)), Z_f, Z_1, Z_2 \in Z
\end{aligned}$$

As described, in the above rule, the requester-provider policy is considered secure if the flow is permitted by flow control predicate $secureFlow2(C_1, C_2, Z_f)$, and the two access control predicates $do_{L1}(F_1, N_1, L_1, Re_1 \cup Pr_1, Z_1)$ and $do_{L2}(F_2, N_2, L_2, Re_2 \cup Pr_2, Z_2)$ allow subject $Sub_1 \in F_1$ to execute N_1's interfaces and subject $Sub_2 \in F_2$ to execute N_2's interfaces respectively.

A. The Flow Control Sub-language

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

`owner(s,m,`$\emptyset$`)` with $m \in Names$ and $s \in Sub$ is true if module m is owned by subject s. Similarly, `playRole(s,r)` where $r \in Role$ details 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,`$\emptyset$`)`, saying that the subject `s` works for the organization `org`. 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 $organization(s, org, \emptyset) \leftarrow$. We use two predicates `canFlow2(`F_1`,`C_1`,`F_2`,`C_2`,Z)` and `cannnotFlow2(`F_1`,`C_1`,`F_2`,`C_2`,Z)` in stratum 2. Similar definitions exists for predicates `canFlow3(...)` and `cannotFlow3(...)` used for ternary compositions. These head predicates must be one of `canFlow2()`, `cannotFlow2()`, `canFlow3()` or `cannotFlow3()` and the bodies may contain predicates from stratum 1 and constraints from `CLP(Set)`. For an example see [7].

In stratum 3 we use `canFlow2*(`F_1, C_1, F_2, C_2, Z`)` and `cannnotFlow2*(`F_1, C_1, F_2, C_2, Z`)` to recursively specify flow control permissions and prohibitions. 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. In stratum 4 we use `secureFlow2(`F_1`,`C_1`,`F_2`,`C_2`,Z)` and `secureFlow3(`F_1`,`C_1`,`F_2`,`C_2`,`F_3`,`C_3`,Z)` to evaluate the final decision to allow information flow. This stratum contains the following rule:

$$secureFlow2(F_1, C_1, F_2, C_2, Z) \leftarrow$$
$$canFlow2*(F_1, C_1, F_2, C_2, Z_1),$$
$$\neg cannotFlow2*(F_1, C_1, F_2, C_2, Z_2), Z_1, Z_2 \in Z.$$

FlexFlow [6] 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 II shows how to use the flow control sub language in enforcing security policies for module composition.

B. 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 four level stratified logic programming language with reserved predicates appropriately adopted from [5].

Stratum 1 has predicates to represent basic facts such as ownerships, subject-role assignments, object and subject hierarchies etc. We specify basic relationships and application specific facts written as predicate instances (i.e. rules with empty bodies). Stratum 2 has predicates $cando(F_n, N_n, L_n, Re_n \cup Pr_n, Z)$ and $cannotdo(F_n, N_n, L_n, Re_n \cup Pr_n, Z)$ that state which module options are permitted or prohibited from executing. Rules using `cando` and `cannotdo` heads may have predicates from lower strata and constraint expressions. These are used to state basic facts about granting/denying access to services. `cando*(`$F_n, N_n, L_n, Re_n \cup Pr_n, Z$`)` and `cannotdo*(`$F_n, N_n, L_n, Re_n \cup Pr_n, Z$`)` are two predicates used in stratum 3 to recursively extend the definitions of `cando()` and `cannotdo()`. Rules with these heads can have `cando*`, `cannotdo*` predicates in their bodies only positively, but may have `cando` and other non-reserved predicates and constraint terms. In stratum 4, `do(`$F_n, N_n, L_n, Re_n \cup Pr_n, Z$`)` expresses the final decision about a module being able to execute with specified options. $do(\ldots, Z) \leftarrow cando*(\ldots, Z_+), \neg cannotdo*(\ldots, Z_-)$, $Z_+, Z_- \in Z$ is the only rule at this stratum.

Any finite collection of rules conforming to constraints in strata (1) through (4) is said to be an access control policy. Wang et al. [5] 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.

V. QoS Policies

Our representation of QoS policies is similar in many ways to our QoP representation. 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 four level stratified constraint structure similar to QoP.

Stratum 1 specifies service dependencies. The predicate $needs(S, T)$ mandates that the (subsidiary) set of services T are required to provide the set of (primary) services S. $allNeeds(S, T)$ dictates that the set of services T consists of all subsidiary services are required to provide the

set of primary services S. 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. Stratum 2 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 $comRes(C_1,C_2,R,Z)$ and $comRes*(C_1,C_2,R,Z)$ are used. $comRes(\ldots)$ specifies that R resources are required to communicate between modules $N1$ and $N2$. The second predicate, $comRes*(\ldots)$, is used to recursively compute resource needs of dependent services. Finally, the predicate $allComRes(\ldots)$ mandates 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 predicates $localRes(C_1,R,Z)$, $localRes*(C_1,R,Z)$, and $allLocalRes(C_1,R,Z)$. Rules at this stratum recursively define `comRes*()` and `localRes*()` using positive occupancies of themselves, `comRes()`, `localRes()` and (multi)set constraints respectively. Consequently bodies of $comRes*()$ may have any predicate from stratum 1, but it and $localRes*()$ should not have `comRes()` and `localRes()` appearing negatively. `allComRes()` and `allLocalRes()` collect all resources needed to communicate between sites $L1 \in C_1$ and $L2 \in C_2$ and at site $L1$ respectively. Bodies of $allComRes()$ and $allLocalRes()$ may have any other predicates belonging to a lower strata. Stratum 3 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 indicates 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 $R2$. 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. Stratum 4 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, $comResOK(C_1,C_2,R,Z)$ and $localResOK(C_1,R,Z)$, 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. Rules at this stratum have $comResOK(C_1,C_2,Z)$ or $localResOK(C_1,Z)$ 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. 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()`. At this stratum, QoS requirements are satisfied iff the local computing requirements and communication requirements are satisfied.

VI. Language 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 [8], [9] 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 [12], [13]. This is because the third alternative namely using constructive negation as proposed by Stuckey [14], [15] 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 [16]. 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 [12], [13]. Formalities follow. We first repeat some standard definitions as they appear in [17] in order to clarify notation.

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: See [7]. ■

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 shows 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 [18]. ■

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 [19], [13]. ([19] gives a simple counter example)

VII. Case Study (Continued)

We can now continue up our case study by showing its 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 I. 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 I 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`.

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

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

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.

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

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

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

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

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

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 I
FACTS STORED IN LOCAL ACCESS CONTROL POLICY BASES

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 V, we first state service dependencies as rules. They are stated in Table II 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:

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

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.

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 II
SERVICE DEPENDENCIES OF THE TELECONFERENCING APPLICATION AT LOCAL SITES

VIII. RELATED WORK

The idea of composition of software modules to form applications has received much attention: *component-based software engineering* [1], *architecture description languages* [20] and, to some extent, *aspect-oriented software design* [21]. Our work is closer to aspect-oriented software design since our policies can be considered aspect-oriented (QoS, QoP).

The most recognized Architecture Description Languages (ADLs) are summarized in [20]. There exist about five main ADLs: Rapide [22], UniCon [23], ArTek [24], Wright [25], Meta-H [26] and Darwin [27], of which we review and compare with. More recently, an XML-based ADL has been introduced by Dashofy et al. [28]. Broy et al. [29] also addresses the central question of *what characterizes a software component?* The authors argue that a component should (1) encapsulate data, (2) implementable in most programming languages, (3) can be hierarchically nested, (4) has clearly defined interfaces and (5) can be incorporated in a framework. Our framework has all these properties.

Allen and Garlands's *A Formal Basis for Architectural Connection* [30] provides an operational semantics for the `Wright` architecture description language. They posit that modules connecting to each other must express a *role* and a *port* for that purpose [30], [25]. The *role* describes the purpose of the connection and exposes its interface name so that it can be used by the *port* which formally specifies the allowable interactions. They are connected using a *connection* that defines 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 [32].

Two other publications have described the behavioral specification of ports and their connections. The first is the ADL *Darwin* [27] where port behaviors are specified using π-Calculus [33], [34]. Contrary to previous work, our approach enables us to reason about making connections to mobile modules.

Gensler et al. [35] and Kim et al. [36] describe rule-based formulations of module compositions. We expand their theoretical work to develop a comprehensive rule-based framework that defines policies for module composition, QoP, and QoS.

In terms of implementation, JBCDL [37] introduces the *Jade Bird Component Description Language*, that is a part of a component description language developed for the *Jade Bird Component Library*. JBCDL allows hierarchical composition of classes of libraries through the use of templates to describe software mofules. This work does not explicitly cover QoS and security either. Moreover, parameterized templates made for software modules are difficult to use to express compositional policies.

IX. CONCLUSIONS

We introduced a CLP(Set) based policy language that is capable of specifying module composition based on their exposed requires-provides interfaces. Our language is able to specify and enforce policies related to multiple *aspects* of the composition. Furthermore, we demonstrated how QoP and QoS can be defined as aspects. In addition, we provided a modular way to decompose and specify policies that govern the composition into sub-aspects within an aspect. As an example, we showed how access and flow control policies can be specified as sub components of security policies. Finally, we developed operational semantics for the entire policy language and shown that our policies are flounder-free: the rule execution engine will always return an *yes* and *no* response to every composition request.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants CT-20013A, CT-0716567, CT-0716323, and CT-0627493; by the Air Force Office of Scientific Research under grants FA9550-07-1-0527, FA9550-09-1-0421, and FA9550-08-1-0157; and by the Army Research Office DURIP award W911NF-09-01-0352.

REFERENCES

[1] G. T. Heineman and W. T. Councill, *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley professional, 2001.

[2] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi, "Sets and constraint logic programming," *ACM Transactions of Programming Languages and Systems*, vol. 22, no. 5, pp. 861–931, 2000.

[3] A. Dovier, C. Piazza, and G. Rossi, "A uniform approach to constraint-solving for lists, multisets, compact lists, and sets," Department of Mathematics, University of Parma, Italy, Tech. Rep. Quaderno 235, 2000.

[4] A. Dovier, A. Policriti, and G. Rossi, "A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms," *Fundamenta Informaticae*, vol. 36, no. 2/3, pp. 201–235, 1998.

[5] L. Wang, D. Wijesekera, and S. Jajodia, "A logic-based framework for attribute based access control," in *FMSE*, 2004, pp. 45–55.

[6] S. Chen, D. Wijesekera, and S. Jajodia, "Flexflow: A flexible flow control policy specification framework," in *DBSec*, 2003, pp. 358–371.

[7] P. Seymer, A. Stavrou, D. Wijesekera, and S. Jajodia, "Security policy cognizant module composition," Available at http://cs.gmu.edu, Department of Computer Science, George Mason University, Tech. Rep. GMU-CS-TR-2010-1, 2010.

[8] K. J. Kunen, "Negation in logic programming," *Journal of Logic Programming*, vol. 4, no. 4, pp. 298–308, December 1987.

[9] M. C. Fitting, "A kripke-kleene semantics for logic programs," *Journal of Logic Programming*, vol. 2, no. 4, pp. 295–312, 1985.

[10] D. Chan, "Constructive negation based on the completed databases," *Proc. International Conference on Logic Programming (ICLP)*, pp. 111–125, 1988.

[11] ——, "An extension of constructive negation and its application in coroutining," *Proceedings North-American Conference on Logic Programming*, pp. 477–489, 1989.

[12] F. Fages and R. Gori, "A hierarchy of semantics for normal constraint logic programs," in *Algebraic and Logic Programming*, 1996, pp. 77–91.

[13] F. Fages, "Constructive negation by pruning," *Journal of Logic Programming*, vol. 32, no. 2, pp. 85–118, 1997.

[14] P. J. Stuckey, "Constructive negation for constraint logic programming," in *Logic in Computer Science*, 1991, pp. 328–339.

[15] ——, "Negation and constraint logic programming," *Information and Computation*, vol. 118, no. 1, pp. 12–33, 1995.

[16] A. Dovier, E. Pontelli, and G. Rossi, "Constructive negation and constraint logic programming with sets," *New Generation Comput*, vol. 19, no. 3, pp. 209–256, May 2001.

[17] M. C. Fitting, "Fixedpoint semantics for logic programming," *Theoretical Computer Science*, vol. 278, pp. 25–31, 2002.

[18] S. Barker and P. J. Stuckey, "Flexible access control policy specification with constraint logic programming," *ACM Transactions on Information and System Security*, 2004, to appear.

[19] M. C. Fitting and M. Ben-Jacob, "Stratified, weak stratified, and three-valued semantics," *Fundamenta Informaticae, Special issue on LOGIC PROGRAMMING*, vol. 13, no. 1, pp. 19–33, March 1990.

[20] P. Clements, "A survey of architecture description language," in *Eighth International Workshop on Software Specification and Design*, 1996, pp. 16–28.

[21] R. E. Filman, T. Elrad, S. Clark, and M. Aksit, *Aspect-oriented Software Development*. Addison-Wesley, 2005.

[22] D. Luckham, J. J. Kenney, L. M. Augustine, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using Rapide," Stanford University, Tech. Rep., 1993.

[23] *The UniCon Architecture Description Language*.

[24] T. Hays-Roth and R. Klein, "Abstractions for software architectures and tools to support them," 1994, carnegie Mellon University.

[25] R. Allen and D. Garlan, "Beyond definition/use: Architectural interconnection," in *Proceedings of the Workshop on Interface Definition Languages*, Portland, Oregon, 1994.

[26] S. Vestal, "Mode changes in a real-time architecture description language," in *Proceedings of the International Workshop on Configurable Distributed Systems*. Honeywell Technology Center and University of Maryland, 1994.

[27] R. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architecture," in *Proceedings of the Fifth Europeon Software Engineering Conference ESEC'95*, 1995.

[28] E. M. Dashofy, A. van der Hoek, and R. Taylor, "A highly-extensioble, xml-based architecture description language," in *Proceedings of the IEEE/IFIP Working Conference on Software Architecture*, 2001, pp. 103–112.

[29] M. Broy, A. Dieme, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski, "What characterizes a (software) component," *Software-Concepts and Tools*, vol. 19, pp. 49–56, 1998.

[30] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, July 1997.

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

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

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

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

[35] T. Gensler and C. Zeidler, "Rule-driven component composition for embedded systems." [Online]. Available: citeseer.ist.psu.edu/483080.html

[36] J. A. Kim, J. Taek, and S. Hwang, "Rule-based component development," in *SERA '05: Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 70–74.

[37] W. Qiong, C. Jichuan, M. Hong, and Y. Fuqing, "Jbcdl: An object-oriented component description language," in *TOOLS '97: Proceedings of the Technology of Object-Oriented Languages and Systems-Tools - 24*. Washington, DC, USA: IEEE Computer Society, 1997, p. 198.