Design profiles: toward unified tool support for design patterns and UML profiles

Imed Hammouda*, †, Anna Ruokonen, Mika Siikarla, André L. Santos, Kai Koskimies and Tarja Systä

Department of Software Systems, Tampere University of Technology, P.O. Box 553, FI-33101 Tampere, Finland

SUMMARY

In the current trend of software engineering, patterns and profiles lend themselves as two powerful complementary mechanisms for documenting and enforcing design rules. In this paper, we study how unified tool support can be achieved for patterns and profiles by interpreting them as role-based structural descriptions and exploiting existing tool support for role modeling. The approach is realized using so-called design profiles, which allow both validation and generation of models according to rules expressed in the patterns and profiles. A prototypical tool environment supporting the approach has been developed. The practical applicability of the approach is demonstrated by specifying design rules of the enterprise Java beans (EJB) platform and by producing an environment for designing EJB-based applications. We also discuss possible usage scenarios in the context of this case study. Copyright © 2008 John Wiley & Sons, Ltd.

Received 10 July 2007; Revised 26 February 2008; Accepted 24 May 2008

KEY WORDS: UML profiles; software patterns; model generation and validation; tool support

*Correspondence to: Imed Hammouda, Department of Software Systems, Tampere University of Technology, P.O. Box 553, FI-33101 Tampere, Finland.
†E-mail: imed.hammouda@tut.fi

Contract/grant sponsor: Finnish Funding Agency for Technology and Innovation; contract/grant numbers: 40226/04, MoDES 40081/07
Contract/grant sponsor: Nokia
Contract/grant sponsor: Nokia Siemens Networks
Contract/grant sponsor: Nokia Foundation
Contract/grant sponsor: TietoEnator
Contract/grant sponsor: Plenware
Contract/grant sponsor: Solita
Contract/grant sponsor: John Deere
Contract/grant sponsor: TISE
Contract/grant sponsor: SoSE

Copyright © 2008 John Wiley & Sons, Ltd.
1. INTRODUCTION

Software development is typically guided and constrained by various kinds of design rules implied by the domain, architectural decisions, middleware standards, platform conventions, company-specific best practices, etc. Development tools should support the specification and enforcing of such design rules during the development process. In particular, as the design rules concern issues emerging at the architecture and detailed design levels, the rules should be enforced by the modeling tools employed for architectural and design descriptions.

In the context of UML modeling, profiles [1] have been a standard mechanism to express design rules. A profile is a specialization of the UML metamodel, introducing new standard elements (stereotypes) as extensions of the basic UML concepts and well-formedness rules specifying the validity of models. Essentially, a profile is a UML-based modeling language customized for a particular purpose. If the purpose is to support the design of certain kinds of systems, the profile factually defines a coherent set of design rules that are enforced on UML models through stereotyping. Examples of such profiles include the enterprise Java beans (EJB) profile [2] and the CORBA profile [3].

On the other hand, various kinds of design rules documenting proven good solutions and best practices are expressed using design patterns [4]. Pattern languages [5] are organized collections of patterns that provide a comprehensive, consistent set of proven solutions for a particular system category or domain. In contrast to profiles, patterns are usually described in the literature informally, without an underlying metamodel or formal constraints. Design patterns can be expressed using the collaboration concept in UML [6], defining a structure of collaborating roles. A role prescribes properties that the participating objects playing the roles must provide. Examples of pattern languages include EJB patterns [7,8] and CORBA patterns [9].

When used to document the design rules of platforms such as EJB and CORBA, profiles and patterns can be regarded as complementary. Profiles are typically used to document the concepts and constraints of a platform, whereas patterns are used to document general proven solutions applied in the context of that platform. For example, we could have a design model for an EJB-based application that is valid according to the rules of the EJB component framework (documented in the EJB profile), but yet the model does not necessarily follow EJB best practices (documented in EJB design patterns).

In principle, a profile defines a set of rules a designer is committed to follow throughout the model, whereas a design pattern gives only a recommended way of solving a problem. However, sometimes this distinction may be obscure. For example, some EJB patterns are ‘almost mandatory’ in the sense that missing their usage in appropriate places can be seen as a design flaw. Artificial division of design rules into ‘mandatory’ profiles and ‘recommended’ patterns can lead to overlapping and even conflicting profiles and patterns.

Profile and design pattern specifications can be exploited in various ways. In principle, there are two basic types of tool support for design rules provided by modeling environments: generative tool support allows the designer to automatically produce some part of a model according to the design rules, whereas validating tool support gives facilities for checking a model against the rules. Both tool types can be further classified on the basis of the designer’s involvement: generation or checking can be performed as a batch operation without the designer’s involvement (e.g. [10,11]), or interactively with the designer (e.g. [12,13]). Profile tools incorporated in modeling environments (e.g. [14,15]) typically support the validating approach, either in the
batch mode or in the interactive mode. However, support for the generative usage of profiles is modest.

On the other hand, design pattern support typically emphasizes the generative mode: an instance of a design pattern can be produced automatically by the tool, using the application-specific information provided by the designer (e.g. [16,17]). Often design pattern support relies on non-standard notation (e.g. [18,19]). We conclude that currently available tool support for design rules is rather diverse and strongly depends on whether the rules are part of profiles or design patterns.

Our aim is to strengthen tool support both for design patterns and profiles by conceptual unification, developing a tool concept for design rules, which is neutral with respect to the origin of the rules and to the type of support provided by the tool. We do not see a fundamental difference with the design rules that are expressed in profiles and with those expressed in design patterns, and we believe that the same tool concept should allow all the usage modes discussed above in a uniform way for all design rules. A basis of this work is the equivalence result presented in [20], showing that, with a certain interpretation of patterns and profiles, profiles can be transformed into patterns and vice versa.

The main contributions of this paper are a proposal for a unified tool concept covering the structural aspects of both design patterns and profiles, its prototype implementation, and a preliminary evaluation of the concept. We argue that such a tool unification is valuable per se, because it reduces the tool concept set required for software architecting. The main concrete benefit of the approach is that it extends available tool support especially for profiles: profiles can exploit generative support, which provides efficient assistance for creating models complying with the profile rules.

Our approach is based on the idea that design rules can be expressed on an abstraction level higher than profiles and patterns based on role modeling. Role modeling has been used for the abstract representation of design patterns [21], but we argue that role modeling applies to profiles as well. Thus, the design rules originating from both design patterns and profiles can be expressed in terms of role models. Following the UML terminology, we will call sets of such specifications of design rules design profiles; that is, a design profile is a role-model-based description of a set of design rules given for a particular domain or system category.

An advantage of this approach is that we can exploit an existing tool platform for role-based modeling [12]. Our prototype tool, named INARI (integrated architecting environment) [22], provides facilities for both generative and validating usage modes. The tool can thus generate missing elements implied by the design rules, and check the models against the design rules. To make the creation and understanding of design profiles easier, we have developed a UML-based representation of design profiles where a design rule is given as a class diagram, similar to the way the solution parts of design patterns are given in the literature. We will here focus on structural modeling, although in principle similar ideas could be applied to behavioral modeling as well. In this work, we demonstrate our approach by specifying design rules of the EJB platform, and by producing an environment for designing EJB-based applications. A video recording of INARI applied in the context of the EJB case study (discussed later) is available in the INARI project Web site [23].

We proceed as follows. In Section 2, we discuss the main characteristics of patterns and profiles and their specification using role modeling. A concrete realization of such a role model together with tool support is presented in Section 3. In Section 4, we show how the proposed approach can be applied for specifying a design language for EJB-based applications. In Section 5, experiences
on the usage of the tool are briefly discussed. In Section 6, we relate our work to other approaches. We conclude with final remarks in Section 7.

2. ROLE MODELING OF PROFILES AND PATTERNS

As a first step toward the ‘marriage’ of profiles and design patterns, let us study some of the main features of both concepts. Table I summarizes the main characteristics of profiles and patterns with respect to a number of issues. From our viewpoint, the most interesting observation is that the concepts use different terminologies in the structural specification: the solution part of design patterns is presented in terms of element (participant) configurations with well-defined responsibilities, whereas UML profiles are based on the use of stereotypes associated with a set of constraints. Nevertheless, in both cases there are distinguished design elements in a model, which either play a role of a participant of a pattern or are instances of a stereotype of a profile. Both patterns and profiles define certain structural constraints for those elements, concerning their type, properties, or relationships to other elements.

The purpose of a design pattern is to provide a systematically documented recipe for a ‘good’ solution of a design problem in the presence of certain forces. As such, a design pattern is intended for human readers rather than for machine processing. However, a design pattern implies a definition of the possible configurations of design elements that constitute the solution of the pattern. Typically, the structural solution part is given in the pattern description as a class diagram depicting a typical configuration of the solution with generic names. The associated explanations describe possible additional constraints and ways to expand the typical configuration.

Here we concentrate on the structural solution part of the pattern and ignore the informative parts such as motivation, context, forces, implementation hints, etc. An approach for specifying the structural solution part of design patterns has been the use of role models [21]. Briefly, a role model consists of a specification of configurations of roles, where each role has a type representing the kind of concrete elements that can be bound to the role. In addition, roles can be associated with formal constraints reflecting the characteristics and structural relationships of the bound elements.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Design patterns</th>
<th>UML profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td>Documenting proven design solutions</td>
<td>Capturing the characteristic of a domain</td>
</tr>
<tr>
<td>Specification</td>
<td>Informal semantics, solution structure formalized</td>
<td>Informal semantics, abstract syntax usually formalized</td>
</tr>
<tr>
<td>Form of structural specification</td>
<td>Configuration of elements with specific relationships</td>
<td>Stereotypes with associated constraints</td>
</tr>
<tr>
<td>Scope of applicability in the concrete system</td>
<td>Mostly localized effect on the system</td>
<td>Mostly to specify global system properties</td>
</tr>
<tr>
<td>Application to a system model</td>
<td>Given in terms of pattern instantiation</td>
<td>Given in terms of stereotype mapping</td>
</tr>
<tr>
<td>Mutual relationships</td>
<td>Pattern composition principles, related patterns</td>
<td>References to other stereotypes and other profiles</td>
</tr>
</tbody>
</table>

Similar to design patterns, a profile definition consists of structural specifications with associated semantic explanations. The structural specifications consist of rules stating various kinds of requirements for valid models applying the profile. Technically, a profile specification relies on the usage of stereotypes that bind the rules with actual models: when a model element is marked with a stereotype, the rules given for that stereotype apply to the element. Again, here we will concentrate only on the structural part of a profile specification.

The structural rules of a profile are typically organized according to the stereotypes: for each stereotype, the associated rules give the requirements for a legal model, possibly referring to other stereotypes. Even though a profile as a whole concerns a complete model, the individual rules concern fragments of the model marked with the appropriate stereotypes; stereotypes are used to localize the effect of a profile. Essentially, an individual profile rule implicitly defines a configuration of model elements marked with stereotypes, namely, the elements the rule applies to and can be checked against. This configuration of model elements can be viewed as an instance of the rule, in the same sense as a pattern has an instance in the model. In both cases, the instances can be overlapping.

Thus, when viewed as structural specifications of models, both patterns and profiles describe model fragments as configurations of model elements that are in some respect valid. An attractive approach to bridge patterns and profiles is therefore to specify profiles in a pattern-like style using role modeling. This is the core of our technique: in the next chapter we will introduce a tool concept that captures this kind of a configuration based on the role model idea.

Our next step is then to draw correspondences between elements of profiles, patterns, and role models. This step is important in that it provides guidelines for specifying profiles and patterns as role models (i.e. our bridging mechanism). Roles, in role modeling, can be used to represent the participants of a pattern and the stereotypes of a profile with role names holding participant names and stereotype values. The type of a role is used to represent the type of the corresponding participant in a pattern and the metaclass associated with the corresponding stereotype in a profile. Role relationships can be exploited to represent the structural constraints of pattern participants and the constraints in a profile. Furthermore, pattern instantiation guidelines and stereotype mapping rules can be expressed in role modeling as role-binding instructions. Inter-pattern relationships and relationships between stereotypes (such as stereotype inheritance), in turn, can be expressed in terms of role overlapping rules. Finally, role properties can be used to express meta-attributes assigned to stereotypes in profiles and informal notes on pattern participants.

Let us consider the unification of patterns and profiles from the viewpoint of a specific example, the J2EE platform architecture. On the one hand, well-proven design solutions for EJB-based applications have been documented as so-called EJB design patterns [7,8]. On the other hand, the domain characteristics and rules for building EJB-based systems have been specified as UML profiles [2]. We first show how the elements of a profile can systematically be mapped into the role model space. The UML profile for EJB discusses a number of logical constructs of the platform architecture. For each logical construct, four elements are discussed:

- stereotypes: the set of stereotypes corresponding to the construct;
- syntax: type rules concerning the constructs and the relationships to the other constructs;
- mapping: the rules to attach the stereotypes to concrete elements;
- constraints: the set of constraints associated with the stereotypes.
Table II. Example mapping of a UML profile for EJB to a role model.

<table>
<thead>
<tr>
<th>Element</th>
<th>Profile specification [2]</th>
<th>Role model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stereotypes</td>
<td>‘EJBPrimaryKey’</td>
<td>There is a role named EJBPrimaryKey</td>
</tr>
<tr>
<td>Syntax</td>
<td>‘The declaration of an EJB primary key class extends the declaration of a Java class...’</td>
<td>The type of role EJBPrimaryKey is a Java class</td>
</tr>
<tr>
<td>Mapping</td>
<td>‘The mapping for an EJB primary key class extends the mapping for a Java class’</td>
<td>Role EJBPrimaryKey is bound to a Java class</td>
</tr>
<tr>
<td>Constraints</td>
<td>‘The class must contain implementation for operations named hashCode and equals’</td>
<td>There are two roles of type ‘operation,’ with name constraints enforcing the bound operations to have names hashCode and equals, and containment constraints enforcing the bound operations to belong to the class bound to role EJBPrimaryKey</td>
</tr>
</tbody>
</table>

Table II presents a description of the above elements in the case of the EJB Primary Key (PK) class logical construct as documented in the profile. For every element, the table depicts the meaning of the textual specification in the role model space.

In addition, the textual specification of the stereotype discussed in the table indicates that ‘the supplier of the usage represents the EJB PK class for the EJB entity home represented by the client’. Such an informal statement implies that there is an implicit connection to the Home Interface logical construct of an EB. In the role model, such a connection can be represented by a relationship between the corresponding roles.

The constraints associated with the logical constructs of the profile are sometimes given at different levels. The constraints for the EJB Method construct, for example, are applied to all EJB Method elements. Other sets of constraints have a narrower scope of applicability, for example, the ones given for the EJB Create Method, EJB Finder Method, and the EJB Remote Method, which are also EJB methods. The different levels of constraints can conveniently be represented in a role model by having four roles corresponding to the four stereotypes, each having its own set of constraints. An overlapping relationship is then defined between each of the three latter roles and the first role. In this way, any element playing the role of the EJB Remote Method, for example, should follow the constraints associated with both the EJB Remote Method role and the EJB Method role.

In a similar manner, we could map elements of EJB design patterns (for instance, those listed in [7]) to elements of role models. Let us consider, for example, the Session Facade (SF) design pattern presented earlier. The pattern description sections named ‘Structure’ and ‘Participants and Collaborations’ [7, p. 294] list the pattern participants and their relationships. When building a role model for the pattern, a separate role can be used to represent each participant (such as SessionFacade, BusinessObject, etc.) and each relationship (such as the accesses relationship between SessionFacade and BusinessObject). Because ‘The SessionFacade interacts with multiple BusinessObject instances to provide the service’, the role for BusinessObject (of type UML Class) in the role model should allow multiple bound elements.

Besides, the role representing the accesses relationship should be constrained so that the relationship appears between instances of SessionFacade and instances of BusinessObject. Finally,
as an example of inter-pattern relationships, the ‘related patterns’ section of the SF pattern description mentions that ‘The SF is used by the Business Delegate when the client requests access to business services’. In the role model, such a relationship can be expressed as an overlapping relationship between the role for SessionFacade in the SF pattern and the role for BusinessService in the Business Delegate design pattern [7, p. 250].

3. REALIZATION OF THE APPROACH

To be useful for a tool concept, a concrete representation of role models has to be derived, applicable to both patterns and profiles. The representation format should be intuitive in that it illustrates the desired structural configuration graphically as well as possible. In the following, we describe the concrete representation format we have used in this work, and its interpretation as a role model. We also briefly discuss the tool implementation supporting this representation.

3.1. Design profiles

In the following, we present a concrete realization of a generic role model used to specify patterns and profiles called a design profile. Role modeling represents the core of our tool support for design profiles. This is presented in detail in the following subsection and the case study. A design profile defines a set of concepts and rules for a target domain constraining how application models are built in that domain. For the purpose of this work, we assume that the models are given as UML class diagrams. A design profile is specified using design forms and composition forms, explained below.

3.1.1. Design forms

Design forms are the main elements of design profiles. A design form is a class diagram that describes a role model. Syntactically, a design form is a class diagram consisting of packages, classes, attributes, operations, associations, realizations, generalizations, dependencies, and constraint notes, located in a package stereotyped as ⟨⟨Design Form⟩⟩. The role model described by a design form defines an abstract configuration of model elements and relationships between the elements. Each model element (that is, package, class, etc.) corresponds to a role in the role model. The connections and containment relationships between the model elements imply relationships between the corresponding roles. For example, if a design form contains an attribute attr within a class cls, the role model will contain an attribute role attr, a class role cls, and a relationship stating that attr is located within cls. Similarly, if a design form contains classes cls1 and cls2, with an association assoc between them, the role model will have class role cls1, class role cls2, and association role assoc, together with relationships stating that assoc is attached to cls1 and cls2.

An instance of a design form is an occurrence of the configuration in an actual model, where the concrete model elements play the roles of the role model, and the relationships between the roles match those between the concrete model elements. In addition, we allow various properties to be attached to the roles for pragmatic reasons; these properties depend on the underlying tool environment and are typically related to the way tools make use of design forms. We will not discuss properties here in more detail. As an example, a property of a class role might be the specification
of a default class element that can be generated by a tool if one is missing from the configuration specified by the design form. We will use the term design form to refer both to the description of a role model and to the role model itself; we make the distinction explicit only when it is not clear from the context.

The resulting role model, in turn, implies a part of a UML profile, the design profile. The names of the role elements in the design form automatically become stereotypes of the corresponding design profile, and the relationships of the role elements are interpreted as structural constraints of the profile. A constraint note associated with a role implies an additional constraint on the stereotype corresponding to the role.

Note that design forms resemble the way general structural solutions are described as class diagrams in the literature, for example, in the context of design patterns (e.g. [4]). Such descriptions are intended to depict a general solution: the class diagram is interpreted as a general form of the models applying a design pattern, rather than directly as a model of an application of the design pattern. This comes close to the role-model-based interpretation of design forms.

Figure 1(a) depicts a conceptual model for design forms. A design form is a collection of hierarchically organized roles. A design form is instantiated in a particular context by binding its roles to certain concrete model elements. Each role can be associated with a set of constraints expressing conditions that must be satisfied by the element(s) bound to the role. In addition to enforcing structural relationships, constraints can be used to specify general conditions expressed, for example, in OCL. Furthermore, constraints may refer to the elements bound to other roles, implying dependencies between the roles.

As model elements can be hierarchically organized (e.g. a UML class may contain a UML operation), the containment relationships of the bound elements must respect the hierarchy of the roles. A role is associated with a type, which determines the kind of model elements that can be bound to the role. A role type typically corresponds to a metaclass in the metamodel of a given notation (here UML). A cardinality is defined for each role. The cardinality of a role gives the lower and upper limits for the number of elements playing the role in an instantiation of the design form.

Figure 1. Simplified conceptual model (a) and an example EJBSessionFacade design form (b).
Figure 1(b) gives an example design form that corresponds to the EJB SF design pattern discussed earlier. The design form consists of three class roles (Client, SessionFacade, and BusinessObject) and two dependency roles. The purpose of the design form is to ensure that business objects are accessed through a facade component. Role BusinessObject has a child attribute role description and a child operation role getObjectDesc. Similarly, role SessionFacade has a child operation role getObjectDesc. A constraint note is attached to this role stating that the name of the bound UML operation should be the same as the name of the UML operation bound to role getObjectDesc in role BusinessObject, implying a dependency between the two operation roles. The constraint value is expressed using a pseudo-language that allows compact representation of constraints. In this work, we do not discuss the complete syntax of this pseudo-language. As a matter of fact, all constraints (including OCL constraints) are specified using constraint notes.

The cardinality of role BusinessObject, marked after the role name between square brackets, is ‘+’, stating that the facade may encapsulate one or more business objects. The other possible cardinality values are [1] for exactly one concrete element, [*] for any number of concrete elements including zero, and [?] for stating that the binding is optional. If no cardinality value is mentioned, then the value [1] is used.

3.1.2. Composition forms

When a design profile consisting of multiple design forms is used, we may require that some roles belonging to different design forms are to be bound to the same model element (i.e. role overlapping). We specify such a situation using composition forms. A composition form is a configuration of role elements that defines the overlapping relationships of roles present in different design forms. Syntactically, a composition form is expressed as a class diagram with classes and dependencies (stereotyped as ⟨⟨Overlap⟩⟩), located in a package stereotyped as ⟨⟨Composition Form⟩⟩. The names of the classes refer to role names in existing design forms.

Let us consider the example inter-pattern relationship discussed in Section 2 (SessionFacade and BusinessDelegate). Figure 2(a) depicts an example specification of a design form for the BusinessDelegate design pattern. Figure 2(b), in turn, shows an example composition form establishing an overlapping relationship between role BusinessService in EJBBusinessDelegate and role BusinessObject in EJBSessionFacade. The rationale for the composition form is to ensure that client requests to business services are granted through a business delegate. The composition relationship is specified using a stereotyped dependency connecting the two overlapping roles.

Figure 2. EJBBusinessDelegate design form (a) and a corresponding composition form (b).
In this case, the direction of the dependency indicates that all BusinessObject elements should also play the role BusinessService but not vice versa.

Another typical usage of composition forms is to enforce constraints that concern multiple parts of a model. Instead of attaching such constraints to each of the roles in the design profile, composition forms can be used as a more convenient and flexible mechanism. The EJB profile specification, for example, defines certain constraints for any UML subsystem that represents an EJB Enterprise Bean (stereotyped as ‘EJBEnterpriseBean’) by stating that ‘The Subsystem must have a visibility property value of public, and isLeaf and isAbstract property values of false.’ In our example design profile, these constraints are specified in the design form Constraints shown in Figure 3(a). This is in fact an example of a design form that is based on the UML profile specification for EJB, compared with the earlier design forms, which represent EJB design patterns.

For enforcing these constraints, Figure 3(b) depicts a second example of a composition form stating that the elements bound to roles BusinessObject and SessionFacade in the EJBSessionFacade design form should conform to the constraints specified in the Constraints design form. Similar to the previous example, the direction of the dependency (marking the composition rule) indicates, for example, that all SessionFacade elements are enterprise beans but not vice versa. The same constraints could be propagated to other enterprise bean roles by defining similar composition rules between role EJBEnterpriseBean and those roles.

3.2. Tool support—INARI

In order to demonstrate the design profile concept, we use a prototype tool environment called INARI. The INARI tool is built on top of Eclipse and is integrated to IBM’s Rational Software Architect (RSA) [13], which is a CASE tool for a UML-based development also built on top of Eclipse. Figure 4 depicts the INARI tool environment architecture.

A design profile is specified as a set of UML packages containing design forms and composition forms, as presented in the previous subsection. The tool then validates the design profile and automatically transforms its design forms into an INARI role model specification. As INARI directly supports the role types in design forms, the translation is straightforward. All the constraints implied by the structural relationships of the roles in the design form (such as containment relationships between classes and attributes or connection relationships between associations and classes) have also a specific constraint type in INARI, making the translation easy. After the translation, the tool displays a property sheet that can be used to attach further specifications to the roles, like the specifications of model elements used in the generative mode (see below). Additional constraints
given in constraint notes are not translated automatically from design forms, but they have to be written manually through the property sheet in the current tool implementation. INARI supports the use of OCL in the constraint expressions.

In the INARI tool, the role model specification for a design form is used as input for a binding engine that performs instantiations of the role model. When creating design form instances, the specification of a design form can be applied in two modes: the user can either apply an automatic binding tool that binds the roles to existing model elements or he/she can instantiate the design form under the guidance of the tool, step by step.

The automatic binding tool applies the algorithm presented in \[24\], making use of the existing stereotype instances in a model as hints of role bindings. For instance, a UML class stereotyped as $S$ is a binding candidate for a role named $S$. The automatic binding tool also uses structural information (e.g. containment relationships) when detecting instances of a design form. The tool is able to recognize partial instances of design forms as well, allowing the user to switch to the guided mode and to complete the instance.

The instantiation of a design form in the guided mode follows the idea of a task-driven model generation discussed in \[12\]. The basic idea is that a model is created under the control of the binding engine: the binding engine displays a task for each unbound role that can be bound in the current situation, that is, for each unbound role that has not yet reached its maximum number of bindings, and that does not depend on other unbound roles. In that case, the tool can automatically generate a model element playing the role, as all the information it depends on is available. It is also possible to manually bind an unbound role to an existing model element. When a task is performed, the generated (or located) model element is stereotyped according to the corresponding role name.

The INARI tool falls short in its support to the automatic transformation of composition forms; these have to be manually converted to role overlapping relationships using the role model editor in INARI. In the next section, we describe in more detail how design profiles are specified and applied in the context of the EJB framework.
4. CASE STUDY

We have applied our approach to specify design rules of the EJB platform and to produce an experimental environment for designing EJB-based applications. The proposed EJB design profile is based on the specifications given in [2,7,8,25]. To demonstrate the applicability of our approach, we have selected a number of frequently used EJB design patterns. As for the EJB UML profile, we have considered those parts that are relevant to the selected design patterns.

4.1. EJB design profile

Our design profile consists of six design forms and two composition forms, see Figure 5. Three of the design forms follow the specification of the UML profile for EJB [2,25] and are used to model two main EJB components according to the platform rules: a container-managed local EB (EJBLocalEntityBeanForm) and a stateless session bean (SB) (EJBSessionBeanForm). The constraints defined for the two EJB components are encapsulated in the third design form named EJBDesignConstraints.

The other three design forms represent widely recognized EJB design patterns [7,8] and are used to enforce best practices in the EJB-based design: ValueObjectForm (an object used for efficient data

Figure 5. Example forms.
transfer), *PrimaryKeyForm* (a class to be used as the PK for locating EBs), and *SessionFacadeForm* (an SB encapsulating business components and entities).

As an example form, the top left part of Figure 5 shows a role structure of *EJBLocalEntityBeanForm* depicting the specification of an EJB local EB. For brevity, we have omitted details related, for example, to role constraints. In a similar manner, the specification of *EJBSessionBeanForm* is given in the bottom left of the figure.

The EJB platform specification [25] defines several conventions for EJB methods. For example, the home method *create*, defined in the local home interface, must start with the prefix ‘create’ and must have a corresponding bean method in the implementation class. In addition to the home method name, the implementation method should also be prefixed with ‘ejb’. Using design forms, these naming rules can conveniently be expressed by two naming constraints, expressed here using a constraint pseudo-language:

\[
[EJBCreateMethod.naming = RegExp(create.*]) \text{ and } [EJBCreate.name = ejb(EJBLocalEntityHome.EJBCreateMethod.name)].
\]

The latter constraint refers to a role instance of *EJBCreateMethod* and implies that for each method instance a corresponding method in the implementation class must be generated. In the following subsection, we show the way the first constraint is represented in the INARI tool environment.

Another example design form named *SessionFacadeForm* is given in the mid top part of Figure 5. The design form is to enforce the use of a facade when interacting with each business or entity component. As stated earlier, our EJB design profile also includes *PrimaryKeyForm* and *ValueObjectForm* design forms defining common design practices used in EJB-based applications. The specification of the two forms is shown in the right bottom part of Figure 5. *PrimaryKeyForm* consists of two class roles: *EJBHomeInterface* and *PrimaryKeyClass* connected with a dependency role. Role *PrimaryKeyClass* has an attribute role *PrimaryKeyField* (with cardinality value ‘+’) representing the fields that constitute the bean PK. Similarly, *ValueObjectForm* includes *EJBLocalRemote* and *ValueObject* class roles. Role *RemoteInterface* consists of two operation roles representing getter and setter methods for the value object (VO).

An example composition form named *EJBSessionFacadeForm* is given in the upper right part of Figure 5. The composition form defines composition rules to be applied when encapsulating business entities with facade components. The example rules state that role *SessionFacade* must be bound to a stateless SB and that any component bound to role *BusinessEntity* should also be a local EB.

Following the discussion at the end of Section 2, the right lower part of Figure 5 depicts another example composition form named *EJBCommonForm*. The form is to enforce constraints common to all EJB methods on home create methods. In this case, the constraints are defined in the *EJBDesignConstraints* design form shown in the mid bottom part of Figure 5. An overlapping relationship is made between roles *EJBCreateMethod* and *EJBMethd*.

### 4.2. Specifying EJB design profile in the INARI environment

Figure 6 shows the way design forms are specified in the INARI tool environment. A design form is created using the graphical user interface (GUI) in the bottom left of the figure (design form view). As an example, a specification of the *EJBLocalEntityBeanForm* design form introduced earlier
is created. The UML specification of the design form is shown in the upper right part of the figure (RSA UML editor).

The role model specification for the example design form is shown in the bottom right part of the figure (concern explorer). This view represents a property sheet for a design form with the left part depicting the role/constraint hierarchy and the right part showing the properties of a selected role/constraint. The property sheet can be used, for example, to introduce new constraints not otherwise present in the original UML specification of the form. As an example, a name constraint has been introduced for the \textit{EJBCreateMethod} role as discussed in the previous subsection. The value of the constraint is shown in the bottom of the property list (`create`). The design form is added to the project workspace, where informal specification can be added as shown in the top left part of the figure (architecture).

As mentioned earlier, the current implementation of INARI does not support the transformation of composition forms. Composition rules (i.e. role overlapping relationships) have to be added...
manually to the role model. The composition mechanism, which is discussed in detail in [26], is illustrated in the following section.

4.3. Usage scenarios of the example EJB design profile

Using design profiles, generative model development and model validation activities are typically carried out in different combinations, with the possibility for designers to freely edit the model under development any time during the usage of the profile. The usage mode of a design form does not depend on the origin of the design rules captured in that design form. For example, the EJBLocalEntityBeanForm design form could be used to generate new model fragments (i.e. generative usage of UML profiles), whereas the ValueObject design form could be applied to validate an existing design (i.e. validational usage of design patterns).

The elements of a design profile (design forms and composition forms) can be applied differently during different phases of the model development process. In order to illustrate these different combinations in the case of the EJB design profile, let us consider the following design forms: entity bean (EB), SB, SF, VO, and PK.

Table III provides a number of EJB-related design tasks and the way the proposed EJB profile can be used to support these tasks. The design tasks reflect possible combinations of generation, validation, and editing actions. In some scenarios (e.g. third design task in Table III), the same design form is used for both model generation and validation. In other scenarios (e.g. fourth design task in Table III), different design forms are used. The first two scenarios represent the simplest of all. The first is to use design forms in a pure generative way, whereas the second is for model validation. The rest of the scenarios use the first two in different combinations.

Figure 7 depicts a screen of the INARI tool environment using the example EJB design profile. The profile has been applied to build an EJB-based design model for a course management system. In this regard, two container-managed EBs (for course and staff entities) have been developed (first design task in Table III). The model elements are generated by following a task list corresponding to the roles of the design form (more explanation on the tasking GUI is given later in this section). The details of the classes and their relationships have been generated correctly. The model elements are stereotyped according to their corresponding role names in the design form. Some of the model elements follow strict naming constraints (e.g. ejbActivate, ejbCreate, etc.): in this case the constraint value is equivalent to the role name (or the stereotype value). Other elements such as courseNumber have no such naming constraints.

As shown in the top right part of the figure (i.e. design form instances), access to the beans is granted through a facade component. This is achieved by instantiating the design forms SessionFacadeForm and EJBLocalEntityBeanForm composed using the specification of the EJBSessionFacadeForm composition form. The deployment information of the forms is shown in the top left part of Figure 7 (i.e. project view). The composition rules, between the course entity and the facade, for example, are defined in the CourseEntityBean node and are specified manually as illustrated in Figure 8.

Having generated the EBs correctly according to the EJB framework specification, let us assume that the designer wants to apply EJB best practices by providing VO support to EBs. For this purpose, the designer should check which of the EBs use VOs and whether the VOs have been correctly modeled (second design task in Table III). This task can be achieved in the INARI environment by automatically binding the VO design form to the example model.
Table III. Different usage scenarios of design profiles.

<table>
<thead>
<tr>
<th>Design task</th>
<th>Profile usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to construct new entity beans?</td>
<td>Create new instances of $EB$ (generation)</td>
</tr>
<tr>
<td>How to check which entity beans have value objects?</td>
<td>Automatically bind $VO$ to the model (validation)</td>
</tr>
<tr>
<td>How to construct new session beans? Assuming that the model has been edited, how to ensure that the editing actions did not break the rules of a session bean?</td>
<td>Create new instances of $SB$ (generation). After the model is edited (editing), automatically bind $SB$ to the modified model (validation)</td>
</tr>
<tr>
<td>How to check that each entity bean has a primary key class, assuming that the entity beans have been generated by the environment and that the primary key classes have been added manually?</td>
<td>Create new instances of $EB$ (generation). After the primary key classes are added manually (editing), automatically bind $PK$ to the model (validation)</td>
</tr>
<tr>
<td>Assuming that business entities (or business components) have been constructed, how to encapsulate these using facade components?</td>
<td>Automatically bind $SF$ to the model (validation). As a result, $SF$ gets partially bound (hooked into the business/entity components). Follow the task list of $SF$ instances for generating the facade components (generation)</td>
</tr>
<tr>
<td>How to find all entity beans in a model and how to define value objects for a selected subset of those?</td>
<td>Automatically bind $EB$ to the model (validation). Then create new instances of $VO$ for your target beans (generation)</td>
</tr>
<tr>
<td>How to generate all the required elements of entity beans encapsulated by session facades, assuming that the encapsulated components have been added by hand?</td>
<td>Automatically bind $SF$ to the model (validation). Assuming that the encapsulated entity components are added by hand (editing), create new instances of $EB$ for those components (generation)</td>
</tr>
<tr>
<td>How to make sure that all required elements of entity beans are not accidentally removed during manual editing of the model? If so, how to restore them?</td>
<td>Automatically bind $EB$ to the model (validation). If a constraint becomes violated or a role binding becomes unresolved during editing actions (editing), repair the violations using the broken instance of $EB$ (generation)</td>
</tr>
<tr>
<td>How to construct new entity beans and how to check that these beans are encapsulated by session facades?</td>
<td>Create new instances of $EB$ (generation). Then automatically bind $SF$ to the model (validation)</td>
</tr>
</tbody>
</table>

The role structure of the VO design form is shown in the right bottom part of Figure 7. On the basis of the structural relationships (for example, the role hierarchy) and the constraints in the design form (for example, the stereotype value ‘ValueObject’), all instances of the form are detected and highlighted by the tool in the model. The top left part of Figure 7 shows two instances of the VO design form. One of the two instances is complete (class $Course$ and the bean remote interface $CourseLocalRemote$). The other instance is incomplete as the binding tool could find a match for the bean remote interface ($StaffLocalRemote$) but no match for the VO class. An incomplete design form instance is indicated in the tool by a red dot on the design form instance.

Assuming that the designer wants to complete the instance, the missing elements can be added using the generative capabilities of the INARI tool. The bottom part of Figure 7 shows the task(s) corresponding to the missing element(s). For every task, an informal task description is available. In order to enhance the usability of the guided design form instantiation, the tool provides a direct-tasking environment where the existence of undone tasks is directly indicated in the elements.
Figure 7. Applying design forms in the INARI environment.

Figure 8. Composing SessionFacadeForm and EJBLocalEntityBeanForm design forms.
of a model’s diagram. This feature runs in parallel with the user modeling activity and provides context-based mechanisms for helping a model to grow according to the specification of the design form that is being instantiated.

The direct-tasking process navigates through the set of available tasks in order to determine which tasks are depending on each model element. For instance, classes that have related tasks to be performed become marked (highlighted). When selecting a marked class, the user is given the undone tasks corresponding to that class. Considering the example shown in Figure 7, the class StaffLocalremote is marked in the diagram, indicating that there are related tasks to be performed. When selecting the class, the tasks for providing a getter and setter operation for the VO are given (based on the specification of the design form). In the case of Figure 7, however, the user has selected the top context and the tasking tool displays the task for providing a VO class for the staff EB. The designer can perform the task by clicking the ‘Generate’ button, a class is then generated according to the corresponding role specification and a binding between the role and the class is established. A video recording of this scenario is available in the INARI project Web site [23].

5. EXPERIENCES AND EVALUATION

At this point, the profile-based EJB development environment presented in this work has not been applied in an industrial context. However, we have applied an earlier version of the INARI environment and its support for role-model-based design rules to other case studies as well, in cooperation with our industrial partners. Although these case studies did not make direct use of the design profile concept as the specification means of role models, they illustrate the strength and versatility of the underlying role modeling approach. All the case studies have required the building of totally new role models to express the design rules or other fragmentary model structures for the target domains. This requires mining the design rules from various design documents and communications with experts, and specifying them as role models. In some cases, the design rules are readily available in a well-defined form, as in the EJB case study; in such cases the rules can be transformed into role models fairly quickly. For example, it took less than two days to create the EJB design profile discussed in the previous section, including the time required for getting familiar with the EJB profile.

The role modeling approach presented in this paper is best applicable to specify design rules for building applications on top of well-defined software platforms. In this regard, the patterns and profiles represent the best practices and rules implied by the specialization interface of the platform. We have, for example, used INARI and its role modeling approach to build a design environment for the Series 60 Application Framework, which is a software platform on top of the Symbian operation system targeted for mobile devices [12]. In this case, the design rules represented best practices like enforcing the Symbian application architecture and global rules such as the naming conventions in Symbian.

In another industrial case study related to software platforms, we have used our approach to structure and document the specialization interface of the GUI part of a Nokia software platform [27]. The platform is used to create different types of network and service management solutions for managing networks and network elements. In this example, we were given semi-formal design documents that we used to mine the rules for building applications on top of the platform. These rules were transformed into role models and were used to guide application developers in creating
application models. This facility was used to train novices to use the platform in a learning-by-doing manner.

As a third example, we applied our approach to build a guided configuration management environment for a harvester condition monitoring platform [28]. As the configuration descriptions were given in XML, we extended INARI with XML-specific role types. In this case the design rules pertained to the allowed ways of giving the XML configuration descriptions. We exploited the tool in a generative way, so that missing elements of the configuration description could be produced automatically whenever they were implied by existing elements. The environment has been used for evaluation purposes in the industry. However, even though the idea works fairly well also at the XML level, it seems that this kind of guidance support offered at the low abstraction level of XML configuration descriptions is considered less valuable than similar support at the modeling level.

The generative capability of our approach ensures that the produced design conforms to the design rules captured in the role models. These rules can represent positive rules indicating what the designer should do or negative rules indicating what the designer should not do. In the latter case, we assume that the negative rules can be expressed as OCL constraints that can be attached to the roles. These constraints get evaluated when the roles are bound to the concrete design elements. However, it seems that the majority of the design rules are in fact positive rules. The constraints are evaluated every time the model is edited. This quality assurance mechanism ensures that the manual changes will not break any of the design rules specified. This benefit applies to both experienced designers and novices.

In the INARI environment, it is possible to indicate in the constraint expression whether the evaluation context of the constraint is the global context (i.e. the whole model) or just the concrete element(s) bound to the role. The binding information is used to get context information needed for evaluating the constraint. Still, the tool treats all the constraints alike, irrespective of their evaluation context. An obvious limitation in the case of global evaluation context, however, is a situation where the corresponding role does not get bound. In this case, there is no mechanism to evaluate that constraint.

As a pattern tool, the INARI environment falls short in its support to the informal (non-structural) aspects of design patterns such as the forces, consequences, and tradeoff sections of pattern descriptions. Such a limitation is partially overcome by using the documentation mechanism of the INARI environment (e.g. role and task descriptions in the GUI). In general, such documentation capability is very important for guided development environments. However, this seems to be a common shortcoming of environments with generative tool support. As the INARI approach focuses on design patterns whose solution part can be expressed in terms of concrete elements, design profiles cannot be used (at least in a straightforward way) to specify patterns of high abstraction level such as process patterns [29] or analysis patterns [30].

On the profile side, our design profiles cannot represent certain informal profile elements such as the rationale section of the example EJB profile document. Furthermore, our approach is more suitable for profiles emphasizing structural design rules than for profiles whose main purpose is to define a set of concepts (stereotypes) with certain semantics, for example, for testing [31] or code generation [14] purposes. In this sense, the EJB profile we have used as a case study is particularly amenable for this approach. Depending on the amount of structural specifications in a profile, our approach offers a varying degree of support for a designer.

In this work, we have discussed design profiles in the context of UML. However, our approach can be applied to other formalisms and notations as well. Currently, the INARI environment supports
elements of UML, Java, and XML. This allows us to specify design rules that crosscut multiple levels of abstraction. For example, we have exploited this tool property to achieve a development environment for a Nokia GUI platform where variability is managed across multiple artifacts [32]. The variability information embeds design rules that are represented in feature models (UML), design documents (UML), implementation (Java), and deployment information (XML).

It is hard to come up with specific metrics to evaluate the INARI approach, partly because the usage scenarios are so versatile. In general, it seems reasonable to assume that the generative support provided by the approach increases the productivity of especially novices, as the tool remembers and enforces the design rules on behalf of the designer. The generative support is even more powerful when extended up to the code level, because large portions of the code are implied by the model. In an earlier work on the J2EE framework [33], we have shown that up to 60% of the total lines of code was automatically generated by the environment. Most of the generated source code represented the program elements required for the J2EE component environment, the overall code structure of the used patterns, and the way these patterns are interacting. In an experiment [33], we found that manually it could take up to 20 h to build a given J2EE application code from scratch. Using our approach based on role modeling, the same code was generated within a couple of hours. Approximately, one-third of the development time was spent using the environment for automatic code generation, whereas the rest was used to provide custom business code and implement the user interface.

6. RELATED WORK

A wide spectrum of UML profiles has been developed for different purposes and for different domains. A list of those adopted by OMG is available on OMG’s web site. In most of these profiles, constraints related to the use of stereotypes are given in a textual and not a programmatically interpretable way. This limits their efficient use in tools supporting generative approaches or model validation.

Ober et al. [15] discuss the OMEGA profile to support real-time modeling. The profile also includes timed properties that are introduced as lightweight extensions to UML 1.x. These extensions are comparable with corresponding features included in UML 2.0. A formal validation of behavioral models against OMEGA profile is enabled by model checking and simulation tools built for formal IF language [34]. The model simulation and automatic test generation tools correspond to those offered by I-Logix’s Rhapsody tool [35].

Koskinen et al. [36] propose behavioral profiles, given as UML2 sequence and class diagrams, for capturing architecturally significant behavioral rules and tool support for validating call sequences against a given set of profiles. Behavioral profiles can, for example, be used to capture essential rules related to software API usage or application framework specialization, thus supporting software reuse. In our current tool environment, validation of models, which in our case are structural, is supported by automatically binding selected design forms to an existing model. The tool then informs the user about any violations detected by generating new tasks.

UML profiles are also used to support code generation in a number of tools. Meunier et al. [14] propose the HIDOORS profile to support real-time modeling and code generation; Java code is generated from a real-time UML model automatically, based on the HIDOORS profile. In addition, RSA [13] supports a profile-guided code generation. In contrast to this kind of support, our approach
allows the generative specifications to be associated with certain configurations of model elements, rather than with single stereotypes. In addition, we support the generation of models as well as the code. Finally, our approach led to a step-by-step generation, which can be followed by a designer, rather than to a one-shot generation.

The relationship between profiles and design patterns is also studied, for example, by Dong and Yang [37]. In their approach, UML profiles are used to support visualization of design patterns in UML models. The stereotypes and tagged values of a UML profile are attached to model elements to represent the role that the model element plays in a design pattern. Their approach focuses on pattern visualization and does not consider design rule enforcement nor model validation.

Various kinds of tool support for deploying design patterns and for specializing OO frameworks have been proposed over the years [13,16,18,38]. For example, Krasner and Pope [39] proposed a cookbook approach. A cookbook typically contains information descriptions and guidance on framework specialization. The different functionalities supported by a framework can be implemented in the specialized application by following different recipes. A recipe in a cookbook approach is comparable with the support for user interactions during the generative use of profiles in our approach. In contrast to cookbook approaches, our approach relies on well-defined design profile descriptions. Ortigosa and Campo [38] introduce another approach providing guidance for framework specialization called SmartBooks. In a sense, SmartBooks make cookbooks more dynamic: a tool can provide the user choices of different high-level activities, based on the framework specialization rules. The user assistance in [38] thus resembles the user assistance in our approach. These tools typically work on the code level and generate the whole pattern instance based on predefined rules. Moreover, they are used for specializing application frameworks in a pattern-based manner. We, instead, currently work with models expressed in UML and focus on using design profiles for both generative and validation purposes, thus integrating the profile and pattern concepts.

In addition to tool support for pattern instantiation, various formalizations for patterns have been proposed [11,19,21,40], often in the context of specifying the hot spots of frameworks. Eden et al., for example, propose a logic-based approach and language LePUS [18] for specifying design patterns, class libraries, and object-oriented frameworks, and for reasoning about them. LePUS specifications can also be expressed using semantically equivalent graphical diagrams to support the user in understanding the pattern structures. Mikkonen [19] proposes a way to formalize temporal behaviors of patterns using the disco method [41]. Considering the context and utilization of patterns, specifications of complex systems can be composed using property-preserving refinements. Riehle [21], in turn, described patterns as role models, including a notion of a composition constraint. The role models can be visualized graphically as role diagrams. In contrast to these approaches, we use UML as the graphical notation and do not aim at formalizations of patterns.

The MDE-based (model-driven engineering) [42] approach presented in this paper has been carried out in the domain of EJB. Built on top of a well-defined UML profile and pattern language, our example design profile can be regarded as a domain-specific language (DSL) [43] for the EJB platform. The purpose of our DSL is to manage the complexity of the EJB-based development by generating valid design models (generative usage of DSL) and validating existing design models (validational usage of DSL). Indeed, existing meta-case tools such as MetaEdit+ [44] and GME [45] can be used to build an environment similar to ours. These tools offer a framework for defining concepts, properties, and rules of well-defined domains. Compared with these advanced tools, our
approach aimed at a more lightweight generative mechanism and enhanced validational support for DSL tools, integrated with a conventional UML modeling environment.

7. CONCLUDING REMARKS

In this paper, we have studied the practical feasibility of integrating tool support for UML profiles and design patterns. We argued that such an integration can be realized using role modeling. We then presented a concrete role-model-based approach for the structural specification of UML profiles and design patterns using so-called design profiles. We have shown that design profiles can be supported by tools that are able to both guide the designer in creating legal models and check existing models against the profiles. We have applied our approach to build a customized modeling environment for EJB-based applications and analyzed the various usage scenarios that can be realized using our approach.

A major benefit of our design form notation is its intuitiveness: a design form depicts the involved elements and their required relationships in a graphical form that is easy to grasp and associate with models. Our approach offers a UML-based modeling technique for the structural aspects of profiles, with validating and generative tool support. So far this kind of facility has been missing in UML modeling environments. As the design form notation is based on UML, we could exploit an existing UML modeling environment, extending it with specific design form support. A drawback of this approach is that there are rules that cannot be given in the graphical form, but require textual (e.g. OCL) expressions to be associated with the design form. Although the textual expressions can be evaluated in model validation, they cannot be taken into account in generative tool support.

The use of role modeling as the underlying paradigm makes the tool concept very flexible and generic: the instances of profile rules are represented separately from the actual model as a role structure, which can be freely manipulated without disturbing the target model. On the one hand, this makes it relatively easy to integrate this kind of tool with any modeling environment that provides a reasonable API to access the model and editing events; on the other hand, the role structures can be used for various purposes other than profiles, too. In general, roles can be seen as a general marking mechanism for models, allowing certain elements to be marked as parts of some larger logical entity. For example, we have implemented a separate tool that is able to perform set operations on model fragments marked by role bindings, denoting certain concerns in the model. This kind of concern manipulation tool is very different from a profile tool, yet they both exploit the same role structuring mechanisms.

Design forms can be viewed as a higher-level structuring device for UML models. In this sense, our approach represents a step toward structure-oriented editors of UML models, providing assistance for the creation of valid models in the style of modern structure-sensitive source code editors. We anticipate that various kinds of structure-oriented processing tools become essential parts of future modeling environments.

ACKNOWLEDGEMENTS

This study is supported by the Finnish Funding Agency for Technology and Innovation (TEKES, projects Inari 40226/04 and MoDES 40081/07), Nokia, Nokia Siemens Networks, Nokia Foundation, TietoEnator, Plenware, Solita, John Deere, and the graduate schools TISE and SoSE. The authors would like to thank James Noble and the anonymous reviewers for their fruitful comments on an earlier version of this paper.
REFERENCES