

Design-Space Exploration through Constraint-Based Model-Transformation

Bernhard Schätz*, Florian Hölzl †, Torbjörn Lundkvist ‡

*fortiss gGmbH, Guerickestr. 25, D-80805 München, Email: email: schaetz@fortiss.org

†Institut für Informatik, Technische Universität München, Boltzmannstraße 3, D-85748 Garching,
Email: hoelzlf@in.tum.de

‡Department of Information Technologies, Abo Akademi University, Joukahaisenkatu 3–5 A, FIN-20520 Turku,
Email: torbjorn.lundkvist@abo.fi

Abstract—Many design steps during system development - like functional partitioning, refactoring of the architecture, or the mapping to the platform - can be understood as an exploration of the solution space. Each development step is characterized by design constraints, limiting the possible solution space. By using model transformations based on a declarative, relational approach, these constraints can be formalized as transformation rules guiding a mechanized exploration of possible design alternatives. The approach is demonstrated for the (semi-)automatic, incremental deployment of logical architectures to hardware platforms.

Keywords-Model transformation; rule-based; declarative; design-space exploration

I. INTRODUCTION

When developing a (software) system from the initial requirements to the final system, the ideal development process corresponds to a sequence of refinement steps. Each refinement step moving from an abstract model – e.g., the description of the logical architecture of a system consisting of communicating (software) components – to a concrete model – e.g., the description of the technical architecture of a system consisting of communicating control units – generally requires to fix one or more design decisions, each enriching the model under development. These design steps generally involve decision making by the software engineer in form of a search problem. Typically, these decisions are based on constraints, limiting the set of acceptable solutions in the search space, as well as experience, guiding the engineer, and not done fully automatically. Additionally, often there are multiple ways to fix a decision, leaving the system developer with a choice of suitable solutions, possibly parameterized by additional objectives.

The refinement steps from a more abstract to a more concrete model often can be formalized as transformation relations between these two kinds of models, possibly parameterized to reflect additional objectives. By mechanizing these transformation relations, the concrete models can be automatically transformed from the abstract models. Especially in the context of model-driven approaches, model transformation techniques have been developed to support the automatic generation of those transformed models. Currently, most of those approaches have concentrated mainly

on transformations of models to obtain a single specific transformed model from a given one, by repeatedly applying a set of transformation rules.

However, the search problem represented by a design-space exploration step requires to apply rules to generate a potential solution, and to check whether the generated solution fulfills the constraints characterizing acceptable solutions. Such a search-and-check approach requires to take back applied transformations rules in form of backtracking. Furthermore, since in general more than one possible solution exists, for design space exploration this backtracking mechanism must allow to repeatedly and successively generate solutions and present them to the engineer for selection.

This contribution shows how an approach supporting the definition of loose transformations, i.e., transformations with different possible solutions, can be used to mechanize an interactive and incremental development process. The remainder of this contribution is organized as follows: Section II introduces a typical example of an exploration step in the design process of embedded systems: the deployment step, i.e., the mapping of logical components and channels to technical units and links, respecting restrictions concerning the load provided by units as well as links and required by components as well as channels. Furthermore, the section introduces the main characteristics of an exploration process of the design space and relates them to the core features of a declarative, relational descriptions of model transformations. Section III introduces a mechanism supporting the definition of loose transformation relations: a formalism for a declarative, rule-based transformations, based on a term-structure representation of conceptual models and suitable Prolog-predicates for the construction and deconstruction of these term-structures. Section IV demonstrates the application of the mechanism to the deployment problem: an interactive and iterative support for the exploration of the deployment possibilities based on a direct formalization of the constraints of the solution space. Furthermore, the section discusses extensions to improve the efficiency of the transformation as well as its incremental application. Section V finally summarizes the central aspects of the present approach and compares it to related work.

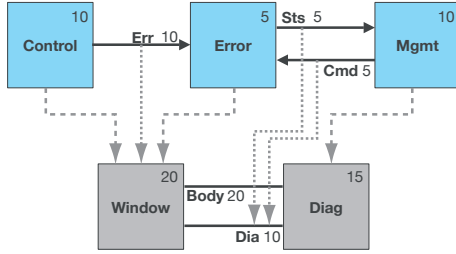


Figure 1. Example: Deployment of Control, Error, and Mgmt

II. DESIGN SPACE EXPLORATION

While most development activities can be understood as refinement steps from abstract to concrete models, aside from rather mechanical activities, these development steps cannot be fully automated, because in general there is not only one possible refinement step to be taken. The engineer has to identify a suitable refinement in case there is more than one possible solution, forcing him to take design decisions. However, often the set of suitable solutions can be characterized by constraints. Examples for development steps with such implicitly defined solutions are the mapping of software components to hardware units or the identification of minimal sets for software components required for self-contained system releases.

In such cases, the engineer can be supported in his search for the right solution. This section demonstrates that design space exploration – i.e., the construction of different possible alternatives within a constrained solution space – is a problem accessible to model transformations based on declarative, relational transformation rules. Such an approach can be advantageous, since for domain experts, in general, characterizing the set of possible solutions is more straight-forward than defining a suitable search strategy. As such a rule-based, declarative characterization of this set of solutions can be directly evaluated by an interpreter, this characterization can be immediately used as the definition of a suitable search strategy.

A. Example: Component Deployment

Component deployment of soft real-time systems, commonly found in the model-based development of embedded software, deals with the *resource-constrained mapping* of a set of logical components connected by channels to distributed electronic control units connected by links. This mapping is constrained to be *resource-consistent*, i.e., the (average) load required by a component or channel is less than the load provided by a control unit or a link, in case the former are mapped to latter. Computational load is required by components and provided by units; communication load is required by channels and provided by links.

A mapping is called a *complete deployment* if all components and channels are mapped. For a complete mapping,

a component is always mapped to a unit; a channel is mapped to a link only in case the corresponding components connected by the channel reside on different units; otherwise unit-internal communication can be used. A mapping is called a *consistent deployment* if the required loads of the mapped components and channels do not exceed the provided loads of the units and links they are mapped to.

Figure 1 depicts such a deployment for a power window control functionality including error management, mapped to automotive control units for the window movement and diagnosis. The upper half shows the logical components Control, Error, and Mgmt, with channel Err from Control to Error, and channels Sts and Cmd between Error and Mgmt. The *required* computational and communication load of components and channels is indicated by the adjoined integer numbers. The lower half shows the control units Window and Diag, connected by links Body and Dia. Again, the adjoined integer numbers indicate the corresponding *provided maximum* computational and communication load.

Finally, the deployment is shown by arrows from the components and channels to the units and links, resp. Components Control and Error are mapped to unit Window, while component Mgmt is mapped to Diag. Similarly, channels Sts and Cmd are mapped to link Dia. Since channel Err connects two components mapped to the same unit, it can also be mapped to the same unit. The deployment depicted in Figure 1 is both complete and consistent. E.g., the load required by Control and Error – in total 15 – does not exceed the load of 20 provided by Window. Similarly, the required load of Sts and Cmd – in total 10 – does not exceed the load of 10 provided by Dia.

While in total up to several hundreds of components are deployed to several dozens of units, in practice it is sufficient to tackle smaller deployments concerning around 10-20 components including connecting channels and up to four units with their connected links, since components and units are grouped to domains like motor management, chassis electronics, or infotainment. Nevertheless, even for those smaller deployment problems, it is necessary to find a few resource-consistent deployments out of several millions of mappings from components/channels to units/links.

B. Approach

Design space exploration consists of finding a solution from the set of possible designs, respecting some given design constraints. In general, these characteristics of a possible solution in the exploration space can be described in a declarative fashion rather straightforwardly. E.g., as discussed in the previous subsection, a deployment can be easily described as a complete mapping from components to units as well as channels to busses, consistent concerning the provided and required average computation and communication loads.

Mechanized exploration support therefore consists in providing means to automate the systematic search of the design space for those complete and consistent solutions. For that purpose, the declarative description of the design constraints must be turned into an operational version guiding the search process.

To support an effective and efficient process of design space exploration, these operational version should also fulfill additional properties:

- The approach must support an interactive process; i.e., if there are several different solutions to the design problem – e.g., different mappings of components to units – all possible solutions should be presented to the engineer, to support him in making a selection.
- The approach must support an incremental process; i.e., if design constraints are given – e.g., in terms of a partial deployment – all generated solutions should be extensions of these partial solutions.

In [1], an approach is introduced that allows the formalization of (model) transformations by characterizing the properties of a model before and after the transformation in a relational, declarative fashion. By interpreting a model as a structured term, logic programming using Prolog can be used to execute this declarative representation of transformation rules. Since a solution within the design space can be interpreted as a characterization of the model after implementing the corresponding design decision, the exploration process itself can be understood as a transformation step. Of course, this step is generally under-specified and therefore has different possible solutions. Nevertheless, due to the executable interpretation of such a formalization, this approach can be used to automate the search process.

In the remainder, we show how this mechanism can be used to turn a declarative description of design constraints into an automated process supporting the interactive and incremental exploration of the design space. By using a rule-based relational formalization of these constraints, and interpreting them as transformation relation, possible solutions within the design space are generated. Due to the relational style of the formalization and the backtracking mechanism provided by the framework, the different possible solutions can be easily generated. Since the relational approach allows characterizing a *set of possible solutions*, this generation mechanism can be used to incrementally and interactively generate these different possible solutions.

Finally, since the model before the transformation step may already contain elements corresponding to a partial solution, these constraints are directly incorporated in the search process, supporting an incremental approach.

III. MODELS AND TRANSFORMATIONS

To construct formalized descriptions of a system under development, a ‘syntactic description’ or *conceptual (domain) model* is needed. This conceptual model characterizes all

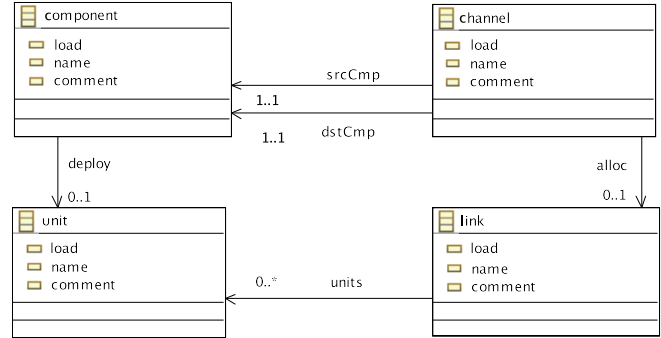


Figure 2. Simplified Conceptual Domain Model for Component Deployment

possible system models built from the *modeling concepts and their relations* used to construct a description of a system; typically, class diagrams are used to describe them.¹

Figure 2 shows the conceptual elements and their relations used to describe the logical and technical architectural structure of a system. These concepts are reflected in the techniques used to model a system. In the following, a *formalization of conceptual domain models and conceptual product models* based on relations is given as well as their representation using Prolog style term-structures and support to manipulate this representation.

For sake of brevity, while supported by the approach, the use of inheritance is not discussed in this section. E.g., the concepts *component*, *channel*, *unit*, and *link* can all be specialized from a common concept defining the attributes *name*, *comment*, and *load*.

A. Formalization

A *conceptual domain model* gives an interpretation for syntactical descriptions like in Figure 2. It defines the primitives used to describe a system: *concepts* characterizing unique entities used to describe a system, like *component* or *channel* to define the components and channels of the description of the logical architecture of a system; *attributes* characterizing properties, like *name* or *load* to define the name of a component and its required average computational load. Concepts and attributes form the conceptual universe, consisting of a collection of infinite sets of conceptual entities, and a collection of – finite or infinite – sets of attribute values. In case of the deployment, examples for suitable sets of conceptual entities are $CompId = \{comp_1, comp_2, \dots\}$, and $ChanId = \{chan_1, chan_2, \dots\}$; typical examples for set of attribute values are $CompName = \{\text{‘Control’}, \text{‘Error’}, \dots\}$ or $CompLoad = \{0, 1, 2, \dots\}$.

Based on these primitives, the *conceptual domain model* defines elements corresponding to objects used to model

¹The class diagram-like definition of a conceptual domain model is generally called *meta model*.

a system, like *Component*, or *Channel* as the components and channels of the description of the logical architecture of a system; and relations corresponding to dependencies between the elements, like *srcCmp* or *dstCmp* as the source or destination component of a channel.

The conceptual domain introduces element relations between conceptual entities and attribute values, and (binary) association relations between conceptual entities. In case of the conceptual domain model as provided in Figure 2², examples for element relations are $Component = CompId \times CompName \times CompLoad$ with values $\{(comp_1, 'Control', 5), (comp_2, 'Error', 5), \dots\}$ or $Channel = ChanId \times ChanName \times ChanLoad$ with values $\{(channel_1, 'Err', 10), (channel_2, 'Sts', 5), \dots\}$; examples for association relations are $srcCmp = ChanId \times CompId$ with values $\{(channel_1, comp_1), \dots\}$ or $dstCmp = ChanId \times CompId$ with values $\{(channel_1, comp_2), \dots\}$.

Intuitively, the conceptual domain describes the domain, from which specific instances of the description of an actual system – called conceptual product model – are constructed. Thus, the conceptual domain model is the set of all possible product models that can be constructed within this domain. Each product model is an “instance-model” of the conceptual domain model, with sub-sets of its entities and relations. In order to be a proper product model, such a conceptual domain model generally must fulfill additional constraints. In case of the above conceptual domain model shown in Figure 2, e.g., each channel must have an associated source and destination component in the *srcCmp* and *dstCmp* relation.

B. Structure of the Model

To provide mechanisms for model transformation, the framework provides access to EMF Ecore-based models [2]. As described in Subsection III-A, a (conceptual) product model is a collection of sets of elements (each described as a conceptual entity and its attribute values) and relations (each described as a pair of conceptual entities). To syntactically represent such a model, a Prolog term is used. Since these elements and relations are instances of classes and associations taken from an EMF Ecore model, the structure of the Prolog term – representing an instance of that model – is inferred from the structure of that model. The term comprises the classes and associations, of which the instance of the EMF Ecore model is constructed. It is grouped according to the structure of that model, depending on the package structure of the model and the classes and references of each package, using only simple elementary Prolog constructs, namely compound functor terms and list terms.

A *model term* describes the content of an instance of an EMF Ecore model. It consists of a functor – identifying the

²For simplification purposes, the *Comment* attribute is ignored in the following.

<i>ModelTerm</i>	::=	<i>Functor</i> (<i>ClassesTerm</i> , <i>AssociationsTerm</i>)
<i>ClassesTerm</i>	::=	[] [<i>ClassTerm</i> (, <i>ClassTerm</i>)*]
<i>ClassTerm</i>	::=	<i>Functor</i> (<i>ElementsTerm</i>)
<i>ElementsTerm</i>	::=	[] [<i>ElementTerm</i> (, <i>ElementTerm</i>)*]
<i>ElementTerm</i>	::=	<i>Functor</i> (<i>Entity</i> (, <i>AttributeValue</i>)*)
<i>Entity</i>	::=	<i>Atom</i>
<i>AttributeValue</i>	::=	<i>Atom</i>
<i>AssociationsTerm</i>	::=	[] [<i>AssociationTerm</i> (, <i>AssociationTerm</i>)*]
<i>AssociationTerm</i>	::=	<i>Functor</i> (<i>RelationsTerm</i>)
<i>RelationsTerm</i>	::=	[] [<i>RelationTerm</i> (, <i>RelationTerm</i>)*]
<i>RelationTerm</i>	::=	<i>Functor</i> (<i>Entity</i> , <i>Entity</i>)

Table I
THE PROLOG STRUCTURE OF A MODEL TERM

model – with a classes term and an associations term as its argument.³ The *classes term* describes the EClasses of the corresponding package. It is a list of class terms, one for each EClass of the package. Each *class term* consists of a functor – identifying the class - and an elements term. An *elements term* describes the collection of objects instantiating this class; it is a list of element terms. Note that each elements term comprises only the collection of those objects of this class, which are not instantiations of subclasses of this class; objects instantiating specializations of this class are only contained in the elements terms corresponding to the most specific class. Finally, an *element term* - describing such an instance – consists of a functor – again identifying the class this object belongs to – with an entity identifying the element and attributes as arguments. Each of the attributes are atomic representations of the corresponding values of the attributes of the represented object. The entity is a regular atom, unique for each element term.

Similarly to an elements term, each *associations term* describes the associations, i.e., the instances of the EReferences of the EClasses, for the corresponding package. Again, it is a list of association terms, with each *association term* consisting of a functor – identifying the association - and an relations term, describing the content of the association. The *relations term* is a list of relation terms, each *relation term* consisting of a functor – identifying the relation – and the entity identifiers of the related objects. In detail, the Prolog model term has the structure shown in Table I in the BNF notation with corresponding *non-terminals* and *terminals*.

The functors of the compound terms are taken from the EMF Ecore model, which the model term is representing:

- the functor of a *ModelTerm* corresponds to the name of the EPackage the term is an instance of;
- the functor of a *ClassTerm* to the name of the EClass the term is an instance of; and finally

³Actually, a model term describes the packages of the EMF Ecore model. This aspect of the term representation is skipped here for simplification purposes. A complete description can be found in [1].

- the functor of an AssociationTerm corresponds to the name of the EReference the term in an instance of.

Since EMF – unlike MOF [3] – does not support associations as first-class concepts like EClasses but uses EReferences instead, EReference names are not necessarily unique within a package. Therefore, if present in the ECore model, EAnnotation attributes of EReferences are used as the functors of an AssociationTerm. Similarly, the atoms of the attributes are deduced from the instance of the EMF Ecore model, which the model term is representing:

- the entity atom corresponds to the object identifier of an instance of an EClass
- the attribute corresponds to the attribute value of an instance of an EClass.

C. Construction Predicates

In a strictly declarative rule-based approach, a transformation is described as a predicate relating the models before and after it. Therefore, predicates are needed to deconstruct a model into its parts as well as to construct it from them. As the structure of the model is defined using only compound and list terms, only two forms of predicates are needed: union and compound predicates.

1) *List Construction*: The construction and deconstruction of lists is managed by means of the union predicate `union/3` with template⁴

```
union(?Left, ?Right, ?All)
```

such that `union(Left, Right, All)` is true if all elements of list `All` are either elements of `Left` or `Right`, and vice versa. Thus, e.g., `union([1, 3, 5], R, [1, 2, 3, 4, 5])` succeeds with `R = [2, 4]`.

2) *Compound Construction*: Since the compound structures used to build the model instances depend on the actual structure of the EMF Ecore model, only the general schemata used are described. Depending on whether a class/element or association/relation is described, different schemata are used. In both schemata the name of the class or relation is used as the name of the predicate for the compound construction.

Class and Element Compounds: The (de)construction of classes/elements is managed by means of class/element predicates of the form `class/2` and `class/N+2` where `N` is the number of the attributes⁵ of the corresponding class, with templates

```
class(?Class, ?Elements)
class(?Element, ?Entity, ?Attr1, ..., ?AttrN)
```

where `class` is the name of the class and element (de)constructed. Thus, e.g., the class named `component`

⁴According to standard convention, arbitrary/input/output arguments of predicates are indicated by `?/+/-`.

⁵If a class does not contain any attributes, `N` is zero, leading to an overloading of the corresponding predicate.

in the EMF Ecore model in Figure 2 is represented by the compound constructor `component`. The class predicate is true if `Class` is the list of `Objects`; it is generally used in the form `class(+Class, Objects)` to deconstruct a class into its list of objects, and `class(-Class, +Objects)` to construct a class from a list of objects. Similarly, the element predicate is true if `Element` is an `Entity` with attributes `Attr1, ..., AttrN`; it can be used to deconstruct an element into its entity and attributes via `class(+Element, -Entity, -Attr1, ..., -AttrN)`, to construct an element from an entity and attributes (e.g. to change attributes of an element) via `class(-Element, +Entity, +Attr1, ..., +AttrN)`, or to construct an element including its entity from the attributes via `class(-Element, -Entity, +Attr1, ..., +AttrN)`. Thus, e.g., `component(Components, [Control, Error, Mgmt])` is used to construct a class `Components` from a list of objects `Control`, `Error`, and `Mgmt`. Similarly, `component(Mgmt, Management, 10, "Mgmt")` is used to construct an element `Mgmt` with entity `Management`, load `10`, and name `"Mgmt"`.

Association and Relation Compounds: The construction and deconstruction of associations and relations is managed by means of association and relation predicates of the form `association/2` and `association/3` with templates

```
association(?Association, ?Relations)
association(?Relation, ?Entity1, ?Entity2)
```

where `association` is the name of the association and relation constructed/deconstructed. Thus, e.g., a relation named `subComponent` in the EMF Ecore model in Figure 2 is represented by the compound constructor `subComponent`. The relation predicate is true if `Association` is the list of `Relations`; it is generally used in the form `association(+Association, -Relations)` to deconstruct an association into its list of relations, and `association(-Association, +Relations)` to construct an association from a list of relations. Similarly, the relation predicate is true if `Relation` associates `Entity1` and `Entity2`; it is used to deconstruct a relation into its associated entities via `association(+Relation, -Entity1, -Entity2)` and to construct a relation between two entities via `association(-Relation, +Entity1, +Entity2)`. E.g., `srcComp(SrcComps, [ErrCtrl, StsErr, CmdMgmt])` is used to construct the source-component association `SrcComps` from the list of relations `ErrCtrl`, `StsErr`, and `CmdMgmt`. Similarly, `subComponent(ErrCtrl, Err, Control)` is used to construct relation `ErrCtrl` with `Control` being the

source-component of `Err`.

IV. EXPLORATION BY TRANSFORMATION

In general, design space exploration is done by extending the abstract model by implementation constraints to identify suitable solutions. However, while often it is rather straightforward to characterize whether a solution is acceptable, it is more complicated to effectively construct such a solution. Thus, design space exploration is often performed by constructing potential solutions and then checking whether these solutions are acceptable or consistent, potentially using automatic techniques for the latter step.

This section illustrates how a characterization of the solution space can be used to effectively perform a mechanized search for a consistent solution in this space. This is achieved by interpreting the declarative characterization of the solution space as an operational description of a possibly ambiguous transformation, allowing an automatic search for suitable solutions within the space.

A. Description of Solution Space

In the relational approach, a model is represented as a *single structured term* using *named compounds* as well as *anonymous sets*. The model term has a *hierarchic structure*, consisting of *classes* and *associations*. Classes and associations consist of sets of *elements* and *relations*, resp., wrapped in compounds. Finally, elements and relations are formalized as compounds of values. Applied to the running example, the representation of the model in Section II-A is shown in Figure 3.⁶ As shown in line 1 it consists of a package named `Model`. Its classes and associations – as shown in lines 2 and 3 – are the sets *Components*, *Units*, *Channels*, and *Links*, as well as *SrcCmp*, *DstCmp*, *Deploys*, and *Allocs*, identified by suitable named compounds (e.g., `comp`, `unit`, `srcCmp`, or `deploy`). These sets – like *Components* and *srcComps* – consist of elements and relations, resp., with themselves are named compounds – like `comp(ctrl, "Control", 10)` and `srcCmp(err, ctrl)` in lines 10 and 6 – using element identifiers like `ctrl` and `err`.

Using this representation of a model, the notion of a complete and consistent deployment of components and channels to units and links, resp., can be defined. For sake of brevity, only the deployment of components to units is illustrated: A collection `Units` of units is called *resource-consistent with a collection Comps of components deployed via associations Deploys* if and only if

- Either the sets `Comps` and `Deploys` are empty (indicating that no components must be deployed)
- Or `Units` contains a unit `Unit` with load `Load`, `Comps` contains a subset `UnitComps`, and `Deploys` contains a subset `UnitDeploys` such that `UnitComps` deployed to

⁶The relational representation uses shortened functors like `comp` for component; furthermore, the `comment` attribute is skipped for sake of brevity.

`Unit` via `UnitDeploys` is resource-consistent with `Load` and the remaining units are resource-consistent with the remaining components deployed via the remaining associations

A collection `Comps` of components deployed to a unit `Unit` via associations `Deploys` is called *resource-consistent with available load RestLoad* if and only if

- Either the sets `Comps` and `Deploys` are empty (indicating no components to be deployed to units) and there is a sufficient (i.e., positive) load `RestLoad`
- Or `Comps` contains a component `Comp` with load `CompLoad` deployed via an association `Deploy` between `Comp` and `Unit` in `Deploys` and the remaining collection of components deployed to `Unit` via the remaining set of associations is resource-consistent with the remaining load `RestLoad – CompLoad`

Such an informal characterization of the notion of a consistent and complete deployment is generally straightforward for a domain expert. Using a rule-based relational style, this characterization can be directly translated to formalization of a complete and consistent deployment in a declarative manner, . Figure 4 shows the corresponding formalization where `consUnit` checks whether the deployment of components to a specific unit is resource-consistent, while `consUnits` checks whether the collection of components deployed to the units is resource-consistent. The allocation of channels to links can be performed in a similar fashion, as indicated by the use of relations `consLinks` and `consLink` in Figure 5.⁷

To check the consistent deployments for all units, `consUnits` selects a *Unit* with load *Load* from the set of *Units* (line 3) as well as corresponding subsets *UnitComps* and *UnitDeploys* from the sets *Comps* and *Deploys* of components and deployments (line 4) – and checks whether all components in *UnitComps* deployed to *Unit* via *UnitDeploys* are source-consistent via `consUnit` (line 5); unless all components have been deployed (line 1) this is repeated for all remaining components (line 6).

Similarly, to check the consistent deployment for a specific unit, `consUnit` selects a *Comp* with required load *CompLoad* from the set of *Comps* (line 10) as well as the deployment relation mapping *Comp* to the unit under consideration from the set *Deploy* (line 11); unless all components have been consistently deployed (line 8) this is repeated for all remaining components (line 12).

The relations `consUnits` and `consUnit` introduced in Figure 4 provide a declarative characterization whether a set `Deploys` of deployment relations between units from `Units` and components from `Comps` is complete and consistent with respect to these components and units. It therefore can be used to *check* whether a given deployment

⁷The definition of relations `consLinks` and `consLink` is skipped for sake of brevity.

```

1  Model = model(Classes,Assocs)
2  Classes = [comp(Components), unit(Units), chan(Channels), link(Links)]
3  Assocs = [srcCmp(SrcComps), dstCmp(DstComps), deploy(Deployments), alloc(Allocs)]
4  Components = [Control, Error, Mgmt], Units = [Window, Diag]
5  Channels = [Err, Sts, Cmd], Links = [Body, Dia]
6  SrcComps = [srcCmp(err,ctrl), srcCmp(sts,error), srcCmp(cmd,error)]
7  DstComps = [dstCmp(err,error), dstCmp(sts,mgmt), dstCmp(cmd,mgmt)]
8  Deploys = [deploy(ctrl,win), deploy(err,win), deploy(mgmt, diag)]
9  Allocs = [alloc(sts,dia), alloc(cmd, dia)]
10 Control = comp(ctrl,"Control",10), Error = comp(error,"Error",5)
11 Mgmt = comp(mgmt,"Mgmt",10), Window = unit(win,"Window",20), Diag = unit(diag,"Diag",15)
12 Err = chan(err, "Err", 10), Sts = chan(sts, "Stst", 5), Cmd = chan(cmd, "Cmd", 5)
13 Body = link(body, "Body", 20), Dia = link(dia, "Dia", 10)

```

Figure 3. Relational Representation of Power Window Model

```

1  consUnits(Units,[],[]).
2  consUnits(Units,Comps,Deploys) :-
3    unit(Unit,Ident,Load,Name), union([Unit],OtherUnits,Units),
4    union(UnitComps,OtherComps,Comps), union(UnitDeploys,OtherDeploys,Deploys),
5    consUnit(Ident,Load,UnitComps,UnitDeploys),
6    consUnits(OtherUnits,OtherComps,OtherDeploys).
7
8  consUnit(Unit,RestLoad,[],[]) :- RestLoad >= 0.
9  consUnit(Unit,RestLoad,Comps,Deploys) :-
10   comp(Comp,Ident,CompLoad,Name), union([Comp],OtherComps,Comps),
11   deploy(Deploy,Ident,Unit), union([Deploy],OtherDeploys,Deploys),
12   consUnit(Unit,RestLoad - CompLoad,Comps,Deploys).

```

Figure 4. Rule-based Formalization of a Deployment

is complete and consistent. However, due to its rule-based declarative character, this definition can also be used to *generate* suitable deployments.

In order to generate the deployment, the above definition has to be embedded in a premodel-postmodel relation, linking a model without deployment to a model extended with a corresponding deployment. Figure 5 shows the embedding, where relation *generate* builds a consistent deployment for a given model resulting in an extended model if possible.

Relation *generate* amends the given model `model(Classes,PreAssocs)` resulting in an extended model `model(Classes,PostAssocs)` by adding new relations to the pre-model to obtain the post-model and leaving the classes unchanged (line 1). While the approach described in Section III also supports the introduction of new elements, this is not necessary for the construction of a deployment. To that end, the deployment relations `PreDeploys` and the allocation relations `PreAllocs` are taken from the `PreAssocs` (line 7) and – after generating a complete deployment if possible via relations `consUnits` and `consLinks` (lines 8 and 9) – added to the `PostAssocs` (line 11).

This *generate* relation can not only be applied to pre-models containing no deployment (as well as allocation) relations; furthermore, the same relation can also be applied to models with an already existing partial deployment (or

allocation) in `PreAssocs`, which is extended to a complete deployment (allocation) if possible.

B. Execution of Transformation

The approach has been implemented as an Eclipse plugin using the tuProlog engine [4], supporting the transformation of EMF Ecore [2] models. The implemented plug-in provides tool support both for the definition of transformation and the transformation execution.

The transformation is provided in form of a transformation wizard, guiding the user through the transformation process of selecting an executing the transformation, and identifying and applying the intended solution. The execution of the transformation itself involves the translation of the pre-model from the EMF to the Prolog representation, the application of the transformation relation, and finally the translation of the post-model from the Prolog to the EMF representation.

As the transformation relation specified in Figures 4 and 5 is executed by the Prolog mechanism, different solutions consistent with this relation can be explored. By repeatedly evaluating the corresponding Prolog term, all possible solutions can be generated and inspected. Once a suitable solution is returned by the Prolog backtracking mechanism, this solution is then transformed to the corresponding EMF form. Figure 6 shows two solutions generated for deployment problem described in Figure 1.

```

1 generate(model(PreModel), model(PostModel)) :-
2   model(PreModel,Classes, PreAssocs), model(PostModel, Classes, PostAssocs),
3   comp(Comp,Components), unit(Unit,Units), chan(Chan, Channels), link(Link, Links),
4   union([Comp, Unit, Chans, Links], [], Classes),
5   srcCmp(Src,SrcComps), dstCmp(Dst,DstComps), units(Uts,Units),
6   deploy(PreDep, PreDeploys), alloc(PreAll, PreAllocs)],
7   union([Src, Dst, Uts, PreDep, PreAll], [], PreAssocs),
8   union(PreDeploys, AddDeploys, Deploys), consUnits(Units, Comp, Deploys),
9   union(PreAllocs, AddAllocs, Allocs), consLinks(Links, Chans, Src, Dst, Uts, Allocs, Deploys),
10  deploy(PostDep,Deploys), alloc(PostAll,Allocs),
11  union([Src, Dst, Uts, PostDep, PostAll], [], PostAssocs).

```

Figure 5. Generation of a Deployment

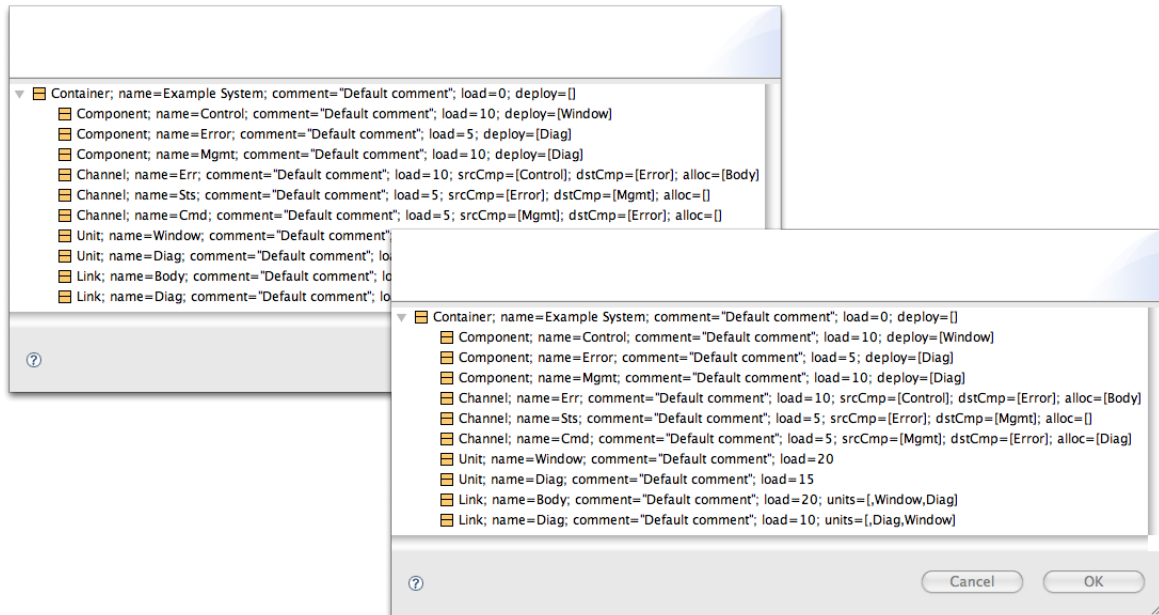


Figure 6. Example Solutions Generated for the Deployment

To improve the efficiency of the exploration, the original characterization of the set of possible solutions can be improved, e.g., by reordering the clauses or adding extra restrictions. In case of the `consUnits` relation of Figure 4, the constraint `RestLoad >= 0` of line 8 can be moved to line 10 to avoid the unnecessary further construction of already inconsistent deployments.

By applying these kind of improvements, the original characterization of a consistent solution can be turned into a more efficient search strategy, allowing to generate solutions for more realistic problems. In case of the deployment generation, these improvements can be applied to generate a search strategy for typical medium-sized problems.

Table II shows that these improvements ensure the scalability of the transformation. While the direct application of the transformation relation – indicated by *direct* entry in column *Spec* – grows drastically in execution – from 142.48

msec for a system with 3 component, 3 channels, 2 units, and 2 busses to 217.08 sec for a system with 8 components, 10 channels, 3 units, and 2 busses – the efficiency can be drastically improved by standard optimizations as described above. After applying the reordering described above – shifting the constraint check from the termination rule to the iteration rule, or moving partially instantiated clauses in front of completely uninstantiated ones – the same models – as indicated by entry *reordered* – can be transformed within 42.23 msec and 771.37 msec, resp.

By using a more operational form of specification – e.g., by using recursion over the components rather than constructing subsets as in line 4 of Figure 4 – execution can be furthermore improved substantially. As shown by the *operational* entry in Table II these transformations can be executed in 22.74 msec and 97.18 msec, resp. Thus, for example, a system with 14 components and 18 channels to

# Components	# Channels	# Units	# Busses	Spec	Time [msec]	Space [Mb]
3	3	2	2	direct	142.48	33.90
3	3	2	2	reordered	42.23	32.23
3	3	2	2	operational	22.74	30.14.
4	5	3	2	direct	2137.34	33.61
4	5	3	2	reordered	58.27	32.17
4	5	3	2	operational	51.99	26.75
8	10	3	2	direct	217083.83	39.29
8	10	3	2	reordered	771.37	41.35
8	10	3	2	operational	97.18	32.92

Table II
SAMPLE MEASUREMENTS FOR DIFFERENT MODELS AND FORMALIZATIONS ON AN 2.8 GHZ INTEL CORE 2 DUO

be deployed to a platform with three units and two buses – leading to a solution space of potentially $3^{14} \times 2^{18} \approx 10^{12}$ different possible configurations - can be handled within 194.83 msec.

V. CONCLUSION

In the previous sections, an approach for the exploration of possible solutions within a design-space defined by a declarative, rule-based characterization of valid solutions was introduced. This approach allows to define the characteristics of the solution space via constraints given as relational rules, immediately giving rise to an executable search strategy for finding solutions. This section compares the approach to related work and discusses possible further extensions.

A. Related Work

Especially in the context of embedded software systems, a development process based on techniques for design-space exploration via an incremental extension of models has been advocated by different approaches, e.g., in the Metropolis approach [5]. Despite this general appreciation of design-space exploration, so far little tool support for the mechanical exploration of these design options has been provided. Furthermore, in general, little infra structure is provided to construct such support techniques based on a description of the design constraints; current approaches like [6] or [7] require realizations on the level of the tool implementation rather than the conceptual domain level. Other approaches like [8] or [9], which are closest to the approach presented here, use a constraint logic programming technique to search the design space. Unlike the approach here, however, they are not incorporated in current modeling tools like Eclipse, and therefore do not allow a seamless integration. Furthermore the approach presented here also allows to explore unbounded domains, as needed, e.g., in the introduction of new clustering components in the refactoring of architectures.

The alternative approach presented here allows to use a declarative form of the constraints characterizing the design space directly for the generation of solutions for these

constraints, interpreting these constraints as transformation relations. A transformation framework – provided as an Eclipse PlugIn [10] – supports the transformation of EMF Ecore models using a declarative relational style. The purely relational declarative form of specification can be tuned to ensure an efficient execution. Obviously, the rule-based approach allows very general forms of application, using the back-tracking mechanism to explore alternative transformation results. The approach presented here extends [1] by using the formalism of relational model transformations to formalize the constraints characterizing the design space, interpreting these relations as a search strategy to find solutions within that space, and thus allowing to generate all possible solutions respecting these constraints.

The purely relational approach combined with the rule-based execution mechanism including backtracking is a necessary pre-requisite to support the design space exploration, lacking in approaches like MOFLON/TGG [11], Viatra [12], or FuJaBa [13], or GME [14]. Also the QVT approach [15] and its respective implementations like ATLAS [16], F-Logics based transformation [17], or TEFKAT [18] lack its capability to interpret loose characterizations of the resulting model, supporting the exploration of a set of possible solutions. Similar to PROGRES [19], the approach presented here makes use of a back-tracking mechanism, specifically the one provided by Prolog. However, in contrast to it backtracking is additionally used to produce alternative transformation results not only by automatically searching for a constraint-satisfying solution, but also be incrementally generating it to allow the user to interactively identify and select the appropriate solution.

B. Outlook

The approach presented here has been successfully applied to medium-sized problem spaces. Other applications of this approach include the identification of self-contained sub-systems to support testing of incremental releases, or the generation of different equivalent refactorings of component architectures; the latter also includes the introduction of new elements in form of clustering components during the generation of the solutions. To improve performance also

for larger systems, currently the infrastructure – e.g., the implementation of the `union` relation – are optimized.

Furthermore, the exploration tasks considered in the previous sections target the class of problems, where different possible solutions respecting the design constraints are considered equally suitable. In the example of the generation of a deployment, it is sufficient to identify a solution consistent with the provided resource constraints; therefore, different possible solutions are provided in no specific order. However, other design space exploration tasks, e.g., the choice of suitable hardware units for a given software architecture, require to select solutions respecting the imposed constraints and to additionally maximize (or minimize) an additional design object, e.g., the cost of the chosen hardware units. For such class of problems, additional constraints can be added to support the generation of solutions maximizing this specific design objective. These additional constraints are then used to enhance the search strategy with suitable heuristics to generate solutions in an order favoring maximal (or minimal) solutions first.

Finally, in the current approach, the propagation and retraction of constraints – like the remaining available load – is handled implicitly via the rules-based characterization of the design space. To optimize the search, the declarative characterization is improved by refactoring the clauses of the characterization or by adding additional constraints guiding the search. In rule-based approaches to constraint problems, the use of constraint handling rules – as provided in constraint logic programming – has proven to support especially effective solution strategies, as e.g. in [8] or [9]. This technique allows to formalize the constraints independent of their order of application; the concrete order of application is defined by the system, by translating the abstract rules in more concrete ones. For the approach presented here, a similar technique can be provided, translating a more declarative characterization in a more efficient version.

REFERENCES

- [1] B. Schätz, “Formalization and Rule-Based Transformation of EMF Ecore-Based Models,” in *Software Language Engineering*, ser. LNCS, E. V. W. Dragan Gasevic, Ralf Laemmel, Ed. Springer, 2009.
- [2] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2007, second Edition.
- [3] *Meta Object Facility (MOF) Specification*, Object Management Group, Inc., 2000, <http://www.omg.org/>.
- [4] E. Denti, A. Omicini, and A. Ricci, “Multi-paradigm Java-Prolog integration in tuProlog,” *Science of Computer Programming*, vol. 57, no. 2, pp. 217–250, 2005.
- [5] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli, “Formal Models for Embedded System Design,” *Design and Test of Computers*, 2000.
- [6] B. Dougherty, J. White, C. Thompson, and D. Schmidt, “Automating Hardware and Software Evolution Analysis,” in *International Conference on the Engineering of Computer Based Systems (ECBS)*, 2009.
- [7] A. M. Alan Dearle, Graham Kirby, “A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications,” University of St Andrews, Tech. Rep., 2004.
- [8] K. Kuchcinski, “Constraints-driven design space exploration for distributed embedded systems,” *Journal of Systems Architecture*, no. 47, 2001.
- [9] B. K. Eames, S. K. Neema, and R. Saraswat, “DesertFD: a finite-domain constraint based tool for design space exploration,” *Design Automation for Embedded Systems*, 2008.
- [10] “Prolog EMF Transformation Eclipse-PlugIn,” www4.in.tum.de/~schaetz/PETE, 2009.
- [11] F. Klar, A. Königs, and A. Schürr, “Model transformation in the large,” in *ESEC/FSE’07*. ACM Press, 2007.
- [12] D. Varro and A. Pataricza, “Generic and meta-transformations for model transformation engineering,” in *UML 2004*, T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, Eds. Springer, 2004, INCS 3273.
- [13] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf, “Tool integration at the meta-model level: the fujaba approach,” *STTT*, vol. 6, no. 3, pp. 203–218, 2004.
- [14] J. Sprinkle, A. Agrawal, and T. Levendovszky, “Domain Model Translation Using Graph Transformations,” in *ECBS2003 - Engineering of Computer-Based Systems*, 2003.
- [15] OMG, “Initial submission to the MOF 2.0 Q/V/T RFP,” Object Management Group (OMG), <http://www.omg.org>, Tech. Rep. ad/03-03-27, 2003.
- [16] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, “ATL: a QVT-like transformation language,” in *OOPSLA ’06*. ACM Press, 2006, pp. 719–720.
- [17] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, “Transformation: The Missing Link of MDA,” in *Graph Transformation*, ser. LNCS, vol. 2505, 2002.
- [18] M. Lawley and J. Steel, “Practical declarative model transformation with Tefkat,” in *MoDELS Satellite Events*, ser. LNCS, J.-M. Bruel, Ed., vol. 3844. Springer, 2006.
- [19] A. Schürr, A. J. Winter, and A. Zündorf, “The PROGRES Approach: Language and Environment,” in *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999.