Abstract: Especially in the domain of embedded systems, system development is performed via step-wise design-space exploration, using an incremental addition of design decision. Each development step is characterized by design constraints, limiting the possible solution space. By applying model transformations based on a declarative, relational approach, these constraints can be used to support this exploration of design alternatives. The approach is demonstrated for the (semi-)automatic deployment of logical architectures to hardware platforms.

1 Introduction

The development process – of software systems in general and of embedded system in particular – can be understood as a sequence of design decisions with each 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.

Thus obviously, the development process can be understood as a sequence of transformations, each enriching the model under development. Especially in the context of model-driven approaches, model transformation techniques have been developed to support the automatic generation of those transformed models. However, most of those approaches have concentrated mainly on transformations of models to mechanically obtain a single specific transformed model from a given one.

However, the software design process generally involves decision making by the software engineer. Typically, these decision are based on analysis and experience, and cannot be done fully automatic. The decisions are based on abstract models and lead to more concrete, refined models. Often there are multiple possible ways to solve the problem and the developer has to decide which way to go.

This contribution shows how approaches allowing for the definition of loose transformations, i.e., transformations with different possible solutions, can be used to mechanize and interactive and incremental development process.
The remainder of this contribution is organized as follows: Section 2 introduces deployment – i.e., the mapping of logical components and channels to technical units and links – as a typical example of an exploration step in the design process, requiring the evaluation of different variants of such mappings with respect to load restrictions. Furthermore, the relation between such an exploration of the design space and declarative, relational descriptions of model transformations is established.

Section 2 gives a short introduction to a representation for conceptual models, enabling the application of those declarative, rule-based transformation specifications to these representations using a Prolog.

Section 4 shows how this technique can be used to obtain an interactive and incremental support for the exploration of the deployment possibilities by providing a formalization of the constraints of the solution space.

Section 5 finally summarizes the central aspects of the present approach and compares it to related work.

2 Design Space Exploration

In an ideal systems development process, most development activities can be understood as refinement steps from abstract models towards more concrete ones, adding additional information by making design decisions. Besides rather mechanical activities, obviously these development steps cannot be fully automated, because in general there is not only one possible refinement step to be taken. The development process requires the designer to identify a suitable refinement in case there is more than one possible solution, forcing him to take design decisions, some even being equally optimal. Thus development depends mainly on the developer’s abilities and experience of constructing and evaluation these different possibilities.

However, the developer can be supported in his search for the right solution. In the following we show that design space exploration – i.e., the construction of different possible alternatives within a constrained solution space – is suitable problem field to apply (semi-automatic, i.e. interactive) model transformation techniques to guide and support the developer in his decision process.

2.1 Example: Component Deployment

For the demonstration of the approach, we use component-based deployment of soft real-time systems as a running example. Component-based deployment, commonly found in the model-based development of embedded software, encompassed the resource-constrained mapping of a set of logical components connected by channels to distributed electronic control units connected by links. The constraints imposed on the mapping from component and channels to control units and links, resp., require the mapping to be resource-
**consistent** by enforcing that the (average) load required by a component or channel is less than load provided by a control unit or a link, in case the former are mapped to latter. Components require computational load, while units provide computational load. Channels require communication load, while links provide communication load.

While 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; it is not mapped to a link if unit-internal communication can be used.

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 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 mapping is called a *complete deployment* if all components and channels are mapped to units and links. Furthermore, the 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. 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.
2.2 Approach

As mentioned above, 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.

A mechanised 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 operative version guiding the search process.

To support an effective and efficient process of design space exploration, these operative 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 [Sch09], 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 remained, we show how this mechanism can be used to turn a declarative description of design constraints into an automized 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. 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.
3 Defining Models, Constraints, and Transformations

As mentioned in the previous, the purpose of the approach presented here is the construction (or rather completion) of descriptions of systems under development – as shown in Figure 1 – to increase the efficiency and quality of the development process. To construct formalized descriptions of a system under development, a ‘syntactic vocabulary’ – also called conceptual (domain) model in [SH99] – is needed. This conceptual model characterizes all 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.

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 subsection, a formalization of conceptual domain models and conceptual product models based on relations is given as well as their representation in a declarative fashion using Prolog style.

3.1 Formalization

A conceptual domain model provides an interpretation for syntactical descriptions like in Figure 2. Basically, the conceptual domain model defines the primitives used to describe a system: concepts characterizing unique entities used to describe a system, with examples in the deployment domain 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

\[\text{component} \]

\[\text{name} \]

\[\text{comment} \]

\[\text{deploy} \]

\[\text{srcComp} \]

\[\text{dstComp} \]

\[\text{unit} \]

\[\text{name} \]

\[\text{comment} \]

\[\text{load} \]

\[\text{SLOC} \]

\[\text{units} \]

\[\text{load} \]

\[\text{name} \]

\[\text{comment} \]

\[\text{In the context of technologies like the Meta Object Facility, the class diagram-like definition of a conceptual domain model is generally called meta model.}\]
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 $\text{CompId} = \{\text{comp}_1, \text{comp}_2, \ldots\}$, and $\text{ChanId} = \{\text{chan}_1, \text{chan}_2, \ldots\}$; typical examples for set of attribute values are $\text{CompName} = \{\text{‘Control’}, \text{‘Error’}, \ldots\}$ or $\text{CompLoad} = \{0, 1, 2, \ldots\}$.

Based on these primitives, the conceptual domain model consists of elements corresponding to objects used to model a system, like $\text{Component}$, or $\text{Channel}$ to define the components and channels of the description of the logical architecture of a system; and relations corresponding to dependencies between the elements, like $\text{srcCmp}$ or $\text{dstCmp}$ to define the source or destination component of a channel.

The conceptual domain consists of a collection of element relations between conceptual entities and attribute values, and a collection of (binary) association relations between conceptual entities. In case of the above conceptual domain model for structural descriptions as provided in Figure 2\(^2\), examples for element relations are $\text{Component} = \text{CompId} \times \text{CompName} \times \text{CompLoad}$ with values $\{(\text{comp}_1, \text{‘Control’}, 5), (\text{comp}_2, \text{‘Error’}, 5), \ldots\}$ or $\text{Channel} = \text{ChanId} \times \text{ChanName} \times \text{ChanLoad}$ with values $\{(\text{channel}_1, \text{‘Err’}, 10), (\text{channel}_2, \text{‘Sts’}, 5), \ldots\}$; examples for association relations are $\text{srcCmp} = \text{ChanId} \times \text{CompId}$ with values $\{(\text{channel}_1, \text{comp}_1), \ldots\}$ or $\text{dstCmp} = \text{ChanId} \times \text{CompId}$ with values $\{(\text{channel}_1, \text{comp}_2), \ldots\}$. Intuitively, the conceptual domain describes the domain, from which specific instances of the description of an actual system – called conceptual product model in the following – are constructed.

Intuitively, the conceptual domain model is the set of all possible product models that can be constructed within this domain. Thus, each product model is a “sub-model” of the conceptual domain model, with sub-sets of its entities and relations. In order to be a proper product model, such a subset of the conceptual domain model generally must fulfill additional constraints; typical examples are the constraints in meta-models represented as class diagrams. 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 $\text{srcCmp}$ and $\text{dstCmp}$ relation.

### 3.2 Structure of the Model

The transformation framework provides mechanisms for a pure (i.e., side-effect free) declarative, rule-based approach to model transformation. To that end, the framework provides access to EMF Ecore-based models [SBPM07]. As described in Subsection 3.1, formally, a (conceptual) 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,

---

\(^2\)For simplification purposes, the $\text{Comment}$ attribute is ignored in the following.
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. The structure of the model is built using only simple elementary Prolog constructs, namely compound functor terms and list terms.

To access a model, the framework provides construction predicates to deconstruct and reconstruct a term representing a model. Since the structure makes only use of compound functor terms and list terms, only two classes of construction predicates are provided, namely the union operation and the composition operations.

3.2.1 Term Structure of the Model

A model term describes the content of an instance of a EMF Ecore model, i.e., the instances of its classes and associations. It consists of a functor – identifying the model – with a classes terms 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, and thus – in turn – 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 a 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 1 in the BNF notation with corresponding non-terminals and terminals.

The functors of the compound terms are deduced 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

\(^3\) Actually, 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 [Sch09].
Table 1: The Prolog Structure of a Model Term

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

Since EMF – unlike MOF – 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 a EClass, while
- the attribute corresponds to the attribute value of an instance of an EClass.

Currently, while basically also multi-valued attributes can be handled by the formalism, only single-valued attributes like references (including null references), basic types, and enumerations are supported by the implementation.

### 3.3 Construction Predicates

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

The construction and deconstruction of lists is managed by means of the union predicate union/3 with template\(^4\)

\[
\text{union}(?\text{Left}, ?\text{Right}, ?\text{All})
\]

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

3.3.2 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 \(\text{class}/2\) and \(\text{class}/N+2\) where \(N\) is the number of the attributes of the corresponding class, with templates

\[
\begin{align*}
\text{class}(?\text{Class}, ?\text{Elements}) \\
\text{class}(?\text{Element}, ?\text{Entity}, ?\text{Attribute1}, ..., ?\text{AttributeN})
\end{align*}
\]

where \(\text{class}\) is the name of the class and element (de)constructed. Thus, e.g., the class named component in the EMF Ecore model in Figure 2 is represented by the compound constructor component. The class predicate is true if \(\text{Class}\) is the list of Objects; it is generally used in the form \(\text{class}(+\text{Class}, \text{Objects})\) to deconstruct a class into its list of objects, and \(\text{class}(-\text{Class}, +\text{Objects})\) to construct a class from a list of objects. Similarly, the element predicate is true if \(\text{Element}\) is an Entity with attributes \(\text{Attribute1}, ..., \text{AttributeN}\); it can be used to deconstruct an element into its entity and attributes via \(\text{class}(+\text{Element}, -\text{Entity}, -\text{Attribute1}, ..., -\text{AttributeN})\), to construct an element from an entity and attributes (e.g. to change attributes of an element) via \(\text{class}(-\text{Element}, +\text{Entity}, +\text{Attribute1}, ..., +\text{AttributeN})\), or to construct an element including its entity from the attributes via \(\text{class}(-\text{Element}, -\text{Entity}, +\text{Attribute1}, ..., +\text{AttributeN})\). 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", "The management component") is used to construct an element Mgmt with entity Management, load 10, name "Mgmt", and comment "The management component".

\(^4\)According to standard convention, arbitrary/input/output arguments of predicates are indicated by \(?/+/-\).
**Association and Relation Compounds**  The construction and deconstruction of associations and relations is managed by means of association and relation predicate of the form `association/2` and `association/3` with templates

\[
\text{association}(\text{?Association}, \text{?Relations}) \\
\text{association}(\text{?Relation}, \text{?Entity1}, \text{?Entity2})
\]

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., `srcCmp(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`.

4 Exploration by Transformation

As mentioned in Section 2 general, design space exploration is done by imposing additional design restrictions, 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 or consistent, it is more complicated to effectively construct such a solution. Thus, often design space exploration is understood as constructing potential solutions and then checking whether these solutions are acceptable or consistent, often using automatic techniques for the latter step.

In this section we use the example of generating a resource-consistent deployment to illustrate 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 into an operational description of a possibly ambiguous transformation, allowing to automatically search for suitable solutions within the space.

4.1 Description of Solution Space

In the relational approach, a model is represented as a single term using named compounds with named constructors as well as anonymous sets with union constructor. The model has
a hierarchic structure, consisting of packages that – in turn – may consist of sub-packages. Each package consists of a set 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 2.1 is shown in Figure 3. As shown in line 1 it only 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, Deployments, and Allocations, 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, 10) and srcCmp(err, ctrl) in lines 10 and 6 – using element identifiers like ctrl and err.

Using the above formalized representation of a model, the notion of a complete and consistent deployment of can be defined. In the first step, we will only consider the deployment of components to units: 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 no to be deployed components)
- 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 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 the available (positive) load RestLoad if and only if

\footnote{The relational representation uses shortened functors like comp for component; furthermore, the comment attribute is skipped for sake of brevity.}
Figure 4: Rule-based Formalization of a Deployment

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

This formalization of a complete and consistent deployment immediately leads to a definition in a declarative manner, using a rule-based relational style. Figure 4 shows the corresponding formalization where consUnit checks whether allocation of components to a specific unit respects maximum limit, while consUnits checks whether all components are deployed to a unit respects maximum limit. 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.

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 refDeploy:splitUnitDeploys) – and checks whether all components in UnitComps deployed to Unit via UnitDeploys respect maximum limits 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 deployments 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 refDeploy:consUnitSplitDeploy) – and checks whether the load Comp required by the component does not exceed the remaining load provided by the unit (line 12); unless

---

6The definition of relations consLinks and consLink is skipped for sake of brevity.
generate(model(PreModel), model(PostModel)) :-
model(PreModel,Classes,PreAssocs), model(PostModel,Classes,PostAssocs),
comp(Comp.Components), unit(Unit,Units), chan_CHAN(Channels), link(Link,Links),
union([Comp,Unit,Channels],[],Classes),
srcComp(Src,SrcComp), dstComp(Dst,DstComps), deploy(PreDep,PreDeploys), alloc(PreAll,PreAllocs),
union([Src,Dst,PreDep,PreAll],[],PreAssocs),
union(PreDeploys,AddDeploys,Deploys), consUnits(Units,Comp,Deploys),
deploy(Deploys,PostDep), alloc(PostAll,Allocs),
union([Src,Dst,deploy(Deploys),alloc(Allocs)],[],PostAssocs).

Figure 5: Generation of a Deployment

all components have been deployed (line 8) this is repeated for all remaining components (line 13).

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 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 and extended model if possible.

Relation generate amends the given model model(Classes,PreAssocs) resulting in an extended model model(Classes,PostAssocs) by only adding new relations to the pre-model to obtain the post-model and leaving the classes unchanged (line 1). To that end, the deployment relations PreDeploys and the allocation relations PreAllocs are taken from the PreAssocs (line 6) and – after generating a complete deployment if possible via relations consUnits and consLinks (lines 7 and 8) – added to the PostAssocs (line 10).

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.

4.2 Execution of Transformation

The approach has been implemented as an Eclipse plugin using the tuProlog engine [DOR05], supporting the transformation of EMF Ecore [SBPM07] models. The implemented plug-in provides tool support both for the definition of transformation and the transformation
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.

5 Conclusion

The development of embedded systems via design-space exploration has been repeatedly be defined as an incremental extension of models, e.g., in the Metropolis approach [?]. However, support for the mechanical exploration of these design options has been sparse. Furthermore, in general, little infrastructure is provided to construct such support techniques based on a description of the design constraints, requiring to provide realizations like [?] on the level of the tool implementation rather than the conceptual domain level.

Here, an alternative approach is presented, allowing to turn a declarative formalization of the constraints of the design space into an operational mechanism for the generation of solutions for these constraints, interpreting these constraints as transformation relations. For that purpose, a transformation framework is used, supporting the transformation of EMF Ecore models using a declarative relational style. By taking the operational aspects into consideration, the purely relational declarative form of specification can 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 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 [KKS07], VIATRA [VP04], FuJaBa [GGL05], DlAGEN [Min01], or GME [SAL03]. Also the QVT approach [OMG03] and its respective implementations like ATLAS [IAB+06], F-Logics based transformation [GLR+02], or TefKat [LS06] lack its capability to interpret loose characterizations of the resulting model, supporting the exploration of a set of possible solutions. By making use of the back-tracking mechanism provided by Prolog, alternative transformation results can be applied to automatically search for an optimized solution, which can also be incrementally generated to allow the user to interactively identify and select the appropriate solution.
References


