Software and Systems Modeling
https://doi.org/10.1007/s10270-023-01129-x

REGULAR PAPER

DEPS: a model- and property-based language for system synthesis problems

Pierre-Alain Yvars[1] · **Laurent Zimmer**[2]

Received: 7 November 2022 / Revised: 1 September 2023 / Accepted: 5 September 2023

Abstract
DEPS (design problem specification) is a new modeling language designed to pose and solve system design problems. DEPS addresses problems of sizing, configuration, resource allocation and of architecture generation for systems. Unlike system modeling languages, which are dedicated to the representation of a defined system for evaluation or analysis, we propose a problem modeling language for representing the design problem with a view to its automatic resolution. Compared with other declarative problem modeling languages, DEPS is a declarative structured and property-based language that combines structural modeling features specific to object-oriented languages with problem specification features from constraint programming. The mathematical nature of the problems is described by formal properties encapsulated in models organized according to the architecture of the studied system. The main features of the language are presented in details and are illustrated with examples in different domains. An integrated modeling and solving environment called DEPS Studio allows the designer to express its models in DEPS, to compile the models and to compute automatically the solutions. The validation of the approach is done through two case studies. Finally, we will conclude with the studies and developments in progress which will be integrated into the next version of DEPS Studio.

Keywords Model-based system synthesis · Specification · Design problem modeling language · Problem solving · Constraint programming

1 Introduction

1.1 System modeling languages and problem modeling languages

A lot of work in the field of model-based system engineering (MBSE) has focused on the representation of fully defined systems in order to verify them, evaluate their performance or simulate their operation. It is therefore above all a question of applying an analytical approach to a known system rather than formalizing the engineering problem to be solved [1].

Communicated by Antonio Vallecillo.

✉ Pierre-Alain Yvars
pierre-alain.yvars@isae-supmeca.fr

Laurent Zimmer
laurent.zimmer@dassault-aviation.com

[1] QUARTZ, EA 7393, Institut Supérieur de Mécanique de Paris (ISAE-Supméca), Saint Ouen, France

[2] Direction de la Prospective, Dassault Aviation, Saint Cloud, France

The development and diffusion of modeling languages for software engineering (UML) [2] and then for system engineering (SysML) [3] have produced languages well adapted to the description of systems. Nevertheless they remain solution-oriented. AADL [4] for software systems and Modelica for physical and multi-physical systems [5] are other examples of system description formalisms intended for analysis and simulation.

Moreover, still in the field of solution-oriented approaches, the development of the software tools that support them have motivated research [6, 7] to add the computational capabilities that are naturally missing in basic environments. Indeed, as stated in the OMG SysML tutorial [3]: "Computational engine is provided by applicable analysis tool and not by SysML". Let us also note some recent initiatives to introduce variability in SysML, i.e., degrees of freedom in a system description language coupled with a constraint programming library [1]. These works assume that the problem description is done in UML or in SysML and that its resolution is delegated to a suitable solver after a model transformation. Unfortunately these

Published online: 19 October 2023

languages are made to describe systems and not to model problems. Finally, the development of such a model requires fast development phases which do not seem to us compatible with the heaviness of a model transformation approach [8].

These solution-oriented approaches are necessary for the detailed design phases of systems but insufficient for the preliminary design phases in which detailed models of system components are useless since the aim is to obtain one or more admissible system architectures as soon as possible. This limitation had been pointed out by [6] but does not seem to have been resolved since. Shah suggests that in order to make real progress it would be necessary to have a real object-oriented modeling language for engineering design problems, which would be the counterpart in the field of mathematical programming of what Modelica [5] is in simulation.

The OCL (object constraint language), a complement to UML despite its acronym, is not an exception to this rule. OCL is a specification language that typically allows us to specify invariant conditions that a UML model must check to be correct. But even in its most recent version [9] OCL constraints remain specifications, whose evaluation for verification purposes is not supported by the language itself and whose resolution for sizing purposes is not considered. This language is designed to verify that an instance of a model conforms to its model when it is entered; it is not designed to solve design problems.

The IEEE1220 [10] standard is representative of the solution-oriented approach. Its chapter "system definition stage" recommends a development process based on checking and system analysis.

Conversely, the recent ISO/IEEE 42020 standard [11] recommends, in the preliminary stages of system development, to conceptualize the system architecture by means of a representation of the problem space and then to determine its admissible technical solutions. It is obviously representative of a synthesis approach and our work fully fits into it.

More precisely, we are interested in the description of a language for modeling the design problems of technical systems in order to solve them. The systems considered are highly structured and can be either predominantly technological (mechanical, electrical, electronic, energetic systems, …) or either software intensive (embedded systems) or mixed (cyber physical systems).

The name modeling language is generally insufficient: We model something in order to do something with what has been modeled. In practice, we distinguish between system modeling languages and problem modeling languages. For the field of system design there are:

- languages used to model a system in order to manage it during the design project, or to evaluate its performance.

- languages for modeling a design problem with a view to its automatic resolution, i.e., to generate correct systems by construction.

1.2 Research goal

Our work falls within the scope of the second category of languages. Languages coming from mathematical programming and operational research allow a flat representation of a problem to be solved using variables, matrices, vectors, equations and inequalities to be satisfied. They also enable the mathematical formalization of an objective function to be optimized. Conversely, they have no structuring and abstraction capacity which does not allow an explicit representation of the structures inherent to the architecture design problem modeling activity. Our research goal is to propose a language built for modeling design problem for automatic computer resolution. The aim of this language is to fill the gap between current flat modeling languages and the need for sophisticated representation of structured problems.

We present in this paper the DEPS language. The purpose of DEPS is to allow the user first to formally pose a given design problem and then, after compilation, either to solve it, i.e., to find one or more solutions if they exist, or to optimize it, i.e., to find the best solution(s) in relation to given criteria. DEPS is a declarative language, based on models and properties that facilitate the modeling of system design problems. On the one hand, the structure and abstraction features allow to represent the architecture (the structure) of the system to be designed while the use of properties using mathematical concepts (parameters, unknowns, equations, inequalities) allows to pose the design problems associated with it: sizing, configuration, resource allocation etc.

1.3 Structure of the paper

The paper is organized as follows: Sect. 2 explains our motivation. The needs are exposed in Sect. 2.1, an analysis of the state of the art is done in Sect. 2.2, and the genesis of the DEPS project is presented in Sect. 2.3. In Sect. 3, we describe the language, focusing in particular on its capacities to express value domains, physical quantities, structural aspects and properties. Section 4 focuses on language constants and variables. In Sect. 5, the way of dealing with model instances is described in detail. Section 6 describes how to structure a problem and create relationships between model. The various properties that can be used in the language are described in Sect. 7. Section 8 presents the modeling and resolution environment called DEPS Studio which, through an implementation of the language, includes model edition, compilation and generation of solutions. Section 9 is dedicated to the validation of the approach through two case studies. Finally, Sect. 10 describes the ongoing work as well as some perspectives of future evolution.

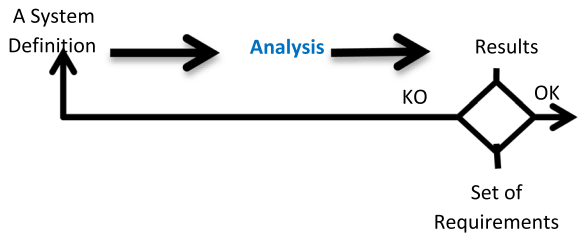

Fig. 1 Analysis approach [17]

2 Motivations

2.1 Representation and resolution needs for model-based system synthesis

The targeted systems of the MBSE that we consider, are physical, software-intensive or mixed (embedded, mechatronics, cyber-physics) systems.

We have seen that the classical MBSE approach is an analysis approach that does not model the problem but describes a solution. The classical calculation process associated with the definition model often integrates a loop for evaluating the performance of the described system to check whether the requirements of the specifications are met. In case of failure, certain parameters of the described system are modified to produce another system and so on inside a loop (cf Fig. 1).

Architecture description languages (ADL) all have in common that they represent a system architecture produced by the designer without any guarantee as to its eligibility. Once modeled, the system must be evaluated according to a point of view adopted by the language in order to verify the requirements considered. Some languages are generalist, others are more specific to the architecture of the system and/or the requirements to be verified. As described in the INCOSE SE vision 2035 [12], ADLs allow to formalize the result of the architecture activity. On the one hand, for several years, work on systems engineering has led to the emergence of the SysML [3] system modeling language based on UML [2] and OCL [9]. Initially, a model realized in UML was supposed to represent a computer application, i.e., a solution to a given problem which one wishes to solve by the realization of a software. In the same way, SysML makes it possible to describe a system architecture that can be solved from several points of view using different diagrams. Recent works on SysML V2 [13] propose a dedicated formal language independent of OCL, but still position itself as the vector of a description of architecture and requirements, with the aim of evaluating this described architecture. On the other hand, several domain-specific modeling languages (DSMLs) have emerged in recent years. They depend on the type of system to represent and the type of point of view to be addressed. Let us mention, widely and without being exhaustive, Modelica [5] for the representation of physical systems with a view to simulating their dynamic behavior, AADL [4] for describing architectures of embedded systems, DEVS for discrete event systems [14], B and Event-b for real-time systems [15] or S2ML [16] for modeling systems with a view to their safety. Because of their common founding myth "to describe a system or an architecture without having formal assurance on the satisfaction of the requirements of the specifications", sometimes bidirectional bridges have been established between these languages. In this context, the design process is conducted in an iterative manner, possibly using parametric optimization tools. The evaluation and/or proof software dedicated to each formalism acts as an evaluation function in the implementation of the optimization loop. These languages are thus adapted to express an architectural solution to be evaluated.

If we wish to work in the problem space and solve the problems encountered in system design, we will need to address the following types of problems [17]:

- System sizing (PT1): problem for which the architecture of the system is known but its dimensions are not (e.g., the length of an object). Unknowns are variables that are often continuous, sometimes discrete. Functional requirements can be quite complex and can be expressed as linear or nonlinear algebraic relationships between constants and variables.

- System configuration (PT2): configuration problems involve the choice of components based on a set of compatibility relationships, options, and cardinality. They are most often discrete-dominant problems.

- Resource allocation (PT3): allocation problems involve allocating physical resources to system functions (or deploying system functions on a set of resources) based on a set of functional and non-functional requirements.

- Architecture generation (PT4): System architecture problems combine the three previous problems. They are based on a specification combining requirements and constraints to produce architectures that will meet the specification. We can also talk about architecture synthesis.

To represent and solve such design problems, we will need additional capabilities to:

1. Formalize a sub-defined or partially defined system, i.e., one with degrees of freedom both from the point of view of the possible value ranges (discrete or continuous) for unknowns but also from the point of view of optional parts of the system structure (choices about components …). Partially defined system modeling includes structural and behavioral modeling.
2. Formalize the functional and non-functional requirements on the system in terms of declarative properties.
3. Solve the problem posed by finding values for the unknowns that are compatible with the declared properties.

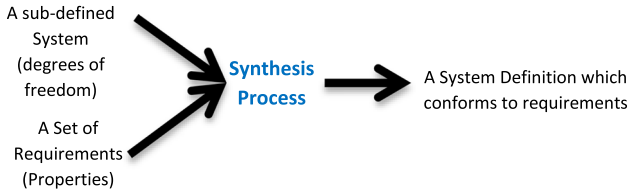

Fig. 2 Model-based system synthesis approach (MBSS) [17]

As summarized in Fig. 2, this synthesis activity is complementary to the usual MBSE analysis and performance evaluation activity that [17] called MBSS for model-based system synthesis. The MBSS approach supports the idea of modeling the problem rather than the candidate solution and then solving the problem by producing correct-by-construction solutions instead of evaluating the performance of a candidate solution.

Of course, the analysis approach (cf Fig. 1) and the MBSS point of view (cf Fig. 2) are complementary. MBSS is suitable for preliminary system design phases, while system modeling and analysis tools are more suited to detailed design phases. They enable detailed analyses and simulations that are not possible with the algebraic tools of MBSS.

Several authors such as [6, 18–21] have shown the limits of the "analysis approach" in the context of the activity of representation and resolution of the problem of architecture synthesis, showing the need for a language of description of the problem for its resolution. Thus another approach is possible (as illustrated in Fig. 2), in which one starts from a sub-definite system architecture and a representation of the requirements to be satisfied and in which, with the help of a solution generation tool, one tries to obtain one or several architecture descriptions correct-by-construction.

A subdefinite architecture is an architecture for which the values of the design variables are not fixed. This approach requires a description language of the problem to solve. In theory, these kinds of languages are rather adapted to the design stage of the system architecture by allowing the automatic production of pre-sized and configured architectures that necessarily meet the expressed requirements. On the one hand, languages coming from mathematical programming and operational research allow a flat representation of a problem to be solved using variables, matrices, vectors, equations and inequalities to be satisfied. They also enable the mathematical formalization of an objective function to be optimized. Let us mention the OPL [22], AMPL [23], MiniZinc [24] and GAMS [25] languages as the main representatives of this type of formalism.

From our point of view, they suffer from a lack of structuring and abstraction capacity which does not allow an explicit representation of the structures inherent to the architecture design activity. On the other hand, work on the representation of software product lines has led to the emergence of interesting problem description languages for design. Thus, Clafer [19] proposes a unified model based on objects and features and allows the user to easily express logical relationships between these elements. Unfortunately for solving the full range of design problems, these languages only address configuration issues.

The DEPS language that is the subject of this paper is a proposed solution to tool the MBSS approach from the point of view of the representation of the design problem.

In addition, due to the typology of problems to be covered we will need to work both on real and integer variables. This point will be illustrated in Sect. 9.

2.2 Related work

The idea of combining object-oriented approaches and computational methods to propose high level declarative problem solving formalisms in engineering sciences is not new. In the following, we describe several related works, distinguishing, on the one hand, from the precursory works that can be linked to the general problem of knowledge representation in artificial intelligence and, on the other hand, from the more recent works undertaken in systems engineering and which is are related to the development and diffusion of MBSE.

In ThingLab [26], constraints stored in the class of an object authorize the expression of relations. One can express, for example, that any resistance obeys Ohm's law. However, since ThingLab was designed to produce animated simulations, the solving methods used, which are sufficient in this field, are ineffective for solving the systems of equations and inequalities of engineering design problems. Moreover, in ThingLab, a constraint is declared in an object and its behavior is programmed in methods of this object. This procedural character of the representation of constraints makes it impossible to consider ThingLab as a declarative modeling language.

The NEMO-TEC language [27] is described as an object-oriented extension of the Unicalc mathematical problem solver based on the technology of sub-defined models. This technology is comparable to interval constraint programming methods. NEMO-TEC was essentially designed to generate a stand-alone computing system, consisting of a compiled model of the dependency network between the unknowns of the problem and a user interface. This system must guide a user in his successive design choices and validate them by propagation. The interactive nature of the design is prioritized to the detriment of the resolution capacity in this work.

ONERA, in the framework of a research work in preliminary aircraft design, has proposed to combine logic programming with constraints (CLP) and object concepts to represent and solve design problems [28, 29]. Logic programming brings the capabilities of tree searching and logic rules writing; its generalization to the CLP framework brings the capabilities of formulating and solving mathematical

constraints on discrete or continuous variables. The object-oriented concepts (classes, attributes, instances) enable to capture the structure of the product (i.e., its decomposition in classes) and to use it to propose configurations (i.e., different instantiations of the product). The proposal mixes in the object representation the procedural features of predicates with the declarative features of clauses. Additionally all the equations must be decomposed into binary and ternary predicates. Finally although the proposed implementation makes it possible to simulate object features in a Prolog program, the framework still remains fundamentally that of logic programming with constraints. Moreover, the scope remains limited to the expression of configuration problems and the representation of mathematical concepts remains procedural. This is neither a full-fledged object language nor a true declarative mathematical modeling language.

The COB language [30] developed at about the same time at the New York University in Buffalo also proposes to combine constraints and objects by relying on the resolution capabilities of a logic programming language with constraints, in this case CLP(R). But it differs significantly from the previous development by a number of points which constitute notable advances. COB is a true language. Its syntax is formally specified by a grammar and its operational semantics is validated by the development of a compiler [31]. This compiler translates a COB program into a CLP(R) program. Unfortunately COB is still too much influenced by CLP(R). In addition to attributes and constraints, a class is also composed of predicates and constructors. Finally COB is more like an attempt at a general multi-paradigm programming language (classes and objects, methods, constraints and rules) than a specific modeling language dedicated to the engineer.

In the lineage of predicate-based languages, the Alloy software system specification language [32] is worth mentioning. Alloy consists of a declarative language inspired by the Z language and its analyzer. It associates first order logic with relational capabilities. The Alloy Analyzer is a fully automatic tool that finds instances of Alloy specifications (or Alloy model), i.e., it is able to assign values to the sets and relationships of the specification so that for each assignment all formulas of the specification are valid. It transforms the set of models created in Alloy into a SAT problem. Alloy is a true language with a compiler. Unfortunately, it only handles Boolean variables without possible resolution on real ones.

The s-COMMA [33, 34] platform is an attempt to use UML to produce an environment combining modeling and resolution. The idea is to use the capabilities of the metamodeling and model transformation tools associated with the UML standard to build a visual and object-oriented language for modeling constraint satisfaction problems. The platform is composed of two main parts, a modeling tool and a projection:

- The visual modeling language combines the declarative aspects of constraint programming with the structuring aspects of object-oriented programming:
- The projection tool is a translator that transforms the modeling of a problem described in the previous language into a constraint programming program dedicated to a targeted solver.

The advantages of the approach are:

- A certain independence of the problem modeling language from the solver languages;
- A development of projection tools on new solvers facilitated by the use of model-driven development tools such as KM3 (Metamodel Specification Language) or ATL (Model Transformation Rule Description Language).

However, the model-driven development approach used in s-COMMA presents some drawbacks. The description of the compilation architecture presented in the user manual shows us that it is organized in three layers (modeling, projection, resolution) and that two transformations are needed to go through them: the s-comma object model is first compiled into a flat-s-comma model with a syntax closer to that of the target resolution languages, then this flat-s-comma model will itself be compiled into the target language of a given solver.

This complex compilation chain decouples the problem description model from the solution finding and poses the fundamental question of the debugging of the model.

The Deklare [35] and KoMod [36] projects focused on the representation of the functional and organic structures of the products to be designed in order to enable the associated design problems to be posed and solved using a commercially available constraint programming library.

The objective of the CO2 (constraint-based design) project (a Grant from the French National Research Agency) was to develop methods and tools to solve design problems. The project produced Constraint Explorer (CE), a problem solving software environment comprising a modeling language and a numerical solver based on interval methods [18]. The solving characteristics and performances of CE are relevant, but the language lacks the object-oriented structuring features required for system design.

Clafer [19], a recent proposal seems particularly interesting. It proposes an object-based and feature-based model and allows the user to easily express logical relationships between these elements. A Clafer model can be transformed into an

intermediate format [20] so that it can be taken into account by the constraint programming library Choco or by Alloy [19]. Clafer remains for the time being oriented towards the representation of configuration problems or software product lines and deals essentially with discrete problems. In addition, the environment requires the use of an external solver to solve the models.

Finally, let us mention the work around the taking into account of uncertainty in software engineering. [21] talks about partial models represented in the form of graphs with alternatives and a resolution process using an external SAT solver. However, they do not propose any formal language. Moreover, only Boolean degrees of freedom can be represented.

2.3 Our proposal

The current needs expressed in preliminary design, particularly in the field of the generation of architecture of eligible systems, are not covered [1, 37]. This observation led us to start this work of specification and development of a declarative language adapted to system design.

Our proposal to meet these needs is:

- A native language, declarative, structured and property based, offering the possibility to describe an ontology of physical quantities and enabling the representation of variables and properties in discrete and continuous domains.
- An Integrated Modeling and Solving Environment enabling model edition, compilation and resolution within the same integrated environment.

As partial system descriptions are naturally more frequent in the upstream phases of system definition, DEPS is a natural candidate for preliminary design even if its use is not limited to this stage. In preliminary design, DEPS Studio can be seen as a synthesis tool that proposes eligible systems as opposed to analysis tools used at later stages to verify or validate a system.

We describe the main features of the language in the remainder of this paper.

3 DEPS fundamental features

In this section, after giving an overview of DEPS, we describe the fundamental elements of the language, which will serve as a basis for the rest of the paper.

3.1 Overview

In DEPS, a design problem is represented by a set of DEPS models so the fundamental feature of the DEPS language is the "Model". Any model encapsulates in order: a set of arguments, a set of constants, a set of variables, a set of elements and a set of properties. Arguments can be either constants or elements identifiers. Elements are instances of other models.

Figure 3 gives a representation of the DEPS meta-model and synthesizes all the concepts that will be presented in the rest of this paper.

3.2 Models

The fundamental feature of the DEPS language is the model. Any model is defined using the keyword *Model* followed by its name and its (possibly empty) list of arguments. It contains in order: an area for declaration or definition of model constants (keyword *Constants*), an area for definition of model variables (keyword *Variables*), an area for declaration or definition of model elements (keyword *Elements*) and an area for definition of model properties (keyword *Properties*).

In the following, we will agree that:

- The notion of declaration refers to the description of the type of the constant or element passed as an argument of a model but constructed outside of it,
- The notion of definition refers to the construction of the constant or the element inside a model.

A Model is defined according to the following syntax:

```
<Model> ::=
        Model <ModelName>( <listOfArguments> )
        <ModelOptions>
        Constants  <listOfConstantDeclarationOrDefinition>
        Variables <listOfVariablesDefinition>
        Elements   <ListOfElementsDeclarationOrDefinition>
        Properties <ListOf Properties>
        End
```

With

```
<ModelOptions> ::=
                    |
                    abstract
                    |
                    extends <ModelSignature>
                    |
                    abstract extends <ModelSignature>
```

A model can be qualified as abstract (optional keyword *abstract*). In this case, it cannot be instantiated. An instance

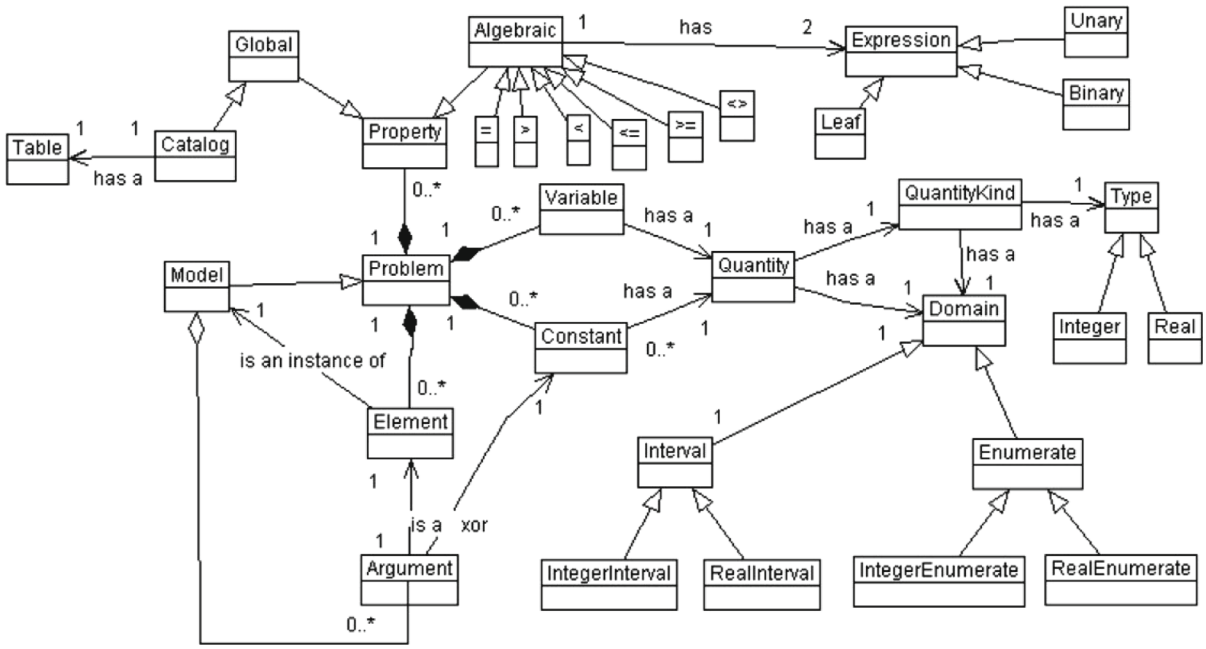

Fig. 3 A part of the meta-model of the current version of the DEPS language

of a model is called an Element. Therefore, no instance of an abstract model can be displayed in the *Elements* area of a model.

It is possible to extend models (*extends* keyword). This functionality will be detailed in Sect. 6.1.

The arguments of a model can be either values, constants or instances of other models (Elements) and nothing else. Any constant argument of a model will have its declaration in the Constants area of the model. Any element argument of a model will have its declaration in the Elements area of the model. These points will be detailed in Sect. 6.2.

3.3 Problem

The problem to be solved is expressed using the keyword *Problem*. Syntactically, a problem is a Model without arguments. For each system design project to be represented in DEPS, there is one and only one *Problem* defined. The problem is broken down into *Elements*. Each Element is an instance of a Model. Since each Model can also be made up of Elements, the Problem is the root of a hierarchy of Elements.

Then, a Problem is defined according to the following syntax:

```
<Problem> ::=
        Problem    <ProblemName>
        Constants  <listOfConstantDeclarationOrDefinition>
        Variables <listOfVariablesDefinition>
        Elements   <ListOfElementsDeclarationOrDefinition>
        Properties <ListOf Properties>
        End
```

3.4 Quantity kind and quantity

Data manipulated in DEPS can be:

- Integer or real values.
- Integer intervals, real intervals, integer enumerated domains, or real enumerated domains.

If we stick to these simple types, we cannot represent the quantities manipulated by the designers of technical systems. Therefore, we have proposed two concepts to represent these quantities: *QuantityKind* and *Quantity*.

Several research works have focused on the integration of quantities in models [38]. These works are based on the definition of UML stereotypes intended to represent quantities as well as the associated operators in the form of member functions. Several other implementation of the concept of quantities exist: QUDV for SysML [39], the QUDT ontology [40] from NASA and the representation of quantities in the Modelica language too [41]. Our proposal is different on several points:

- We make a clear distinction between the *Quantity* that will carry the unit and the *QuantityKind* that will carry the dimension of the quantity.
- Our approach is purely declarative and we do not use member functions. The operations between quantities are directly managed by the DEPS Studio environment that we

```
1  QuantityKind PotentialDifference
2  Type : real ;
3  Min  : 0 ;
4  Max  : +maxreal ;
5  Dim  : ML2T-3I-1 ;
6  End
```

Fig. 4 QuantityKind example

```
1  Quantity Voltage
2  Kind : PotentialDifference ;
3  Min  : 0;
4  Max  : +maxreal ;
5  Unit : V ;
6  End
```

Fig. 5 Quantity example

have developed and that is able to do algebraic operations guaranteed on the domains of possible values of quantities.
- We want to be able to represent both ordinal and cardinal quantities.
- Our aim is not to describe quantities in all their generality, but to provide just enough information to be able to correctly type the constants and variables of the problems we wish to represent in DEPS.

Note that a set of universal quantities and quantity kinds of physics have been predefined in DEPS: Pressure, temperature, electric current, mass, length, power, …

The basics quantities *Real* and *Integer* are also predefined.

They are all included in a dedicated package available with the language.

3.4.1 Quantity kind

In DEPS, both constants and variables are associated with types of physical or technological quantities called quantities (Quantity). They are mandatory in Systems Engineering.

A *QuantityKind* carries a basic type (integer or real), a *min* limit, a *max* limit as well as the dimension in the sense of the dimensional analysis of the quantity [42, 43]. For example L is the dimension of a length, LT^{-2} is the dimension of an acceleration where M represents a mass, L a length and T a time. We can also define dimensionless Quantitykind as processor indexes or other. In this case, the dimensionless symbol u will be used.

The negative and positive infinite values for the domain terminals are noted *-maxreal* and *+ maxreal* for real and *- maxint*, *+ maxint* for integers.

A QuantityKind is defined according to the following syntax:

```
<QuantityKind> ::=
        QuantityKind  <QuantityKindName>
        Type : real | integer ;
         Min  : <ConstantExpression> | -minreal | -minint ;
        Max  : <ConstantExpression> | -maxreal | -maxint ;
        Dim  : <DimensionalExpression>;
        End
```

Figure 4 shows a *QuantityKind* representing a potential difference (*PotentialDifference*). A potential is therefore a zero (*Min*) or positive unbound real quantity (*Max*). According to the dimensional analysis, the dimension of an electric potential is $ML^2T^{-3}I^{-1}$ where M represents a mass, L a length, T the time, and I the electric intensity [42, 44].

3.4.2 Quantity

A *Quantity* is defined from a *QuantityKind*. The *QuantityKind* carries the dimension, and the *Quantity* carries the unit.

More precisely, a *Quantity* has:

- A base quantity type (*Kind*). For example, Real, Integer, Length;
- A *Min* (resp. *Max*) bound that represents the minimum (resp. maximum) value that can be taken by any constant or variable having the defined quantity as its type;
- A *Unit* of the quantity.

Let us assume that:

- $D_{\langle \text{Quantity Name} \rangle}$ represents the domain of values of the Quantity
- $D_{\langle \text{Quantity Kind Name} \rangle}$ represents the domain of values of the QuantityKind

The definitions of the quantity has to satisfy:

$$D_{\langle \text{Quantity Name} \rangle} \subseteq D_{\langle \text{Quantity Kind Name} \rangle}$$

A Quantity is defined according to the following syntax:

```
<Quantity> ::=
        Quantity <QuantityName>
        Kind : <QuantityKindName> ;
         Min  : <ConstantExpression> | -minreal | -minint ;
        Max  : <ConstantExpression> | -minreal | -minint ;
        Unit : <UnitValue> ;
        End
```

For example, the meter m can be the unit for a length and the volt V can be the unit of an electrical potential or voltage. The symbol u is used to designate quantities without a unit (cf Fig. 5).

Separating *Quantity* and *QuantityKind* allows several quantities to refer to the same quantitykind. The quantitykind

```
1   Table Battery
2   Attributes
3     ref :    Index;
4     V :      Voltage;
5     Imax : Current;
6   Tuples
7   [1, 12, 300],
8   [2, 12, 400] ,
9   [3, 6, 300],
10  [4, 6 , 200]
11  End
```

Fig. 6 An example of table in DEPS

carrying the dimension and the quantity carrying the unit, it will be possible to express the same quantity (same dimension) in several units.

Tables are tables of correspondence between values. A table has a name, a set of typed attributes and a dataset organized in tuples. A table ends with the keyword End.

Tables are defined according to the following syntax:

```
<Table> ::=
        Table  <TableName>
        Attributes
        <a1> : <Quantity1>;
        <a2> : <Quantity2>;
        ...
        <an> : <Quantityn>;
        Tuples
        [<va11>, <va21>, ..., <van1>] ,
        [<va12>, <va22>, ..., <van2>] ,
        [<va13>, <va23>, ..., <van3>] ,
        ...
        [<va1p>, <va2p>, ..., <vanp>]
        End
```

The table in Fig. 6 represents a set of tuples for the characteristics of batteries. It describes for each battery referenced by an Index the available combinations for the voltage and the maximum intensity.

For example, the battery whose reference (*ref*) value is 2 will have a voltage of 12 V and a maximum delivered current of 400A.

4 Constants and variables

As mentioned in Sects. 3.2 and 3.3, models are made up of constants and variables. This section focuses on the description of these elements and their interpretation mechanisms.

4.1 Constants

A constant is a numerical quantity whose value does not vary during the lifetime of any copy of the model in which it is declared or defined. A defined constant is a constant local to the model which is defined in its *Constants* area and whose value is that of a local expression to the model. A declared constant is a constant local to the model which is defined in its *Constants* area and whose value is that of an expression passed as an argument of the model. In addition, some universal constants are predefined in DEPS. Thus, Pi represents the constant whose value is that of the transcendental number π.

When they are local to a Model, constants are defined according to the following syntax:

```
<Constant> ::=
     <name> : <quantity> = <value>  <ConstantOption> ;
    |
    <name> : <quantity> in [ <vmin> , <vmax> ]
    = <value> <ConstantOption>  ;
    |
    <name> : <quantity> in { <v1>, ..., <vn> }
         = <value> <ConstantOption>  ;
    |
    <name> : <quantity> = <constant expression> <ConstantOption>  ;
    |
    <name> : <quantity> in [ <cexprmin> , <cexprmax> ]
    = <constant expression> <ConstantOption> ;
    |
     <name> : <quantity> in { <v1>, ..., <vn> }
    = <constant expression > <ConstantOption>  ;


<ConstantOption> ::=
          |
          default
          |
          redefine
```

Constants values and quantities can be set by default and can be redefined too. This point will be detailed in Sect. 6.5

When they are passed as arguments of the Model, constants are declared according to the following syntax:

```
<Constant> ::=
      <name> : <quantity>;
     |
     <name> : <quantity> in [ <vmin> , <vmax> ] ;
     |
     <name> : <quantity> in { <v1>, ..., <vn> } ;
     |
     <name> : <quantity> in [ <cexprmin> , <cexprmax> ] ;
     |
      <name> : <quantity> in { <v1>, ..., <vn> } ;
```

Figure 7 shows some examples of definition of constants. Note that some constants may depend on other previously defined constants both for the definition domain and for the value.

4.2 Variables

A variable is an unknown in the model. It is characterized by its Quantity possibly restricted to a sub-area of possible values. A variable must be defined in the *Variables* field of the

```
1  Constants
2  a : Real = 3.5 ;
3  b : Real in [0,  , 20 ] = a^2 ;
4  c : Real in { -10.2, 0.0, 1.5, 2.3} = 1.5 ;
5  d : Real = a*b+ln(c) ;
6  e : Real in [ 2*a , 2*a+10 ] = 2*a+1 ;
7  f  : Real in { a+1, a+3 , a+5} = a+3;
```

Fig. 7 examples of definition of constants

```
1  Constants
2  a : Real = 3.5 ;
3  Variables
4  V1 : Real;
5  V2 : Real in [0, 100];
6  V3 : Real in {1.5, 3.5, 5.5};
7  V4 : Real in {a+1, a+2, a+3};
8  V5 : Real in [abs(a)*2, abs(a)*3];
```

Fig. 8 examples of definition of variables

model. As an unknown, its value is not defined. It is important to point out that the variables carry the sub-defined character of DEPS models.

Variables are defined according to the following syntax:

```
<Variable> ::=
  <VariablePrefix> <name> : <quantity> <VariableOption>  ;
  |
  <VariablePrefix> <name> : <quantity> in [ <vmin> , <vmax> ] <VariableOption>;
  |
  <VariablePrefix> <name> : <quantity> in { <v1>, <v2> , ..., <vn> }  <VariableOption> ;
  |
  <VariablePrefix> <name> : <quantity> in
        [ <cexprmin> , <cexprmax> ]  <VariableOption> ;


<VariablePrefix> ::=
        |
        expr
        |
        obj

<VariableOption> ::=
        |
        redefine
```

Variables quantities can be set and can be redefined. They can be prefixed too. These points will be detailed in Sect. 6.5.

Figure 8 shows some examples of definition of Variables. Note that some variables may depend on previously defined constants for the definition domain.

Let us define for example an electric dipole model (see Fig. 9). A dipole is characterized by the current flowing through it (I) and by the voltage at its terminals (U). The possible values of the voltage U have been reduced to the continuous range *[0, 1000]*. This model is qualified here as abstract. It cannot be instantiated. In DEPS, model instances are called Elements.

```
1  Model Dipole() abstract
2  Constants
3  Variables
4  U : Voltage in [0, 1000];
5  I : Current;
6  Elements
7  Properties
8  End
```

Fig. 9 A dipole abstract model

5 Elements

As mentioned in Sects. 3.2 and 3.3, elements are parts of models. This section focuses on the different ways of handling elements. An *Element* is an instance of a model. It can be passed as an argument to a model for implementing aggregation relationship. It can also be defined in a model for implementing composition relationship (see Sect. 6.2). An *Element* can be defined or declared as follows:

```
<Element> ::=
        <DefinedElement>
        |
        <DeclaredElement>
```

5.1 Defining an element

When they are local to a Model, Elements are defined according to the following syntax:

```
<DefinedElement> ::=
        <name> : <ModelName>( <listOfArgValues> ) <ModelOption> ;

                                        <ModelOption> ::=
                                                |
                                              redefine
```

< *ListOfArgValues* > is the list of argument values needed to define the element. This list must be compatible with the signature of the model < *ModelName* > of which the element is an instance (cf Sect. 6.3). Moreover, models can be redefined in a certain limit. This point will be detailed in Sect. 6.5.

5.2 Declaring an element

When they are passed as arguments of the Model, elements are declared according to the following syntax:

```
<DeclaredElement> ::=
    <name> :<ModelSignature>;
```

```
1  Model A(c1, c2)
2  Constants
3  Variables
4  c1: Real;
5  c2 : Integer;
6  Elements
7  Properties
8  End
9
10 Model B(DeclElt)
11 Constants
12 cb1 : Real = 3.5;
13 cb2 : Integer = -1;
14 Variables
15 Elements
16 DeclElt : A[Real, Integer];
17 DefElt : A(Cb1, Cb2);
18 Properties
19 End
```

Fig. 10 examples of declared and defined elements

5.3 Model signature

Each Model has a signature. Signatures allow the designer to define several models with the same name, different contents as soon as their signatures are different. A signature is declared according to the following syntax:

<ModelSignature> ::=
<ModelName> [<listofQuantitiesOrModelSignatures>];

In Fig. 10, they are defined:

- A model *A* with two arguments: a real constant (*c1*) and an integer constant (*c2*),
- a model *B* with one argument named *DeclElt* which is an instance element of the model *A*. We will therefore specify its signature *A[Real, Integer]* in its Elements part. Inside the model *B* is also defined in the elements area an instance of *A* named *DefElt,* created with the arguments *cb1* and *cb2*: *A(cb1, cb2).*

Then, as *B* has one argument which is an instance of A which signature is *A[Real, Integer],* the signature of *B* model is:

B[A[Real, Integer]].

6 Organization of the models

DEPS models can be combined with each other using different relationships that we will now detail.

```
1  Model Resistor() extends Dipole[]
2  Constants
3  Variables
4  R : Resistance ;
5  Elements
6  Properties
7  U = R*I;
8  End
```

Fig. 11 A resistor model

```
1  Model Resistor(R) extendsDipole[]
2  Constants
3    R : Resistance ;
4  Variables
5  Elements
6  Properties
7  U = R*I;
8  End
```

Fig. 12 A parameterized resistor model

6.1 Inheritance

DEPS models can simply inherit from each other (keyword extends). This is public inheritance: constants, variables, elements and properties are thus directly inherited.

So be *MExtends* an extended model of a given model *M*, any instance of *MExtends* will contain:

- The arguments of*M* as well as the locally declared arguments of *MExtends.*
- The constants defined or declared in the *Constants* area of *M* as well as the constants defined or declared in the *Constants* area of MExtends.
- The variables defined in the *Variables* area of *M* and the variables defined in the *Variables* area of MExtends.
- The elements defined or declared in the Elements area of M as well as the elements defined or declared in the *Elements* area of MExtends.
- The properties expressed in the *Properties* area of M as well as the properties expressed in the *Properties* area of MExtends.

We thus define in Fig. 11 a first resistor model. A resistor is a dipole with an unknown ohmic resistance (*R*) and subject to Ohm's law.

We can also define a second resistor in which the resistance R is a declared constant whose ohmic value is passed as an argument to the model (cf Fig. 12). In this case, the model is parameterized.

```
1  Model Serial(D1, D2)
2  Constants
3  Variables
4  expr U : Voltage ;
5  expr I : Current ;
6  Elements
7  D1 : Dipole[];
8  D2 : Dipole[];
9  Properties
10   U := D1.U+D2.U;
11   D1.I = D2.I;
12 I := D1.I;
13 End
```

Fig. 13 Serial model of two dipoles

```
1  Model Serial(R1, R2)
2  Constants
3  Variables
4  expr U : Voltage ;
5  expr I : Current ;
6  Elements
7  R1 : Resistor[];
8  R2 : Resistor[Resistance];
9  Properties
10   U := R1.U+R2.U;
11   R1.I = R2.I;
12 I := R1.I;
13 End
```

Fig. 14 Serial model of two different types of resistors

6.2 Composition and aggregation

An element can be passed as an argument to a model to represent an aggregation and must then be declared in the Model Elements field area. It can also be created in a model to represent a composition and must then be defined in the Model Elements area.

When an element is passed as an argument to a model, it is simply declared in the Elements area of the model. It is declared by typing it by the signature of the model to which it refers.

To define an element, it is constructed by calling the name of the model to which it refers followed by the list of its effective arguments.

Thus, if in our example we wish to associate two dipoles in series, we will define a serial link model between two aggregated dipoles (cf Fig. 13).

6.3 Polymorphism and genericity

As previously mentioned in Sect. 5.3, each Model has a signature. Thus the signature of the Serial Model will be denoted Serial [Dipole[], Dipole[]]. This mechanism removes ambiguity in case of aggregation of Models with the same name but different signatures.

The signature of a model consists of its name followed by the list of signatures of its arguments. If the argument is a constant its signature is its quantity.

This notion of signature allows the overloading of models: several models can have the same name as long as their signatures are different.

Thus if we now wish to have a resistance model in series specifying explicitly that the first dipole of the aggregation is a resistor of unknown Ohmic value and that the second dipole of the aggregation is a resistor of known value we will have to remove the ambiguity by means of the signature of each of the resistance models (cf Fig. 14).

```
1  Model Resistor(R0, T) extends Dipole[]
2  Constants
3  R0 : Resistance ;
4  T : Temperature;
5  alpha : Real = 3.91e-3;
6  R : Resistance = R0*(1+alpha*T);
7  Variables
8  Elements
9  Properties
10 Properties
11  U = R*I;
12  End
```

Fig. 15 A temperature-dependent resistor model

Thus, in Fig. 12, the model *Resistor* has an ohmic resistance value as argument. Its signature is therefore *Resistor*[*Resistance*]. Figure 14 shows a serial connection model between a resistor of unknown ohmic value and a resistor of given ohmic value. The signature of the Serial model is therefore Serial[Resistor[], Resistor[Resistance]]. Let us assume that we would like to model a new resistor which depends on temperature (cf Fig. 15) and that we want to connect with a serial connection this resistance to the two previous one. We will create a new model that will extend the two-resistor series model into a three-resistor model, the third of which will be a temperature-dependent resistor (see Fig. 16). The signature of this new Serial model is therefore Serial[Resistor[], Resistor[Resistance], Resistor[Resistance, Temperature]].

Thus if we wish to construct an element *mSerial,* instance of the new *Serial* model, we will have to pass it in order:

- An argument of type instance of a two arguments serial model inherited from Serial[Resistor[], Resistor[Resistance]].

```
1  Model Serial(R3)
2  extendsSerial[Resistor[],Resistor[Resistance]]
3  Constants
4  Variables
5  expr Us : Voltage ;
6  Elements
7  R3 : Resistor[Resistance , Temperature];
8  Properties
9    Us := R3.U+U;
10   R3.I = R2.I;
11 End
```

Fig. 16 Serial model of three different resistors

```
1  Model SysElec(G)
2  Constants
3  Variables
4  Elements
5  G : VSource[] ;
6  R1 : Resistor() ;
7  R2 : Resistor(10);
8  S : Serial(R1, R2);
9  Properties
10  G.V = S.V;
11 G.I = S.I;
12 End
```

Fig. 17 Model of the electric system

- An argument of type instance of a temperature-dependent resistor whose signature is Resistor[Resistance, Temperature].

Now let assume we want to create an electrical system containing two resistors in series, one known and one unknown, connected to a given voltage source. We will create a SysElec model with a voltage source (*G*) as argument. This model (see Fig. 17) will be composed of a resistor with a known Ohmic value (10 Ohm), a resistor with an unknown Ohmic value and a serial link between these two dipoles. *R1* and *R2* being instances of the Resistance Models, themselves derived from *Dipole*, we can pass these resistors as an argument of the *Serial* model.

6.4 Access to model elements

All the elements of a problem are organized using aggregation and compositional relationships forming a tree structure. Access to the elements of this structure is authorized by the use of a doted notation.

A constant, variable, or element at different levels of this tree structure can be designated and manipulated using a path.

```
1  Model A(E, b)
2  Constants
3  c : Real in [1, 10] = 3; default;
4  b : Real; default;
5  Variables
6  V : Integer in [0, 100] ;
7  Elements
8  E : C[Real];
9  Properties
10 End
11
12 Model B()
13 extends A[C[Real], Real]
14 Constants
15 c : Real in [1, 5] = 4; redefine;
16 b : Real in [-10, 10]; redefine;
17 Variables
18 V : Integer in [0, 10] ; redefine ;
19 Elements
20 E : C[Real, Real]; redefine;
21 Properties
22 End
```

Fig. 18 Illustration of redefinition of constants and variables

If the current I flowing through resistor *R1* is to be accessed in the model shown in Fig. 14, this is done via the path *R1.I*.

6.5 Constants, variables and elements redefinition

Let us assume that:

$\text{Dredef}_{\langle \text{Const}\mid \text{Var}\rangle}$ represents the domain of values redefined for a constant or a variable in the extended Model.

$D_{\langle \text{Const}\mid \text{Var}\rangle}$ represents the domain of values defined by default for a constant or a variable in the basic Model.

The definitions of the redefined domains have to satisfy:

$$\text{Dredef}_{\langle \text{Const}\mid \text{Var}\rangle} \subseteq D_{\langle \text{Const}\mid \text{Var}\rangle}$$

Then, in Fig. 18, domain and value of the c constant have been redefined, and the domain of the b constant has been redefined. In the same way, the domain of the V variable has been modified. It is also possible to redefine an element (model instance) under certain conditions. Let us take the example of a Model *SysElecWithResistor* extension of a *SysElec* (cf. Figure 19). Initially, the first *SysElec* Model consisting of two elements *R1* and *R2* instances of *Dipole*. It is possible to extend the first *SySElec* model into a *SySElecWithResistor* model in which the elements *R1* and *R2* are redefined (*redefine*) as an instance of Resistor[] and an instance of Resistor[Resistance]. This is possible if and only if Resistor[] and Resistor[Resistance] are extended models of *Dipole*.

```
1  Model SysElec(G)
2  Constants
3  Variables
4  Elements
5  G : VSource[] ;
6  R1 : Dipole() ;
7  R2 : Dipole();
8  S : Serial(R1, R2);
9  Properties
10 G.V = S.V;      G.I = S.I;
11 End
12
13 Model SysElecWithResistor
15 extends  SysElec[VSource]
16 Constants
17 Variables
18 Elements
19 R1 : Resistor() ; redefine;
20 R2 : Resistor(10); redefine;
21 Properties
22 End
```

Fig. 19 Illustration of an Element redefinition

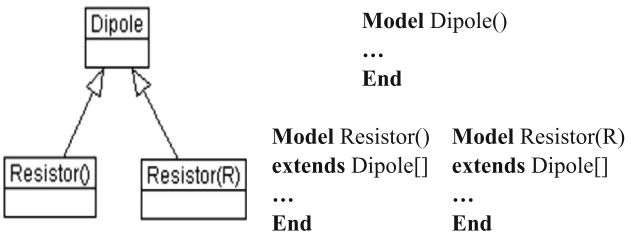

Fig. 20 A very simple hierarchy for demonstration

6.6 Composition, aggregation, inheritance and overloading

It is possible to combine the aggregation, composition, inheritance and overloading capabilities of DEPS models. Consider models *Dipole*, *Resistor*[] and *Resistor[Resistance]* in Fig. 20. *Dipole* is the root model of an inheritance tree. Indeed, *Resistor*[] derives from *Dipole* and *Resistor[Resistance]* derives from *Dipole* too.

The *Agreg* model of Fig. 21 is an aggregation of two instances *d1*, *d2* declared as instances of *Dipole*. Because of the inheritance between models, it is possible to instantiate *Agreg* with as arguments an instance of *Resistor*[] and an instance of *Resistor[Resistance]* as it is the case in the *UseAgreg* model of Fig. 21. The *RootComp* model is composed of two instances *d1* and *d2* of *Dipole*. Because of the inheritance relation, it is possible to extend *RootComp* into an *ExtComp* model by redefining*d1* and *d2* as instances of *Resistor*[] and *Resistor[Resistance],* respectively.

```
1  Model Agreg(m1,m2)
2  Constants
3  Variables
4  Elements
5  d1 : Dipole[];
6  d2 : Dipole[];
7  Properties
8  End

1  Model RootComp()
2  Constants
3  Variables
4  Elements
5  d1 : Dipole();
6  d2 : Dipole();
7  Properties
8  End

1  Model ExtComp()
2  extends RootComp[]
3  Constants
4  Variables
5  Elements
6  d1:Resisor(); redefine;
7  d2:Resistor(20);redefine;
8  Properties
9  End

1  Model UseAgreg()
2  Constants
3  Variables
4  Elements
5  r1 : Resistor();
6  r2 : Resistor(10);
7  a1 : Agreg(r1, r2);
8  Properties
9  End
```

Fig. 21 A very simple hierarchy for demonstration

```
1  Model List(val, next)
2  Constants
3  val : Real;
4  Variables
5  Elements
6  (* neither ending story … *)
7  next : List[Real, List[Real, List[Real,…..
8  Properties
9  End
```

Fig. 22 Wrong definition of self-referencing model

Moreover, this type of mechanism makes it quite easy to have templates. Self-referencing of models is limited. Thus, if one wishes to define a model of chained lists of reals in DEPS, one would be tempted to proceed as in Fig. 22.

And we would be unable to express the signature of the List Model. On the other hand, by using the inheritance mechanism combined with overloading, it becomes possible to define quasi-recursive models (cf Fig. 23).

7 Properties

DEPS is a declarative and property based language. In this way, it is possible to express within a model a set of properties that must necessarily be satisfied by any of the instances of this model. A property is a relationship necessarily respected by any instance of the model that contains it. In the current version of DEPS, properties are either algebraic relationships between expressions (cf. Section 7.3) or relationships defined in extension that use tables of values of variables compatible with each other (cf. Section 7.4).

```
1  Model List( val)
2  Constants
3  val : Real;
4  Variables
5  Elements
6  Properties
7  End
8
9
10 Model List(next) extends List[Real]
11 Constants
12 Variables
13 Elements
14 next : List[Real];
15 Properties
16 End
17
18 Problem CreateList
19 Constants
20 Variables
21 Elements
22 cell1 : List(2.5);
23 cell2 : List(3.5, cell1);
24 Properties
25 End
```

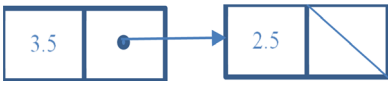

Fig. 23 model self-referencing

sin(<Expression>)	sinus
cos(<Expression>)	cosinus
tan (<Expression>)	tangent
asin(<Expression>)	arc sinus
acos(<Expression>)	arc cosinus
atan(<Expression>)	arc tangent
sinh(<Expression>)	hyperbolic sinus
cosh(<Expression>)	hyperbolic cosinus
tanh(<Expression>)	hyperbolic tangent
asinh(<Expression>)	hyperbolic arc sinus
acosh(<Expression>)	hyperbolic arc cosinus
atanh(<Expression>)	hyperbolic arc tangent
ln(<Expression>)	neperian logarithm
exp(<Expression>)	exponential
abs(<Expression>)	absolute value
sqrt(<Expression>)	square root

Fig. 24 Unary expressions in DEPS

<Expression> + <Expression>	addition
<Expression> - <Expression>	difference
<Expression> * <Expression>	multiplication
<Expression> / <Expression>	division
<Expression> ^ <Expression>	power
min(<Expression> , <Expression>)	minimum
max(<Expression> , <Expression>)	maximum

Fig. 25 Binary expressions in DEPS

7.1 Algebraic expressions

Algebraic expressions can be for integer or real values, integer or real constants and variables of any type (continuous domain, discrete domain or enumerated domain). Usual unary algebraic operators are available such as logarithmic, power, exponential, trigonometric, hyperbolic, … (cf Fig. 24). In DEPS, all algebraic operators are strongly typed. Depending on their operands, they can return either an integer value or a real value or a continuous domain or a discrete domain or an enumeration of integer values or an enumeration of decimal values.

In the same way, the usual arithmetic binary operators are recognized in DEPS as well as the min and max binary operators (cf Fig. 25).

7.2 Piecewise operators

It may be necessary in some design problems to express piecewise functions from $\mathbb{R}$ to $\mathbb{R}$. This is the case when a function is defined continuously by pieces.

As an example, the following function is a nonlinear piecewise function:

$y = f(x)$ defined on [0, 100] such that:

$f(x) = x$ if$x \in [0, 1]$

$f(x) = x^2$ if $x \in [1, 10]$

$f(x) = 1e^3/x$ if $x \in [10, 100]$

In the case where the function to represent is piecewise nonlinear, we have the following operator:

$$\boldsymbol{pw}(< varg >, < I_1 >, < expr_1 >, \ldots, < I_n >, < expr_n >);$$

with < *varg* > being the name of the argument variable of the function. For each *i* from 1 to n, $< I_i >$ represents the ith value domain for < *varg* > and $< expr_i >$ represents the expression of the function on $< I_i >$. Here again, the operator returns a continuous interval.

Then, for representing the previous piecewise function, we will write the following property with the *pw* operator:

$y = pw(x, [0, 1], x, [1, 10]x^2, [10, 100], 1e3/x);$

```
1  Model PieceWiseExample()
2  Constants
3  Variables
4  x : Real;     y : Real;     z : Real;
5  Elements
6  Properties
7  y = pw(x, [0,1], x^2-1, [1, 1e3], ln(x));
8  z + y = cos(x);
9  End
```

Fig. 26 an example of nonlinear piecewise function in DEPS

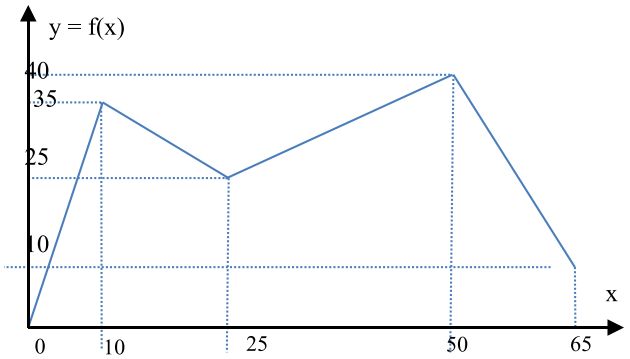

Fig. 27 An example of a linear piecewise function

Figure 26 illustrates the use of a piecewise operator inside a DEPS model.

When the piecewise function is piecewise linear (see Fig. 27), the following operators are available in DEPS:

$$\boldsymbol{pwl}(< varg >,\ < TableName >);$$

with < *varg* > being the name of the argument variable of the piecewise linear function. This operator returns a continuous interval. < *TableName* > is the name of the data table in which the pairs of values characterizing the pieces of line segments of the function will be stored.

It is also possible to enumerate these pairs directly in the operator as follows:

$$\boldsymbol{pwl}(< varg >,\ (varg_1,\ vim_1),\ (varg_2,\ vim_2),\ \ldots,\ (varg_n,\ vim_n));$$

with $varg_i$ the ith value of *varg* and vim_i the ith value of the corresponding image of $varg_i$.

Thus, for the piecewise function of Fig. 27 we will write the following property with the *pwl* operator:

$$y = pwl(x,\ (0,\ 0),\ (10,\ 35),\ (25,\ 25),\ (50,\ 40),\ (65,\ 10));$$

A *pwl* operator can be posted by using a data table (cf Fig. 28) that encapsulate the coordinates of the break points

```
1  Table PwlValues
2  Attributes
3  x : Real;
4  y : Real;
5  Tuples
6  [0, 0],
7  [10, 35] ,
8  [25, 25],
9  [50, 40],
10 [65, 10]
11 End
```

Fig. 28 An example of data table

```
1  Model AlgebraicExemple()
2  Constants
3  a : Real = 4.5 ;
4  b : Integer = 5 ;
5  Variables
6  x : Real in [1, 10] ;
7  y : Real in {-1.5,  1.5, 2.3} ;
8  z : Integer in [1, 100] ;
9  w : Integer in { -2, 1, 10, 200} ;
10 Elements
11 Properties
12 cos(x) + a^2 + b^2 = ln(z)*w^z ;
13 End
```

Fig. 29 Example of an equality constraint between two nonlinear algebraic expressions on mixed domains

as follow:

$$y = pwl(x,\ PwlValues);$$

7.3 Defining properties

Several properties can be expressed in DEPS. A property is a relationship between variables. More precisely, we have in DEPS the usual binary algebraic relations making it possible to establish relations of equality (=), difference (< >) and inequality (< , < = , > , > =) between two algebraic expressions. We have also the affectation relation (: =) for allocating a name to an expression.

In order to illustrate the ability to express complex algebraic properties, we give in Fig. 29 an example of an equality constraint between two nonlinear algebraic expressions on mixed domains.

A declared or named expression (keyword *expr*) points to an algebraic expression and makes it possible to reference it.

Thus, the same algebraic expression can be used at several places in a model through its name without being rewritten.

An expression is declared in the Variables zone of a model in the same way as a variable by prefixing it with the keyword expr.

An *expr* can be initialized (using the assignment operator*:* =) in the Properties area of the model where it is declared or inside the Properties area of another model via a path. Initializing an *expr* means defining the explicit algebraic expression to which it will point and whose value it will take. Each expr is defined once and only once. For a given problem the expr graph must necessarily be a direct acyclic graph (DAG).

In addition to reuse, declaring expressions avoids to artificially increase the number of unknowns of problem. This is the case when these expressions are useful to the system designer only for observation or for renaming purposes (cf examples in Sect. 9).

Thus, in the Serial model in Fig. 13, we have created an *expr U* initialized at the value of the sum of the voltages at the terminals of each dipole of the serial link.

It is also possible to specify in DEPS that the value of a variable is an objective to be minimized by using the *obj* keyword. An objective is an expression whose value defined with the affectation relation (: =) must be minimized. This allows to express optimization problems in DEPS. A problem expressed in DEPS must have at most one *obj* expression.

7.4 Properties defined in extension

It is possible to force a set of variables to take their values in a Table and only in this Table.

To do this we have a *catalog* constraint with the following syntax:

Catalog *([< v1 >, < v2 >, …, < vn >], < TableName >);*

The first argument of this constraint is the list of variables to constrain. The second argument is the name of the constraining table.

Sometimes a data table is reused to put a catalog constraint on a subset of the table's columns. To do this, we have another catalog constraint with the following syntax:

Catalog*([< v1 >, …, < vn >], [< i1 >, …, < in >], < TableName >);*

The first argument of this constraint is the list of variables to constrain. The second argument is the list of the column numbers to be considered in the table. The third argument is the name of the constraining table. This catalog property can work on every kind of domains of variables: continuous, discrete and enumerations.

Figure 30 illustrates the use of such a constraint in the case where we want to create a *VSource* voltage generator model consisting of generators which maximum voltage and current characteristic values are constrained by the *Battery* table.

Figure 31 and Fig. 32 represent a problem in which one wishes to know the unknown resistance of an electrical system *S* represented by two resistors in series connected to a voltage generator *G*. The Power *P* dissipated by the system is constrained to not exceed a value *Pmax*. The goal is to find one value for *R1* and one value for *G* that requires the Power *P* value dissipated between 10 W and Pmax.

```
1  Model VSource() extends Dipole
2  Constants
3  Variables
4  ref : Index;     Imax : Current;
5  Elements
6  Properties
7  Catalog([ref, V, Imax], Battery) ;
8  I <= Imax;
9  End
```

Fig. 30 DEPS model of generator with catalog

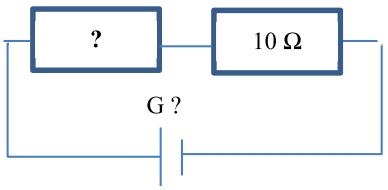

Fig. 31 A very simple electrical system to design

```
1  Problem ElecPb
2  Constants
3  Pmax : Power = 100 ;
4  Variables
5  obj P : Power ;
6  Elements
7  G : VSource();
8  S : SysElec(G);
9  Properties
10 P := S.G.U*S.G.I;
11 P >= 10.
12 P <= Pmax;
13 End
```

Fig. 32 DEPS Problem ElecPb

7.5 Meta properties

The posing of a set of properties can be conditioned in DEPS by the satisfaction of a logical formula. A logical formula is a logical expression tree whose nodes are logical operators (not, and, or, xor) and which leaves carry algebraic properties.

As an example, *((x = y) or (z = cos(y)) and (w + y = z + x)* is logical formula.

The result of the evaluation of a logical formula is done in a trivalued logic. The formula can be certainly true, certainly false or unknown.

As an example:

when $x \in [2, 10]$, $y \in [-1, 4]$,

$(x > 2)$and $(y > 2)$isunknown

but

when $x \in [-1, 1]$, $y \in [-1, 1]$,

$(x > 2)$and$(y > 2)$is certainly false

and

when $x \in [3, 10]$, $y \in [3, 10]$,

$(x > 2)\, and\, (y > 2)\, is certainly true$

These formulas are used in two dedicated meta properties: *IfThen* and *IfThenElse*.

Concerning the *IfThen* property it takes as argument a logical formula and a list of algebraic properties. It is defined using the following syntax:

$$\boldsymbol{IfThen}(< Logical\ Formula >, < List\ Of\ Properties >);$$

If the logical formula is certainly true then the properties of < List of Properties > are active. In the example below, if x is greater than 3 and z is less than 20, then the properties $x\hat{}2 = 16$ and $x + z = 15$ are set.

IfThen$((x > 3)$ and $(z < 20)$, $[x^2 = 16, x + z = 15])$;

Concerning the *IfThenElse* property it takes as arguments a logical formula, a list of algebraic properties to be set if and only if the logical formula is certainly true and a second list of properties to be set if and only if the logical formula is certainly false.

It is defined using the following syntax:

$$\boldsymbol{IfThenElse}(< Logical\ Formula >, < List\ Of\ Properties >, < Other\ List\ Of\ Properties >);$$

If the logical formula < *Logical Formula* > is certainly true, then the properties of < *List of Properties* > are active, and if the logical formula is certainly false, then the properties of < *Other List of Properties* > will be active. In the following example, if x is greater than 3 and z is less than 20, then the properties $x\hat{}2 = 16$ and $x + z = 15$ are set. Otherwise, the properties $x\hat{}2 = 1$ and $x + z = 10$ are set.

IfThenElse$((x > 3)$ and $(z < 20)$, $[x^2 = 16, x + z = 15]$, $[x^2 = 1, x + z = 10])$;

Let us suppose we want to enrich our electrical system model with a diode model. A diode conducts current as a function of the voltage across its terminals. If the voltage at its terminals is lower than a given threshold voltage *Vs,* then the current becomes zero. Otherwise, it follows an exponential law. Figure 33 shows this law using the *If Then Else* metaproperty.

```
1  Model Diode() extends Dipole[]
2  Constants
3  V0 : Voltage = 0.025875;
4  Vs : Voltage = 0.3;
5  I0 : Intensity = 1e-15;
6  Variables
7  Vj : Voltage in [0, 20];
8  Elements
9  Properties
10 IfThenElse( (Vj < Vs),
11         [I = 0],
12         [I = I0*exp((Vj/V0)-1)]);
13 End
```

Fig. 33 Using meta properties to model a diode

8 The DEPS Studio IDE

8.1 Overview

The modeling and resolution process with DEPS is based on a software environment called DEPS Studio [45]..It integrates modeling and solving environment associated with the DEPS language includes a model editor, a project manager, a compiler and a solver (cf Fig. 34). Experience shows that the specification of a system design problem is never right at the first time and that many modeling errors are only detectable by calculation. We therefore decided to develop and integrate our own solver into the development environment so that the solution finding contributes effectively to the problem modeling process. This is a rapid model development approach (analogous to a RAD approach) which, contrary to a model transformation approach, reduces the execution time of the model development loop. It also enables errors to be traced back to the correct level of abstraction. DEPS Studio is currently available as a freeware on request via the contact page of the deps link nonprofit organization website [46]. It can be possible to contact the authors of this paper directly.

8.2 Editor and project management

A problem to be solved is organized into a project. A project consists of several packages. Each package is saved in a file. The packages contain models, kind of quantities, quantities, and tables. One of the packages has to contain a particular model without arguments and declared as the problem. This

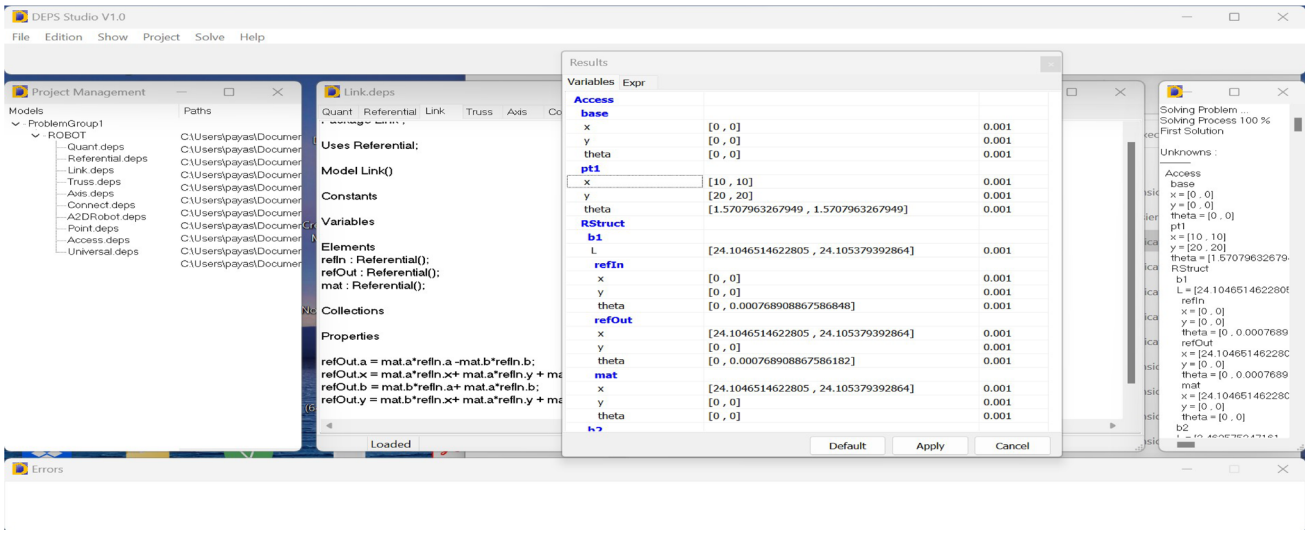

Fig. 34 DEPS Studio IDE

model represents the global problem to be solved expressed with constants, variables, elements and properties.

The environment has:

- A multi-window editor to load, modify and save packages,
- A project manager to load, modify and save the modeling project of a problem made up of all its packages.

A project is defined as a set of packages. Each package follows the following structure:

Package <packageName> ;

Uses <ListOfPackageName> ;

<List of DEPSFeature>

With

<DEPS Feature> ::= <QuantityKind>
| *<Quantity>*
| *<Table>*
| *<Model>*
| *<Problem>*

8.3 The compiler

The compiler we have developed directly transforms the DEPS "source" model of a design problem into a structured network of elements which properties are associated with the < V, D, C > model of a Constraint Satisfaction Problem (CSP) [47]. It is thus a "native" compiler which is not an overlayer of a constraint programming language. Compilation is anticipated; the whole network is thus generated before resolution.

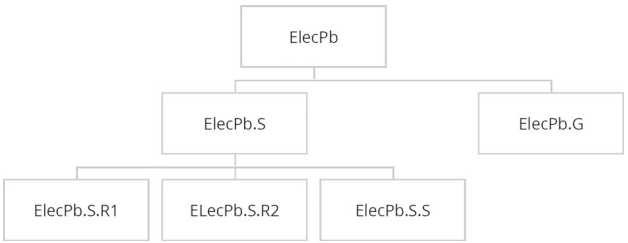

Fig. 35 Example of an instance tree generated by the compiler

The static typing of the DEPS language is exploited by the compiler to detect type errors on constant, variables and elements before resolution.

The compilation is done in two passes:

The first compilation pass checks the packages used by the project, analyzes lexically and syntactically their contents and creates the hierarchy of the project models;

The second pass creates the set of elements that define the problem starting from the creation of the single instance element of the model declared as problem.

Errors are processed and reported to the user at all stages of compilation: package checking, lexical analysis, syntactic analysis, creation of the model hierarchy and creation of the sub-defined elements.

The compiler produces a structured computational model. This computational model is executed on an execution machine, which is the mixed CP solver developed for this purpose (cf Sect. 8.4).

If the compilation step succeeds, then the resolution step will have the task of assigning values to the variables satisfying all the properties/constraints of the problem.

In the case of the *ElecPb* problem (cf Fig. 32), the compilation generates the instance structure shown in Fig. 35.

Each instance contains its own sets of constants, variables and properties. The structure of the problem is thus preserved until its resolution.

8.4 The solver

Solving a design problem requires the ability to take into account:

- Under-constrained problems
- Nonlinear algebraic equations and inequalities on mixed domains
- Other types of relationships such as value tables.

To do so, we have developed a constraint-based solver dedicated to computation on structured DEPS models. The calculation methods we use come from CSP resolution technics [47]. The structure of the DEPS models is preserved throughout the compilation chain up to the calculation models.

The solver implements a revised HC4 propagation method [48] on equations and inequalities. Initially intended for continuous domains, we have extended the method to four types of domains: open real intervals, integer intervals, enumerated sets of floating values and enumerated sets of signed integer values. For performance reasons, reductions are performed directly on the typed domains without going back to the real intervals. The root-finding algorithm uses a branch and prune method. For the moment, only the classical round-robin and first-fail strategies are implemented. In the case of an over-constrained problem, a failure may occur at the first propagation or after exploring the remaining parts of the search tree. Following the CSP paradigm, the failure is interpreted as proof that there is no solution to the problem and not as a failure of the solving algorithm. The object-oriented architecture of the solver has been designed in such a way that it can be extended to other existing propagation and/or resolution methods (box-consistency, local methods, …).

The tree in Fig. 34 is solved instantly by the solver and gives the results in Table 1.

Table 1 Results of DEPS Studio for the ElecPB problem

ElecPb	
P	[7.2, 7.20659340659341]
G	
U	{12}
I	[0.6, 0.600549450549451]
ref	{1}
Imax	{300}
S	
R1	
U	{6}
I	[0.6, 0.600549450549451]
R	[9.99085086916743, 9.99142268984446]
R2	
U	{6}
I	[0.6, 0.600549450549451]

9 Validation

DEPS and DEPS Studio have recently been used successfully on several academic and industrial case studies. In particular, they have been used to express and solve the following problems:

- Deployment of avionics functions on embedded hardware architectures with capacity, safety and reliability requirements in aeronautics [49].
- Software architecture synthesis of embedded electrical generation and distribution system under safety constraints. [50].
- Sizing and architecture generation of battery-type electrical storage system with mission profile requirements [51, 52].
- Configuration and positioning of an on board camera in a UAV [17].
- Sizing of a power transmission system with environmental requirements [53].
- In terms of the typology of problems described in Sect. 2.1, the first two problems are a mix of PT1 and PT3 problems. The last two problems are a mix of PT1, PT2 and PT4 problems.

All these case studies have been solved on a "basic" machine, with the following specification: Gen Intel(R) Core(TM) i7 @ 2.80 GHz, 4 cores. After compiling the models in DEPS Studio, we obtained the following resolution times: 0.297 s for the first case, 0.15 s for the second one, 0.1 s for the third one, 0.19 s for the next one and 0.01 s for the last one.

In order to illustrate the expressiveness and reusability of DEPS language, we present in this paper two other case studies that will be fully described:

- The problem of sizing a planar RR manipulator.
- The problem of deploying software tasks on a hardware architecture embedded in a UAV.

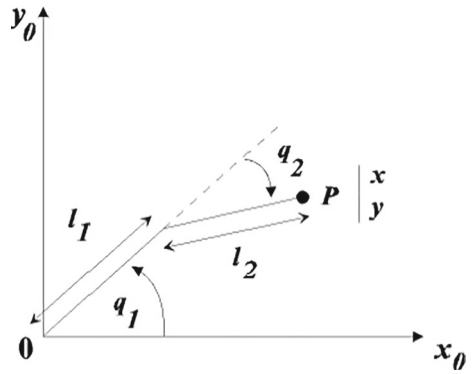

Fig. 36 Geometry of the planar robot

9.1 Sizing of a planar RR manipulator

9.1.1 Problem description

Planar robots are a class of robots whose role is to access points located on a plane. The robot itself is planar, i.e., its effector (the end of the robot) necessarily has a planar trajectory, generally in a plane parallel or coincident with the plane in which the points to be reached are located. These robots can be used to move objects on a plane or to implant electronic components on a board.

Several architectures of planar robots exist [54]. We will limit ourselves here to a simple chain planar robot with two degrees of freedom.

In this paragraph, we present the example of the sizing of a planar RR manipulator (Rotoïde, Rotoïde) (cf Fig. 36). In terms of the typology of problems described in Sect. 2.1, this is a PT1 problem.

A planar RR manipulator consists of a rotoid link parameterized by its rotation angle q_1 followed by an arm of length l_1 and then a rotoid link parameterized by its rotation angle q_2 followed by an arm of lengthl_2.

The problem is to size the RR manipulator to reach one or more points in the plane.

To design this manipulator implemented in (0, 0) in the plane, it is necessary to determine the values of the design variables l_1 and l_2 such that there is a value of the couple of operating variables (q_1, q_2) allowing to reach a reference frame P given in the plane by the coordinates of its origin (x, y) and an angle q of orientation of its x-axis. The notation used is that of Denavit–Hartenberg limited to the dimension of the plane [55].

If we want to reach n points $P_1, P_2, \ldots, P_n$ in the plane with our RR planar robot, we will have to determine a couple of values l_1, l_2 such that whatever the point to be reached among the n, this point is reachable.

$$\exists (l1, l2) \in \mathbb{R}^{+*},\ \forall i \in \{1, .., n\},\ \exists (q_{1i}, q_{2i}) \in [-\pi, \pi],\ \text{IsReached}(P_i, q_{1i}, q_{2i})$$

The referential to be reached by the end of our planar robot (x and y are the coordinates of the origin of the referential and q is the orientation) will therefore be represented by the given 3 × 3 homogeneous matrix P such that:and when P is reached by the robot we have the equality:

$$P = \begin{pmatrix} \cos q & -\sin q & x \\ \sin q & \cos q & y \\ 0 & 0 & 1 \end{pmatrix}$$

$$P = \text{Rot}(q_1) \times \text{Trans}(l_1) \times \text{Rot}(q_2) \times \text{Trans}(l_2) \quad (1)$$

With the rotation matrix $\text{Rot}(q_i)$ such that:

$$\text{Rot}(q_i) = \begin{pmatrix} \cos q_i & -\sin q_i & 0 \\ \sin q_i & \cos q_i & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

And with the translation matrix $\text{Trans}(l_i)$ such that:

$$\text{Trans}(l_i) = \begin{pmatrix} 1 & 0 & l_i \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The second member of Eq. (1) is called the direct geometric model of the RR manipulator.

This case study is academic. The problem presents formalization difficulties that will allow us to illustrate the interest of the DEPS language. From the point of view of resolution, it is an inverse problem.

9.1.2 DEPS modeling

We start by creating a Referential Model (cf Fig. 37). A referential is characterized by an origin (x, y) in the plane, and an orientation angle (theta) relative to an absolute referential located at (0,0) with the unit vectors (1,0) and (0,1).

Thus, the model of a link (cf. Figure 38) will include an input referential (refIn), an output referential (refOut) and a passage matrix (mat), itself a referential, such that (cf Fig. 39):

This last relationship is developed in the properties part of the Link model. Link is an abstract model.

To create the model of a truss of our 2D robot (cf. Figure 40), we will define an extended model of the linkage model which will contain by construction all that a linkage contains with in addition the specific characteristics of a truss, i.e., in our case its length L, design variable of our robot.

```
1  Model Referential ()
2  Constants
3  Variables
4  x : Real;
5  y : Real;
6  theta : Angle;
7  expr a : Real in [-1, 1];
8  expr b : Real in [-1, 1];
9  Elements
10 Properties
11 a := cos(theta);
12 b := sin(theta);
13 End
```

Fig. 37 DEPS model of a 2D referential

```
1  Model Link() abstract
2  Constants
3  Variables
4  Elements
5  refIn : Referential();
6  refOut : Referential();
7  mat : Referential();
8  Properties
9  (* refOut = mat * RefIn  *)
10 refOut.a = mat.a*refIn.a - mat.b*refIn.b;
11 refOut.b = mat.b*refIn.a+mat.a*refIn.b;
12 refOut.x = mat.a*refIn.x+mat.a*refIn.y + mat.x;
13 refOut.y = mat.b*refIn.x+mat.a*refIn.y + mat.y;
14 End
```

Fig. 38 DEPS model of a 2D link

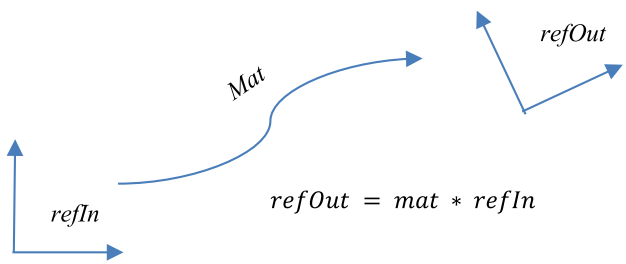

Fig. 39 Mathematical modeling of a link

```
1  Model Truss(Lmax)  extends Link[]
2  Constants
3  Lmax : Real;
4  Variables
5  L : Real in [0, Lmax];
6  Elements
7  Properties
8  mat.y =0;   mat.x = L;   mat.theta = 0;
9  End
10
11 Model Axis()   extends Link[]
12 Constants
13 Variables
14 q : Angle;
15 Elements
16 Properties
17 mat.y = 0;    mat.x = 0;    mat.theta = q;
18 End
```

Fig. 40 DEPS models of 2D Truss and Axis

```
1  Model Connect(L1, L2)
2  Constants
3  Variables
4  Elements
5  L1 : Link[];      L2 : Link[];
6  Properties
7  L1.refOut.x = L2.refIn.x;
8  L1.refOut.y = L2.refIn.y;
9  L1.refOut.theta = L2.refIn.theta;
10 End
11
12 Model Connect(L, P)
13 Constants
14 Variables
15 Elements
16 L : Link[];    P : Point [Real, Real, Angle];
17 Properties
18 L1.refOut.x = P.x;  L1.refOut.y = P.y;
19 End
```

Fig. 41 DEPS model of connection between two links

Moreover, our truss being totally defined once the value of the variable L is fixed, the corresponding referential will be blocked in rotation (mat.theta = 0) and along the y axis (mat.y = 0). The translation along the x axis is L (mat.x = L).

In the same way, we can create an axis model (cf Fig. 40) corresponding to a zero translation and a rotation of an angle q.

We then describe a connection model between two links which specifies that the output referential of the first link will be identical to the input referential frame of the second link (cf Fig. 41).

We can now use these different models to build a model of the robot's structure *RobotStruct* (cf Fig. 42) and a model of its behavior *RobotBehav* (cf Fig. 43). Note that the structure model is an argument of the behavior model. Thus, the structure of the robot can be shared between all these behaviors.

Thus, a 2D robot structure (cf. Figure 42) has a fixed known base represented by the base argument and a 2D robot

```
1  Model RobotStruct(base)
2  Constants
3  Variables
4  Elements
5  base: Point[Real, Real, Angle];
6  b1 : Truss(1000);  b2 : Truss(1000);
7  Properties
8  End
```

Fig. 42 DEPS model of the RR robot structure

```
1  Model RobotBehav(RStruct, effector)
2  Constants
3  Variables
4  Elements
5  RStruct : RobotStruct[Point[Real, Real, Angle]];
6  effector : Point[Real, Real, Angle];
7  j1 :        Axis();          j2 :  Axis();
8  c1:         Connect(j1, RStruct.b1);
9  c2 :        Connect(RStruct.b1, j2);
10 c3 :        Connect(j2, RStruct.b2);
11 access :    Connect(RStruct.b2, effector);
12 Properties
13 j1.refIn.x := RStruct.base.x;
14 j1.refIn.y := RStruct.base.y;
15 j1.refIn.theta := RStruct.base.theta;
16 End
```

Fig. 43 DEPS model of the RR robot behavior

```
1  Model RobotStruct(base)
2  Constants
3  Variables
4  Elements
5  base: Point[Real, Real, Angle];
6  b1 : Truss(1000);   b2 : Truss(1000);
7  b3 : Truss(1000);     (* new Truss b3 for the RRR robot *)
8  Properties
9  End
```

Fig. 44 DEPS model of the RRR robot structure

behavior has an end carrying referential represented by the effector argument.

The models have been written in such a way to facilitate their reusability and extensibility. Thus, if we now wish to represent an RRR robot with a third rotoid joint associated with a third bar, we will simply add a bar in the *RobotStruct* model and a rotoid and a connection in the *RobotBehav* model as shown in Fig. 44 and Fig. 45 where the adds are indicated in red color.

Now the problem (cf Fig. 46) is to look for the robots positioned in (0,0) whose design variables *RStruct.b1.L* and

```
1  Model RobotBehav(RStruct, effector)
2  Constants
3  Variables
4  Elements
5  RStruct : RobotStruct[Point[Real, Real, Angle]];
6  effector : Point[Real, Real, Angle];
7  j1 :        Axis();          j2 :  Axis();
8  j3 :        Axis();  (* new joint for the RRR Robot *)
9  c1:         Connect(j1, RStruct.b1);
10 c2 :        Connect(RStruct.b1, j2);
11 c3 :        Connect(j2, RStruct.b2);
12 (* c4 and c5 new connection for the RRR Robot  *)
13 c4 :        Connect(Rstruct.b2, j3);
14 c5 :        Connect(j3, RStruct.b3);
15 access :    Connect(RStruct.b3, effector);
16 Properties
17 j1.refIn.x := RStruct.base.x;
18 j1.refIn.y := RStruct.base.y;
19 j1.refIn.theta := RStruct.base.theta;
20 End
```

Fig. 45 DEPS model of the RRR robot behavior

```
1  Problem Access
2  Constants
3  Variables
4  Elements
5  base: Point(0,0,0);
6  pt1 : Point(10, 20, Pi/2);
7  pt2 : Point(-10, 20, -Pi/2);
8  RStruct : RobotStruct(base);
9  RBehav1: RobotBehav(RStruct, pt1);
10 RBehav2 : RobotBehav(RStruct, pt2);
11 Properties
12 End
```

Fig. 46 DEPS model of the RR robot sizing problem

RStruct.b2.L are such that there exists a configuration of the axes *RBeehav1.j1* and *RBehav.j2* (i.e., that the angles *RBehav1.j1.q* and *RBehav1.j2.q* exist) in order to reach the frame of reference *pt1* located in (10, 20, π/2). If we want to find a robot that reaches two points pt_1 and *pt2*, we just need to add an instance of the *RobotBehav* model *RBehav2* in the elements of the problem (cf Fig. 45).

9.1.3 Solving results

After having compiled this problem in the DEPS Studio environment, we launch the solver. It immediately tells us by constraint propagation that there is no solution with a single robot to reach the two points pt1 and pt2. We relax the constraint to reach point pt2 in the model and restart the solver and obtain a first solution for the bar lengths b1 and

Table 2 First robot structure solution generated by DEPS Studio

Access	
RStruct	
b1	
L	[24.1046514622805, 24.105379392864]
b2	
L	[3.462575347161, 3.46338845332416]

b2 shown in Table 2 in 0.63 s on a Gen Intel(R) Core(TM) i7 @ 2.80 GHz, 4cores. This illustrates the interest of having an integrated modeling and solving environment for the system design activity.

9.2 Software task deployment on heterogeneous embedded drone architecture

9.2.1 Problem description

The second problem is described in [1, 56] and it deals with Unmanned Aerial Vehicle (UAV) also called drone. The degree of autonomy of these systems evolves progressively from the tracking of a trajectory defined by an operator to a complete autonomy. In the case of trajectory tracking, a key functionality of the UAV is its ability to detect obstacles and plan a new trajectory that will allow it to avoid the UAV while continuing its path to the desired destination. This function composed of obstacle detection, decision making and calculation of a new trajectory must be ensured by the drone alone, without intervention from the ground station, in order to gain in efficiency. It must also be optimized to adapt to the drone's energy resources. The obstacle detection and avoidance functionality is provided by a platform (HW) using a heterogeneous multi-core architecture (SHMC processor), which allows the UAV to have a high computing power while reducing its energy consumption (compared to a single-core or homogeneous multi-core processor). The SHMC processor is composed of cores with different computing power but using the same instruction set. A core can thus be allocated to any software task, without the need for recompilation. The cores can also be shut down when they are not allocated to any task, without the need to be kept in a sleep mode. On this type of processor we generally find a set of more powerful cores (Big Core) and a set of less powerful cores (Little Core). The platform used for the case study is an Exynos 5422 processor, composed of four Cortex A15 cores called "BigCore" and four Cortex A7 cores called "LittleCore". The maximum power dissipation is 4 Watt for a Big Core and 1 Watt for a Little Core.

Table 3 Requirement 2

Task	Core number
T0	[|1, 8|]
T1	{1,5}
T2	{2,6}
T3	[|1, 8|]
T4	[|1, 8|]
T5	[|1, 8|]
T6	{3,7}
T7	[|1, 8|]
T8	[|1, 8|]
T9	[|1, 8|]

The problem consists of deploying ten tasks (image acquisition, filtering, obstacle detection, …) on a maximum of 8 processors (cores) by satisfying a set of constraints. In terms of the typology of problems described in Sect. 2.1, this is a PT3 problem.

A task is characterized by:

- A memory size required for its execution (*MemorySize*).
- A period (*Period*).
- Two WCET (Worst Case Execution Time) depending on whether the task is performed by a Big Core (*WCETBC*) or a Little Core (*WCETLC*).

As an example the first Task T0 called "GetFrame" will capture camera images and place them in memory in a FIFO queue. The second T1 called "ShowPicture" distributes these images so that they can be filtered by the two filtering tasks. Task T2 and T3 called "Filtering 1" and "Filtering 2" will filter the queued images, in order to extract the edges of the objects These two tasks have a high WCET, but can be allocated to two different cores for parallel execution, with a semaphore to prevent concurrent access to the same image.

A core is characterized by:

- An index (*Index*) By convention the indexes from 1 to 4 refer to the BigCore and the indexes from 5 to 8 refer to the LittleCore,
- A maximum amount of available RAM (*Memory*).

One core must be allocated to each of these ten tasks, meeting the following requirements (constraints):

- Req 1: A task T_i is deployed on one and only one core.
- Req 2: Some tasks can only be performed by certain cores according to Table 3.
- Req 3 : Some tasks must be performed by the same core or by two different cores. This type of constraint is noted

On*SameCore* or On*DifferentCore*. These constraints group together tasks sharing a large amount of data (images stored after acquisition) and separate other tasks for parallelism or reliability (in case of failure of a core). Thus, Tasks T1 and T3 as well as Task T2 and T4 must necessarily be deployed on the same core. Similarly, Task T0 and T1 must necessarily be deployed on different cores.

- Req 4: The amount of RAM memory available in each core to execute the tasks assigned to it must not be exceeded. Thus, the sum of the memory sizes required to perform the tasks assigned to a core must be less than or equal to the memory size of the considered core.
- Req 5:The utilization rate of each core, which must not exceed 100%. Thus the sum of the quotients of WCET by the period for each task assigned to a core must be less than or equal to 1.

9.2.2 DEPS modeling

First we will model the Cores and the tasks in DEPS. For the Cores (cf Fig. 47), we create a very general Core model. A core is defined by its index (Index) from 1 to 8 and by a memory capacity (Memory) of a maximum of 2000. We then extend the *Core* model into two derived models *BigCore* and *LittleCore*. We consider that the Cores from 1 to 4 are

```
1  Model Core(Index) abstract
2  Constants
3  Index :  Integer in [1,8] ;
4  Memory : MemorySize = 2000; default ;
5  Variables
6  Elements
7  Properties
8  End
9
10 Model BigCore () extends Core [Integer]
11 Constants
12 Index :  Integer in [1,4]; redefine ;
13 Variables
14 Elements
15 Properties
16 End
17
18 Model LittleCore () extends Core [Integer]
19 Constants
20 Index :  Integer in [5,8]; redefine ;
21 Memory : MemorySize = 500; redefine ;
22 Variables
23 Elements
24 Properties
25 End
```

Fig. 47 DEPS models of the different cores

```
1  Model Task (Memory, WCETBC, WCETLC, Period)
2  Constants
3  Memory : MemorySize;      Period : Time;
4  WCETBC : Time;            WCETLC : Time;
5  Variables
6  CoreNumber : Integer in [1,8]; default ;
7  expr bcRate : PositiveReal;
8  expr lcRate : PositiveReal;
9  Elements
10 Properties
11 bcRate := WCETBC / Period ;
12 lcRate := WCETLC  / Period;
13 End
```

Fig. 48 DEPS model of task

```
1  Model TaskType1()
2  extends Task [MemorySize, Time, Time, Time]
3  Constants
4  Variables
5  CoreNumber : Integer in {1,5}; redefine;
6  Elements
7  Properties
8  End
9
10 Model TaskType2()
11 extends Task [MemorySize, Time, Time, Time]
12 Constants
13 Variables
14 CoreNumber : Integer in {2,6}; redefine;
15 Elements
16 Properties
17 End
18
19 Model TaskType3()
20 extends Task [MemorySize, Time, Time, Time]
21 Constants
22 Variables
23 CoreNumber : Integer in {3,7}; redefine;
24 Elements
25 Properties
26 End
```

Fig. 49 DEPS model of the three types of specialized tasks

BigCore and that those from 5 to 8 are LittleCore. Moreover, the LittleCore instances have a memory capacity reduced to 500. The Index and Memory constants will therefore be redefined.

Regarding the representation of the tasks (cf Fig. 48), they are characterized by their memory requirements (*Memory*), their *WCETBC*, *WCETLC* and their *Period*. Finally, each task carries as unknown the Core to which it will be assigned during the resolution of the problem (*CoreNumber*) in accordance with the Req1 requirement.

Some tasks can only be assigned to certain Cores. We have therefore extended the *Task* model into three models

```
Model ConstraintBetweenTasks (T1, T2) abstract
Constants
Variables
Elements
T1 : Task [MemorySize, Time, Time, Time];
T2 : Task [MemorySize, Time, Time, Time];
Properties
End

Model OnDifferentCore()
extends
ConstraintBetweenTasks
[Task[MemorySize,Time,Time,Time],
Task[MemorySize,Time,Time,Time]]
Constants
Variables
Elements
Properties
T1.CoreNumber <> T2.CoreNumber;
End

Model OnSameCore()
extends
ConstraintBetweenTasks
[Task[MemorySize,Time,Time,Time],
Task[MemorySize,Time,Time,Time]]
Constants
Variables
Elements
Properties
T1.CoreNumber = T2.CoreNumber;
End
```

Fig. 50 DEPS models of constraints between two tasks

(cf Fig. 49) *TaskType1*, *TaskType2*, *TaskType3* by redefining for each one the set of possible core indexes in accordance with the Req2. The TaskType1 model specifies that this type of task can only be executed on core 1 or core 5. In the same way, the TaskType2 model specifies that this type of task can only be executed on cores2 or core 6. In order to be able to express the Req3 requirement, we have defined models (cf Fig. 50) allowing the designer to set properties on a pair of tasks: both tasks on the same processor (O*nSameCore*) or on two different processors (*OnDifferentCore*).

A Model has also been created to express Req4 and Req5 requirements (cf Fig. 51).

Finally, the problem is expressed in Fig. 52:

- The different processors are created: *BCore1* to *BCore4* instances of the *BigCore* model and *LCore5* to *LCore8* instances of the *LittleCore* model.
- The different tasks are created: *T1* instance of the *TaskType1* model, *T2* instance of the *TaskType2* model and *T0, T2, T3, T4, T5, T6, T7, T8, T9* instances of the *Task* model.

```
Model MemoryAndUsageCoreCapacity (T0, …, T9, BigCore)
Constants
Variables
expr b0 : Boolean ; …  expr b9 : Boolean;
expr MemoryCapacity : MemorySize;
expr UsageCapacity : Real in [0, 1];
Elements
T0 : Task[MemorySize, Time, Time, Time];
…
T9 : Task[MemorySize, Time, Time, Time];
Core  : Core[Integer];
Properties
b0 := 0^abs(T0.CoreNumber-Core.Index);
…
b9 := 0^abs(T9.CoreNumber-Core.Index);
(* The sum of the memory sizes of the tasks performed by the
core must be less than or equal to the memory size of the core *)
memoryCapacity := b0*T0.Memory + … + b9*T9.Memory;
MemoryCapacity <= Core.Memory;
(* Core usage rate must be less than or equal to 1 *)
UsageCapacity := b0*T0.bcRate + b1*T1+ …+  b9*T9.bcRate;
UsageCapacity <= 1;
End
```

Fig. 51 DEPS model of the usage rate and memory capacity

```
Problem CoreAllocation
Constants
Variables
Elements
(* creation of the 8 cores *)
BCore1 : BigCore(1);          BCore2 : BigCore(2);
BCore3 : BigCore(3);          BCore4 : BigCore(4);
LCore5 : LittleCore5);        LCore6 :  LittleCore(6);
LCore7 : LittleCore(7);       LCore8 :  LittleCore(8);

(* creation of the 10 tasks *)
T0 :      Task (300,10,30,80);
T1 :      TaskType1 (100,20,45,90);
T2 :      TaskType2 (200,30,55,100);
T3 :      Task (100,30,60,100);    T4 : Task (200,20,55,80);
T5 :      Task (200,15,45,70);   T6 : TaskType3 (200,30,45,90);
T7 :      Task (200,35,50,100);   T8 :  Task (300,20,40,90);
T9 :      Task (400,40,60,100);

(* constraints between tasks Req3*)
Ct1 : OnSameCore(T1, T3);  Ct2 : OnSameCore(T2, T4);
Ct3 : OnSameCore(T6, T7);  Ct4 : OnDifferentCore(T0, T1);

(* BigCore usage rate and capacity constraints Req4 Req5*)
Ct5 : MemoryAndUsageCoreCapacity (T0,T1, …, T9, BCore1);
Ct6 : MemoryAndUsageCoreCapacity(T0,T1, … , T9, BCore2);
Ct7 : MemoryAndUsageCoreCapacity(T0, T1, …, T9, BCore3);
Ct8 : MemoryAndUsageCoreCapacity(T0, T1, …, T9, BCore4);

(* LittleCore usage rate and capacity constraints Req4 Req5*)
Ct9 : MemoryAndUsageCoreCapacity (T0, T1, …, T9, LCore5);
Ct10 : MemoryAndUsageCoreCapacity(T0, T1,…,T9, LCore6);
Ct11 : MemoryAndUsageCoreCapacity(T0,T1,…, T9, LCore7);
Ct12 : MemoryAndUsageCoreCapacity(T0,T1,…, T9, LCore8);
Properties
End
```

Fig. 52 DEPS model of the deployment problem

Table 4 First deployment solution generated in DEPS Studio

Allocation	
T0	
CoreNumber	3
T1	
CoreNumber	1
T2	
CoreNumber	2
T3	
CoreNumber	1
T4	
CoreNumber	2
T5	
CoreNumber	2
T6	
CoreNumber	7
T7	
CoreNumber	7
T8	
CoreNumber	2
T9	
CoreNumber	1

- The constraints between tasks are set: *Ct1* to *Ct3* instances of the *Onsamecore* model and *Ct4* instance of the *OnDifferentCore* model.
- Finally, the capacity and usage rate constraints on the processors are set: *Ct5* to *Ct12* as instances of the *MemoryAndUsageCoreCapacity* model.

9.2.3 Solving results

The experiments were carried out on a "basic" machine, with the following specification.: Gen Intel(R) Core(TM) i7 @ 2.80 GHz, 4 cores. After compiling the models in DEPS Studio, a first solution (cf Table 4) is generated in around 2 s (2, 735 s). It is correct by construction with regard to all the requirements expressed.

9.3 Discussion

In this section, we propose to examine the advantages and limitations of our approach in the light of the previous case studies. Again, the point of view we adopt in DEPS is that of representing the design problem with a view to its automatic resolution. Another important point is that we want to offer a modeling language that can be used by an embedded system designer as well as a mechanical or electrical system designer, and not a programming language that can only be used by a computer scientist. The language was built through an inductive research. Starting with real life cases of design problems, we tried to factorize language elements, gradually leading to the current version of the language. This version was then put to the test on other different problems. Either we were able to represent these problems, or we had to evolve the language. This was and still is our way of enriching the language.

From the point of view of the system design cycle as described in [10, 11], DEPS and DEPS Studio are clearly positioned on the so-called architecture or preliminary design stages of the system. DEPS enables a design problem to be represented using coarse-grained models for doing model-based system synthesis.

From the point of view of the systems to be designed, the language can be used to represent structured systems that can be structurally or functionally decomposed. Technological systems (mechanical, electrical, electronic, …), software-intensive systems (embedded systems) or mixed systems such as cyber-physical systems can thus be taken into account in their preliminary design phases. DEPS Studio can generate several solutions for systems that are correct by construction. However, in its current version, DEPS Studio will not synthesize computer code.

From the point of view of the typology of design problems proposed in Sect. 2.1, we have seen that all the case studies discussed or detailed in this paper illustrate the coverage of the typology's 4 problem types. For the moment, however, the language does not offer the ability to manipulate lists of elements and post properties on these lists. This could be an interesting convenience that is being studied for a future version of the language.

DEPS enables the simultaneous consideration of heterogeneous requirements within a single problem, which can be represented in the form of algebraic, conditional or extensional properties on variables that can take their values in mixed domains (continuous intervals, integer intervals and enumerations). If the requirements to be represented require the use of fine-grained models in the form of differential equations or finite-state automata, their verification will be delegated to the step of detailed system design, using other formalisms (AADL or Modelica, for example).

With regard to the DEPSStudio IDE, the integrated approach of modeling + compilation + resolution in a single software package makes it easier to refine the models. Indeed, the computational model generated by the compiler has the same structure as the problem model expressed in DEPS. The solver's results are therefore given directly in the modeling language and not in the language of an external solver.

Compared with existing general purpose problem-modeling languages such as AMPL [23], MiniZinc [24],

OPL [22] or GAMS [25], DEPS offers structuring capabilities, as well as the ability to deal with mix variables and to combine models. These features naturally favor model reusability and extensibility. It allows the designer, to express algebraic, conditional and extensional properties on mixed variables. Compared with state-of-the-art attempts such as ThingLab [26], S-Comma [33, 34], COB [30], DEPS integrates notions that are non-existent in these works, such as parameterized models, model signatures, aggregation, composition, and model overloading. Compared with product line representation languages such as Clafer [19], DEPS natively allows the representation of mixed problems that must satisfy nonlinear algebraic properties.

From the point of view of scalability, DEPS allows the representation of large-scale problems. Models are extensible and reusable, especially due to the notions of signature and inheritance, as well as the passing of elements and constants as model arguments, which is not the case with other state-of-the-art approaches. As far as solving is concerned, the DEPS Studio solver is subject to the same constraints as all Constraint Programming (CP) solvers: a NP hard problem remains a difficult problem to solve. But most of time the first step is to correctly represent the design problem to solve in a way that is intelligible, extensible and reusable for the designer.

10 Conclusion

In this paper, we have presented the first implemented version of DEPS, a language designed to represent and solve system design problems. The targeted systems are physical, software-intensive or mixed systems (embedded, mechatronic, cyber-physics).

With regard to the research gap, DEPS makes it possible to fill the lack of structuring in current flat problem modeling languages by proposing a declarative, textual and structured language based on properties encapsulated in models.

In its current version, DEPS makes it possible to express sizing problems naturally. It also has functionalities enabling the representation of some problems of configuration, allocation and architecture. We are currently working on enriching the language so that it will express other types of design problems in later versions. We focus on the building of collections of elements and the expression of properties on them. We are also working on the implementation of specialized constraints. The long-term goal is to address complex architecture synthesis problems combining many functional (performance,…) and non-functional requirement (safety, security..)

The results will be integrated in the next version of DEPS Studio.

Moreover, DEPS and DEPS Studio have recently been used to describe and solve various academic and industrial application cases [17, 49–53].

References

1. Leserf, P., de Saqui-Sannes, P., Hugues, J.: Trade-off analysis for SysML models using decision points and CSPs. Softw. Syst. Model. **18**(6), 3265–3281 (2019)
2. Object Management Group (OMG), Unified Modeling Language, Version 2.5.1, formal/17–12–05 (https://www.omg.org/spec/UML/)
3. Object Management Group (OMG). OMG Systems Modeling Language (OMG SysML), Version 1.6. OMG Document Number formal/19–11–011 (https://www.omg.org/spec/SysML/), (2019)
4. Society of Automotive Engineers. SAE Standards: Architecture Analysis & Design Language (AADL), AS5506d, April 2022. (https://www.sae.org/standards/content/as5506d/), (2022)
5. Modelica Association. Modelica: *A unified object-oriented language for systems modelling—Language specifications*. March 2023. https://specification.modelica.org/maint/3.6/MLS.html, (2023)
6. Shah, A.A., Paredis, C.J.J., Burkhart, R., Schaefer, D.: Combining mathematical programming and SysML for automated component sizing of hydraulic systems. J. Comput. Inform. Sci. Eng. **1**(44113), 1231–1245 (2012)
7. Parasolver. Artisan Studio Para Solver^{TM} 7.2 R1 Tutorials. www.InterCax.com. (2013)
8. Creff, S., Le Noir, J., Lenormand, E., & Madelénat, S.: Towards Facilities for Modeling and Synthesis of Architectures for Resource Allocation Problem in Systems Engineering. Proc of 24th Systems and Software Product Line Conference. Montreal. (2020)
9. OCL. OCL 2.4. https://www.omg.org/spec/OCL/2.4/PDF. (2014)
10. IEEE Standard for Application and Management of the Systems Engineering Process, IEEE Std 1220–2005, pp c1–66, 2007. https://standards.ieee.org/standard/1220-2005.html (2007)
11. Technical Committee ISO/IECJTC1/SC7. Iso/iec/ieee 42020:2019—software, systems and enterprise—architecture processes. ISO/IEC/IEEE 42020:2019, pp. 110,07 2019. (2019)
12. INCOSE, Systems Engeneering vision 2035, online (https://violin-strawberry-9kms.squarespace.com/) (2023)
13. SysML V2, 2017, https://www.omgsysml.org/SysML-2.htm (2017)
14. Zeigler, B., Kim, T., Praehofer, H.: Theory of modeling and simulation, Academic Press, (2000)
15. Abrial, J.: Modeling in Event-B: System and Software Engineering, Cambridge Press, (2010)
16. Batteux, M., Prosvirnova, T., Rauzy, A.: System Structure Modeling Language (S2ML) (2015).URL https://hal.science/hal-01234903/document (2015)
17. Yvars, P.A., Zimmer, L.: Towards a correct by construction design of complex systems: the MBSS approach. Proced. Proced. CIRP **109C**, 269–274 (2022)
18. Zimmer, L., and Zablit, P.: Global aircraft predesign based on constraint propagation and interval analysis. CEAS Conference on Multidisciplinary Aircraft Design and Optimization, Köln, Allemagne. (2001)
19. Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wąsowski, A.: Clafer: Unifying class and feature modeling. Softw. Syst. Model. **15**, 811–845 (2014)
20. Eugene, A., Thao, D., Oded, M., and Romain, T.: Using redundant constraints for refinement. In Ahmed Bouajjani and Wei-Ngan

Chin, editors, Automated Technology for Verification and Analysis, pp. 37–51, Berlin, Heidelberg, (2010). Springer Berlin Heidelberg
21. Famelis, M., Salay, R., Chechik, M.: Partial models: Towards modeling and reasoning with uncertainty. ICSE pp. 573–583 (2012)
22. OPL manual. https://www.ibm.com/docs/en/icos/12.8.0.0?topic=manual-opl-modeling-language
23. Fourer, R., Gay, D.M., & Kerdighan, D.W.: AMPL A language for mathematical programming. Duxbury & Thomson. 2003. https://ampl.github.io/ampl-book.pdf (2003)
24. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S. Duck, G.J. and Tack, G.: MiniZinc: Towards a standard CP modelling language. In C. Bessiere, editor, Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, volume 4741 of LNCS, pages 529–543. Springer, (2007)
25. Rosenthal, R.E.: GAMS a users's guide. GAMS Development Corporation, Washington (2007)
26. Borning, A.: ThingLab—An Object-Oriented System for Building Simulations Using Constraints. 5th International Joint Conference on Artificial Intelligence (IJCAI 1977), Cambridge, MA, USA, vol. 1, pp. 497–498. (1977)
27. Shvetsov, I., Semenov, A., Telerman, V.: Application of subdefinite models in engineering. Artif. Intell. Eng. **11**(1), 15–24 (1997)
28. Bensana, E., and Mulyanto, T.: A generic approach for conceptual design based on object oriented and constraint logic programming. EDA 2000. (2000)
29. Mulyanto, T.: Utilisation des techniques de programmation par contraintes pour la conception d'avions. Thèse de l'Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, France. (2002)
30. Jayaraman, B., Tambay, P.: Modeling engineering structures with constrained objects *PADL 2002*. LNCS **2257**, 28–46 (2002)
31. Tambay, P., and Jayaraman, B. The Cob Programmer's Manual.http:// www.cse.buffalo.edu/tech-reports/2003-01.pdf (2003)
32. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press. ISBN 978–0–262–10114–1. (2006)
33. Soto, R.: Langage et transformation de modèles en programmation par contraintes. Thèse de Doctorat de l'Université de Nantes, France (2009)
34. Soto, R. and Granvilliers, L. s-COMMA User's Manual. http://www.inf.ucv.cl/~rsoto/s-comma/ (2007)
35. Vargas, C., Saucier, A., Yvars, P.A.: Ingénierie d'aide à la conception: un environnement pour la réalisation d'un système d'aide à la conception d'organes mécaniques. Revue Int. de CFAO et d'Infographie **10**(1–2), 113–128 (1995)
36. Sellini, F., and Yvars, P.A.: Modèles objet et représentation déclarative du produit en conception mécanique. Revue L'Objet, Numéro spécial: les représentations par objet en conception, 4(2) (1998)
37. Albarello, N., Welcomme, J.B., and Reyterou, C.: A formal design synthesis and optimization for systems architectures. 9th International Conference of Modeling, Optimization and Simulation (MOSIM'12), Bordeaux, France. (2012)
38. Burgueno, L., Mayerhofer, T., Wimmer, M., Vallecillo, A.: Specifying quantities in software models. Inform. Softw. Technol. **113**, 82–97 (2019)
39. OMG SysMl QUDV. https://www.omgwiki.org/OMGSysML/doku.php?id=sysml-qudv:quantities_units_dimensions_values_qudv
40. QUDT Ontology. https://www.qudt.org/
41. Modelica units. https://doc.modelica.org/Modelica%204.0.0/Resources/helpDymola/Modelica_Units.html
42. Taylor, B.N, and Thomson, A. The International System of Units (SI). NIST, http://www.nist.gov/pml/pubs/sp811/. (2008)
43. International Vocabulary of Metrology—Basic and general concepts and associated terms, 3rd edition, (https://www.bipm.org/documents/20126/2071204/JCGM_200_2012.pdf/f0e1ad45-d337-bbeb-53a6-15fe649d0ff1), (2008)
44. Gibbings, J.C.: Dimensional Analysis, Springer, ISBN 978–1–84996–316–9 (2011)
45. Yvars, P.A., Zimmer, L. Integration of Constraint Programming and Model-Based Approach for System Synthesis, proc of the IEEE International Systems Conference, SYSCON, Vancouver, Canada. (2021)
46. DEPS link nonprofit organization. https://www.depslink.com
47. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press, London and San Diego (1993)
48. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising Hull and Box consistency,16th International Conference on Logic Programming, (1993)
49. Zimmer, L., Yvars, P.A., Lafaye, M.: Models of requirements for avionics architecture synthesis: safety, capacity and security, *Proc of the 11th Complex System Design and Management (CSDM) conference*. France, Paris (2020)
50. Yvars, P.A., Zimmer, L.: Synthesis of software architecture for the control of embedded electrical generation and distribution system for aircraft under safety constraints: The case of simple failures, proc of the 14th International Conference of Industrial Engineering, CIGI-QUALITA 2021, Grenoble, France, (2021)
51. Diampovesa, S., Hubert, A., Yvars, P.A.: Designing physical systems through a model-based synthesis approach. Example of a Li-ion Battery for Electrical Vehicles, Computers In Industry, Vol. 129, (2021)
52. Hubert, A., Forgez, C., Yvars, P.A.: Designing the architecture of electrochemical energy storage systems. A model-based system synthesis approach, Journal of Energy Storage, Vol 54, Elsevier, (2022)
53. Yvars, P.A., Zimmer, L.: A Model-based Synthesis approach to system design correct by construction under environmental impact requirements, Procedia CIRP, Vol 103, Elsevier, (2021)
54. McCloy, D.: Some comparisons of serial-driven and parallel driven manipulators. Robotica **8**(4), 355–362 (1990)
55. Khalil, W., Dombre, E.: Modeling, identification and control of robots. Taylor Francis, New York (2002)
56. Leserf, P.: Optimisation de l'architecture de systèmes embarqués par une approche basée modèle, Phd Thesis, Toulouse University (2017)

Pierre-Alain Yvars received an Engineering and Master Degree from the Ecole Centrale de lille, a Master Degree in computer science and automation and a Ph.D. from the Universtity of Lille, France. He has an French habilitation to lead research form the University of Grenoble, France. After conducting research and development in the PSA Peugeot Citroën Company in robotics, AI, constraint programming and design, he joins the ISAE-Supméca as a full Professor. He is currently focusing on the specification and the development of Model-based System Synthesis (MBSS) methods and tools applied to the design and the certification of complex systems architectures be they real-time, embedded, software-intensive or even cyber-physical. He is co-creator and co-designer of the DEPS Language and the DEPS Studio IDE.

Laurent Zimmer is a graduate of the Faculty of Science and Engineering of Sorbonne University. Computer scientist specialized in Artificial Intelligence; he is currently Senior Research Engineer in the Research and Future Business Directorate of Dassault Aviation. As an expert in Model-Based Reasoning and Problem Solving, he has coordinated or participated to several national or European research projects some of them leading to the design and development of Computer Assisted Engineering software such as model-based diagnosis tools, qualitative simulators and constraint-based solvers. More recently he has investigated Knowledge Representation and Reasoning issues raised by the Model-based system engineering (MBSE) approach to systems engineering. He is currently focusing on the specification and the development of Model-based system synthesis (MBSS) methods and tools applied to the design and the verification of complex systems architectures be they real-time, embedded, software-intensive or even cyber-physical. He is co-creator and co-designer of the DEPS Language and the DEPS Studio IDE.

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com