Design Science

Bridging the gap between requirements engineering and systems architecting: the Elephant Specification Language

Tim Wilschut[1,2], Albert T. Hofkamp[1], Tiemen J. L. Schuijbroek[2], L. F. Pascal Etman[1] and Jacobus E. Rooda[1]

[1]*Department of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*
[2]*Ratio Computer Aided Systems Engineering B.V., Klokgebouw 155, 5617 AB Eindhoven, The Netherlands*

Abstract
This article presents a domain-specific language for writing highly structured multilevel system specifications. The language effectively bridges the gap between requirements engineering and systems architecting by enabling the direct derivation of a dependency graph from the system specifications. The dependency graph allows for the easy manipulation, visualization and analysis of the system architecture, ensuring the consistency among written system specifications and visual system architecture models. The system architecture models provide direct feedback on the completeness of the system specifications. The language and associated tooling has been made publicly available and has been applied in several industrial case studies. In this article, the fundamental concepts and way of working of the language are explained using an illustrative example.

Keywords: Requirements, Specification, System architecture, Language design structure matrix, DSM, Function, Design, behavior, dependency derivation

Received 20 July 2023
Revised 10 July 2024
Accepted 12 July 2024

Corresponding author
T. Wilschut
t.wilschut@ratio-case.nl

Des. Sci., vol. 10, e25
journals.cambridge.org/dsj
DOI: 10.1017/dsj.2024.30

1. Introduction

Pahl & Beitz (2007) define engineering design as *'the process of creating and optimizing solutions in the form of technical systems for problems within the design space spanned by needs, requirements and constraints that are set by material, technological, economic, legal, environmental and human-related considerations'*. The needs, requirements and constraints are usually specified in text documents written in natural language, and are often referred to as system specifications (Hull, Jackson & Dick 2017). Design engineers have to interpret these system specifications and convert them into technical designs. A recent development is the transition of document-driven systems engineering toward model-based systems engineering (Madni & Sievers 2018; De Saqui-Sannes *et al.* 2022; Campo *et al.* 2023; Wilking *et al.* 2024).

An important step in the conversion of needs, requirements and constraints into technical solutions is the design of the system's architecture (Eggert 2005).

First of all, the system's architecture has a significant impact on the system's performance, for example, reliability, availability, maintainability and cost of the system (Jiao, Simpson & Siddique 2007; Lough, Stone & Tumer 2009). Second, insight into the system's architecture is required for effective change management (Giffin *et al.* 2009). That is, requirements often change during the design process, which may cause redesign of one or more components. Such design changes may propagate through the system yielding redesign of other components (Jarratt *et al.* 2011). A model of the system architecture can be used to evaluate the impact of design changes (Clarkson, Simons & Eckert 2004).

It is common practice to relate needs, requirements and constraints to technical aspects of the system (Pahl & Beitz 2007; Hull *et al.* 2017). Therefore, engineers need methods to structure, visualize and analyze the relations between system architecture, needs, requirements and constraints. In many existing methods and tools, systems engineers and systems architects have to manually ensure the consistency among systems specifications and various system architecture models. This results in a rather heavy coordination effort. Transitioning from document-driven systems engineering toward model-based systems engineering methods may provide opportunities to bridge the present gap between requirements engineering and system architecting. Automated document derivation from a system model is presented in the review by Wilking *et al.* (2024) as one of the use case categories in MBSE.

Ulrich (1995) defines system architecture as the mapping of a system's functions to the components within the system, and to the dependencies between those components. In our previous work (Wilschut *et al.* 2018*b*), we have shown that the intended system architecture can be derived from structured function specifications. In this article, a fixed grammar is used to describe goal and transformation functions, referred to as exogenous and endogenous functions by Crilly (2013). We showed that by specifying both types of functions using a structured textual format, one can automatically derive dependencies between components, between functions, between parameters, as well as derive the mapping of functions onto components, parameters onto functions and parameters onto components, and visualize those dependencies. The components used within the function specifications may represent functional components without a notion of solution principle and physical components with a distinct embodiment.

This presents an effective method to generate a dependency structure matrix (DSM) model (Steward 1981) of the system architecture. Eppinger & Browning (2012) refer to the DSM as the 'system architecture matrix', defining system architecture as the structure of a system embodied in its elements and their relationships giving rise to its functions and behaviors.

In Wilschut *et al.* (2018*b*), however, we have limited ourselves to a single level description of the system. The functions are specified at a single granularity level. Design processes are usually hierarchical in nature (Estefan 2007). Different parts of a system are usually described at different levels of granularity in system specifications (Maier, Eckert & Clarkson 2017). As the design process progresses, the level of granularity of system specifications may become finer. That is, as more details of the system become apparent, the grain size at which a system is described becomes smaller (Eppinger *et al.* 2014).

Furthermore, in Wilschut *et al.* (2018*b*), we only consider functions of the components of the system. Nonfunctional aspects, for example, geometry of components, are not described. Therefore, only intended component dependencies that result from intended functionality are derived. Unintended dependencies between components, for example, a spatial dependency between two components that have to fit in a predefined space, are not derived. Knowledge of intended as well as unintended dependencies between system components is essential in engineering design.

In this article, we develop a novel specification language that enables the structured specification of a system's functions, behavior and design in terms of needs, requirements and constraints at various levels of granularity. The language is named the Elephant Specification Language (ESL). A parser and compiler have been developed that check the consistency of ESL specifications and derive dependencies between components, variables, needs, goal specifications, transformation specifications, design specifications, behavior specifications, relations and combinations thereof across all decomposition levels of the specification, following a predefined set of mathematical rules.

Contrary to the popular SysML notation, which extends UML with requirements diagrams (De Saqui-Sannes *et al.* 2022), ESL relies on a dedicated syntax to specify the functional requirements and a mathematical network representation to derive the dependencies between the elements in the requirement specifications. This enables the automated generation of system architecting models from the ESL requirements specifications. To achieve such functionality in SysML would require the rigorous and strict use of specification conventions, which is very difficult to achieve in practice.

This article presents the design of the domain specific language with the various language elements, and the principle rules to derive the various dependencies. For the design of the language, we draw upon the design science literature regarding functional modeling and design structure modeling.

The outline of this article is as follows. Section 2 briefly describes the research method that has been adopted for the research and development of the new language. Section 3 elaborates on the typical elements of a system specification and how these elements are modeled. Additionally, existing methods and tools for the creation of system specifications are briefly discussed. In Section 4, the line of reasoning underpinning the ESL structure, ESL language elements and automated dependency derivations is presented. In Section 5, a small three-level ESL example is presented to illustrate parts of the syntax of ESL and show the derived system architecture models at various levels of granularity. Section 6 describes how ESL can be used during the whole engineering design process. The article is concluded with Section 7.

2. Research method

The presented method and language have been developed following the spiral of applied research as presented by Eckert, Stacey & Clarkson (2003). Method and language development started as part of a 4-year PhD project funded by Rijkswaterstaat (RWS), the executive branch of the Dutch Ministry of Infrastructure. RWS frequently conducts European public tenders procedures for the design and realization of a variety of civil works such as bridges, navigation locks, tunnels and roads.

Systems specifications used within these tender procedures have been reviewed to get an overview of the structure and content of these natural language systems specifications used in real-life projects. Herein, we limited the scope of 'system' to the object that had to be designed, realized and integrated in the environment.

Simultaneously, the literature has been consulted for methods on tools on writing system specifications, modeling architectures and the relation between them. The most import works from the literature are summarized in the following section.

The system specification and literature review were the basis for defining the language primitives and grammar, building upon the work presented in Wilschut *et al.* (2018*b*).

The language syntax and semantics have been developed over the course of 2 years during which many variations to describe parts of real-life system specifications and to automatically derive system architectures have been tested. This converged into a stable set of language primitives, language syntax and semantics, and architecture derivations rules.

In the following years, the language has been tested and improved in various industry domains, such as infrastructure (Wilschut *et al.* 2018*a*), high-tech (Meeusen *et al.* 2019; Kools 2022), maritime (Herremans *et al.* 2022) and fusion engineering and related big-science projects (*Beernaert et al. 2024a*,*b*), as part of various Master and PhD projects, and industry design projects.

3. Related work

Writing 'high-quality' system specifications is essential to the success of a design project (Buede 2009). Inconsistencies and ambiguities in system specifications may cause costly and lengthy design iterations. Much has been written on structuring and managing system specifications by academia and industry alike (see, e.g., Hooks 1994; Grady 2014; Hull *et al.* 2017). In this section, we elaborate on what elements are typically described in system specifications.

3.1. Writing system specifications

In the literature, one finds many design process models such as the waterfall model (Royce 1987), the spiral-model (Boehm 1988), the V-model (Forsberg & Mooz 1991) and the onion model (Childers & Long 1994). All these models have in common that one starts with a high-level description of a system, its functions and variables. Subsequently, one decomposes the system, its functions and variables until one obtains a specification (design) that is sufficiently detailed for manufacturing purposes.

Many scientists advocate the usage of separate decomposition trees for the system's components, functions and variables (see, e.g., Suh 1998; Dym & Brown 2012; Pahl & Beitz 2007) and the subsequent manual mapping of components to functions and variables. Crilly (2013) shows, however, that obtaining such a mapping is far from trivial. That is, components may have multiple functions and functions may be fulfilled by multiple components and involve multiple variables. For that reason, Crilly (2013) introduced new function terminology. Crilly (2013) introduced the terms exogenous and endogenous functions. These terms are similar in meaning to the terms goal function and transformation function of Eisenbart, Blessing & Gericke (2012), which are used by Wilschut *et al.* (2018*b*). Exogenous functions describe the purpose of a component with respect to the components around it, while endogenous functions describe processes internal to the component.

The article by Crilly (2013) shows that functions may propagate through nested systems. That is, a function may state that a power source provides power to a lamp. If one subsequently decomposes the power source into a battery and a holder, one may find out that it is actually the battery that provides power to the lamp. This means, the function 'provide power to the lamp' is not decomposed into subfunctions but simply assigned to a subcomponent of the power source.

A design project starts with capturing the Voice-of-the-Customer in a series of statements usually referred to as needs (Eppinger & Ulrich 2015). Needs describe what is wanted by the customer. Needs are typically qualitative, imprecise and ambiguous due to their linguistic origins (Jiao & Chen 2006).

In literature, the term 'requirement' is often used interchangeably with the term 'need', though several scholars define requirements to be structured and formalized needs (Ericson *et al.* 2009; Cascini, Fantoni & Montagna 2013). This is in line with the work of Jiao & Chen (2006) in which they describe an engineering design process of gathering customer needs and subsequent elicitation, analysis and specification of formalized requirements. What is more, the term 'constraint' is often used interchangeably with the term 'requirement' (Glinz 2007). In the Oxford dictionary (Oxford 2009), a requirement is defined as a thing demanded, whereas a constraint is defined as a limitation or restriction. These definitions are consistent with definitions by Koelsch (2016). Constraints may, for instance, be imposed by the laws of nature, material properties and the environment in which the system is ought to function.

In this study, we consider needs to be informal (qualitative) statements on what is desired; requirements to be formal (quantitative) statements on what is desired and constraints to be formal statements that impose limits on what is desired.

Hull *et al.* (2017) point out the benefits of using consistent language in specifying needs, requirements and constraints. These benefits have been recognized by many authors. For example, Cohen (1990) defined linguistic equivalents for mathematical inequality operators, Hooks (1994) debate the usage of the word 'shall' in requirements and van Vliet (2005) introduced the MoSCoW method (Must-o-Should-Could-o-Won't) to create subtle priority differences in requirements, for example, to distinguish between requirements and preferences. Furthermore, several engineers advocate the usage of boilerplates, that is, a fixed layout in which requirements must be written (see, e.g., Arora *et al.* 2014; Mahmud, Seceleanu & Ljungkrantz 2017).

In the literature, one can find a wide variety of terminology stating different types of needs, requirements and constraints, in the following referred to as requirement and constraint specifications, or specifications for short, for example, stakeholder, system, function, behavior, structure, performance, quality and safety specifications (Koelsch 2016). This motivated the increasing popularity of ontology-driven requirement engineering (Chen *et al.* 2013; Dermeval *et al.* 2015), in which prior to writing a specification, one defines the classes of requirements that may occur within a specification.

In general, specifications define the desired system functions, system design (structure), system behavior and derivative properties thereof such as cost, weight and reliability (Fernandes & Machado 2016; Koelsch 2016). A similar classification of requirements is made in the NASA Systems Engineering handbook (Hirshorn, Voss & Bromley 2017): functional requirements are functions needed to accomplish the objectives, while performance requirements define how well the system needs to perform the functions. The INCOSE systems engineering handbook (INCOSE 2023) distinguishes functional/performance requirements, fitness for use requirements (e.g., safety, security and -ilities), design requirements (that constrain the solution) and environment requirements (e.g., operational, maintenance and transportation). As such, any specification method and tool should support the specification of the aforementioned elements.

In the literature, a variety of definitions for function, design and behavior specifications are found. Eisenbart *et al.* (2012), for example, reviewed 12 different definitions of system functions and partitioned them into two groups. The first group are functions that describe a goal of a system. The second group are functions that describe a transformation of flow, such as electrical energy or information. Following this rationale, in Wilschut *et al.* (2018*b*), we defined a specific grammar for function specifications in both groups to reduce ambiguity, that is, a specific sentence structure for writing goal functions to describe the goal of components with respect to other components in the system and a specific sentence structure for writing transformation functions to describe transformations of flow within components. To reduce ambiguity even further, we made use of the functional basis, developed by Stone & Wood (2000) and Hirtz *et al.* (2002)), to restrict the usage of verb synonyms.

System behavior is often modeled in conjunction with system functions and system states. See, for example, the Function–Behavior–State model (Umeda *et al.* 1996), the Structure–Behavior–Function model (Goel, Rugaber & Vattam 2009; Komoto & Tomiyama 2012) and the requirements engineering book of Hull (Hull *et al.* 2017). These models all have their own definitions of behavior, function and state. In general, however, behavior seems to be defined as everything what a 'thing' shall or can do in response to stimuli, that is, when which flows need to be transferred and transformed in what quantity in response to stimuli. Stimuli are changes in (un)controllable flows (state changes). In practice, behavior specifications are usually of the form: when a certain condition holds, then certain behavior shall (not) be exhibited.

The literature has less consensus on design specifications which are referred to by a variety of terms such as performance, quality and safety specifications (Koelsch 2016). We interpret design specifications as descriptions of the conceptual, embodiment and detailed design of a system or derivative properties thereof such as capacity, reliability, availability and cost of components. The conceptual, embodiment and detailed design denote different stages of maturity of a design (Pahl & Beitz 2007).

Hull *et al.* (2017) note that function and design specifications are often combined in single sentences. For example, the sentence 'The power source must provide power with a reliability of 98%' contains the subclause 'with a reliability of 98%' that is subordinate to the main clause 'The power source must provide power'. In this case, the main clause describes the function, whereas the subclause describes a performance bound on the function.

In this study, we adopt the specification rationale of Crilly (2013) and Wilschut *et al.* (2018*b*). By doing so, we aim to avoid the difficulties of connecting separate component, function and variable decomposition trees and aim to enable functions to propagate through the system decomposition tree.

3.2. Dependencies

In engineering design, systems are usually decomposed into more manageable components (Pahl & Beitz (2007)). It is important to specify and actively manage dependencies between components to ensure that the components eventually will fit and function together (Eppinger *et al.* 2014). That is, it is important to keep track of and obtain a complete overview of the system's structure and architecture.

Component dependencies may relate to a wide variety of engineering disciplines such as mechanical, electrical, thermal and software engineering (Tilstra, Seepersad & Wood 2012), making it impossible for a single person to oversee all dependencies (Sosa, Eppinger & Rowles 2007). Therefore, dependencies between components are often visualized and analyzed using graphical models. A major challenge in this process is to ensure and maintain the consistency between the written specifications and the graphical models as the design process progresses. This motivated the development of graphical specification methods such as SysML (Friedenthal, Moore & Steiner 2014).

Written documents, however, remain the primary means of documentation and communication in engineering design (Tomiyama *et al.* 2013). This may be due to the limited scalability of graphical specification methods (Tosserams *et al.* 2010). As a consequence, in practice, requirements engineering and system architecting are often performed as relatively independent tasks. However, they are clearly highly interdependent as the specifications influence the architecture and vice versa. Therefore, we consider a direct relation between the written specification and the graphical models as a desirable feature in engineering design.

3.3. Methods and tools

As discussed in the beginning of this section, we argue that a system specification method or tool should enable a user to specify function specifications, behavior specifications, design specifications and combinations thereof in terms of needs, requirements and constraints. Moreover, we consider it desirable to have a direct relation between written specifications and graphical models to display dependencies between components, functions, variables and combinations thereof. In other words, one should be able to directly derive the structure and architecture of a system from the specifications. Finally, one should be able to easily detail the granularity level of the specification as the design process progresses and enable functions to propagate through the nested systems.

In the literature, a variety of requirement management tools are found (see De Gea *et al.* 2012 for an extensive overview and classification). As the name suggests, these tools mainly focus on the management of large volumes of requirements; they do not consider the conciseness and preciseness of the written specifications themselves. As a consequence, specifications may still be vague and ambiguous. What is more, in most available tooling, users have to manually create the links between components, functions and variables (and many other types of artifacts that one can create within these tools). This results in a significant administrative workload for system engineers and architects. Typically, no direct relation between the written specification and the dependency structure exists. Hence, the resulting dependencies networks are error prone and difficult to verify, validate and maintain.

To improve the preciseness and conciseness of the written specifications, several authors advocate the use of boilerplates (see, e.g., Arora *et al.* 2014; Hull *et al.* 2017; Mahmud *et al.* 2017). Boilerplates predefine a fixed format in which the specifications should be written; however, they do not constrain the actual content of the specifications. Ghosh *et al.* (2016) and Mustafa, Kadir & Ibrahim (2017) develop tooling to automatically convert natural language specifications such that they are formatted following a set of predefined boilerplates. Manual checking is needed to verify the correctness of these conversions.

Other researchers aim at improving the quality of manually written specifications using controlled natural languages. Kuhn (2014) provides an overview of more than 100 controlled natural languages. Most controlled natural languages are used to simplify and automate the translation of user and service manuals in a variety of languages. Only a few focus on system specifications.

Clark *et al.* (2005), for example, developed a controlled natural language for writing knowledge databases for artificial intelligence systems. They noticed that many engineers have difficulties with writing the logical expressions in mathematical form which are stored in such a database. They developed a language that enables engineers to write natural language rules and automatically convert them into logical expressions. Fuchs, Kaljurand & Kuhn (2008) developed Attempto Constrained English (ACE) as a natural language specification tool. ACE supports automatic conversion to first-order logic. To the best of our knowledge, ACE does not support the structured multilevel description of systems. Mavin *et al.* (2009) developed the Easy Approach to Requirements Syntax, which, as we understand them, are automated boilerplates. That is, the tool identifies keywords such as ‘shall’, ‘if’, ‘where’ and ‘while’, but the text between those keywords is not constrained to a predefined format. Feiler, Delange & Wrage (2016) developed the language ReqSpec, which resembles a programming language and is specifically built for the construction industry to perform spatial analysis and design of buildings. Therefore, ReqSpec does not support the concept of system functions. Similar to ReqSpec, the Ψ-language (Tosserams *et al.* 2010) and the z-language (Woodcock & Davies 1996) are domain-specific languages. The former is used for specifying the structure of multidisciplinary optimization problems, and the latter is used for formally describing computing systems.

De Saqui-Sannes *et al.* (2022) categorize languages, tools and methods for MBSE and propose selection criteria in their recent review paper. They distinguish three categories of languages: nonformal, semiformal (often diagrammatic) and languages with a formal semantics enabling mathematical proofs. They list examples of well-known generic semiformal languages such as Matlab Simulink, Modelica and the Object-Proces-Methology (Dori, Reinhartz-Berger & Sturm 2003). Two informal MBSE domain-specific languages are outlined, being SysML and Arcadia/Capella. Formal languages have a clearly defined syntax and a formalized semantics. The semantics of a formal language builds upon a mathematical paradigm. De Saqui-Sannes *et al.* (2022) distinguish state-transition models (such as timed automata and petri nets) and process algebra and logics. Software tools accompanying such informal and formal languages offer model-checking, simulation, verification, code-generation and test generation capabilities. Finally, De Saqui-Sannes *et al.* (2022) note that the languages and tools should be accompanied with an associated method or standard of use.

4. The Elephant Specification Language

Given the state of the art in system specification methods and tools, we postulate that systems engineers and architects would benefit from a domain-specific language that supports the following features:

1. the specification of function specifications, design specifications, behavior specifications and combinations thereof in terms of needs, requirements and constraints of new and existing systems at multiple levels of granularity;
2. the automated derivation of dependencies between components, function specifications, design specifications, behavior specifications and combinations thereof from the specification themselves;
3. the automated propagation of functions through nested systems.

The ESL is specifically designed to support all three features. Moreover, ESL has been designed from an engineering perspective rather than an information management perspective. It has been designed to blend requirements engineering and system architecting by allowing one to automatically generate systems architecture models from the requirement specifications. Perhaps counterintuitively, ESL therefore deliberately deviates from a few classical systems engineering concepts, such as the usage of separate product, function and requirement breakdown structures and the separate modeling of system architectures at functional, conceptual, technological and physical abstraction levels.

Instead, ESL is built upon a system-centered specification perspective. The decomposition of the system into subsystems and components (product breakdown structure) is taken as the central decomposition tree. Components are viewed as black boxes with inputs and outputs (Erden *et al.* 2008). This may seem to be in contradiction to scholars who advocate 'solution-free' modeling, such as Pahl & Beitz (2007) and Stone & Wood (2000), who often advocate a function- or process-centered specification perspective. However, components in ESL may represent conceptual 'blobs', functional units or physical components. A such, in ESL, a component function specification can be solution-free or with a particular solution in mind. As a result, the decomposition tree may contain a mix of these solution-free and solution-specific (physical embodiment) descriptions of components. The component specifications will change, and the decomposition tree will grow (more decomposition levels) as the design process evolves over time.

Moreover, in practice, 100%-new-to-the-world design rarely happens. Engineers typically try to leverage existing designs and solutions as much as possible to reduce costs, lead times and development risks. As such, some parts of a system might already be fully designed and tested in the field, while others are still in the conceptual design phase. Moreover, Caldwell *et al.* (2012) showed that abstract function models are often difficult to understand withouth the context of the (physical) composition of the system at hand. Hence, we adopted a system-centered specification perspective.

Finally, contrary to popular graphical modeling languages such as SysML, ESL is fully text-based. We intentionally chose a text-based format to allow for easy version management using conventional version control methods and tools, such as GIT and SVN, which have been used in software development for decades.

In the remainder of this section, we introduce in an informal way a selection of key ESL elements. For a full overview of the language and the accompanying tools, one is refered to the ESL user manual (Ratio Innovations B.V. 2020*c*), the ESL reference manual (Ratio Innovations B.V. 2020*b*) and the ESL language enhancement proposals (Ratio Innovations B.V. 2020*a*).

4.1. Components

Many engineers experience difficulties in defining and mapping system, function and variable decompositions to each other (Crilly 2013). Therefore, ESL only allows for the definition of a single decomposition: a system decomposition tree (product breakdown structure) consisting of components. The components represent parts of the system. Such a part may represent a conceptual ‘blob’ (a part we wish not to give a particular name yet) or an off-the-shelf component.

Each component must have a definition and must be instantiated as a subcomponent within a (higher-level) parent component. ESL has the built-in `world` component, which is the root of the decomposition tree. The `world` is not part of the system but represents the environment in which the system operates. The component tree (system decomposition) forms the central structure of an ESL specification. All ESL statements are placed within the component tree. For example, Listing 1 shows a `world` containing a single component `pump-module-pm` which is a `PumpModule`.

Listing 1: Decomposing a component

```
world
  component
    pump-module-pm is a PumpModule

define component PumpModule
  components
    motor-mt is an ElectricMotor
    pump-pm is a Pump

define component ElectricMotor
  empty

define component Pump
  empty
```

Component `pump-module-pm` has two subcomponents, being `motor-mt` and `pump-pm`, which are instantiated within the definition of `PumpModule`. Component `motor-mt` is an `ElectricMotor` and `pump-pm` is a `Pump`, both of which are empty. That is, they have no subcomponents (yet). One could, however, define and instantiate subcomponents within the definitions `ElectricMotor` and the `Pump` to add additional layers to the system decomposition.

The definition and instantiation mechanism allows one to create a system decomposition with an arbitrary number of branches each of which has an arbitrary number of levels. Moreover, it allows for the easy reuse of component definitions.

4.2. Variables and types

In engineering design, it is common practice to describe a design in terms of variables. ESL enables the user to declare variables within components. The variables may represent flows from the interaction basis (Tilstra *et al.* 2012), such as electrical energy and information, or properties (attributes) of components, such as length, weight, cost and reliability.

Each variable must have a type, which needs to be defined by the user. The type of a variable provides additional information about the nature of that variable. For example, whether the variable represents a solid or a liquid material flow and what unit it has. The variable types enable consistency checks as variables are passed along multiple levels of the system decomposition tree. ESL supports the mechanism of actual and formal parameters to pass variables across levels of the decomposition tree, which is common in programming languages.

4.3. Verbs and prepositions

The many synonyms of verbs in natural language may cause ambiguity in system specifications (Deng 2002). Therefore, ESL enforces users to define all verb–preposition combinations that are allowed in ESL function specifications. It is advised to use verbs from the functional basis of Hirtz *et al.* (2002), which all have a distinct definition.

4.4. Needs

Needs are informal statements on what is desired (Jiao & Chen 2006). A need specification within ESL has to start with a reference to an instantiated component or a reference to a declared variable. The component or the variable is the subject of the need. All words that follow the subject are unconstrained, that is, a user may write pure natural language. A need may, for instance, describe desired functionality or that a component shall be compliant with a certain standard or norm, as shown in Listing 2. A need can also be used to express political, social and economic factors that are hard to quantify but need to be considered during design of the system.

Listing 2: Example need specification

```
need
   n-cm-01: control-module-cm shall be IP68 compliant
```

4.5. Function specifications

The goal- and transformation-function sentence structures by Wilschut *et al.* (2018*b*) fit well within the system-centered modeling perspective of ESL as goal and transformation functions are formulated in terms of components and variables. The concepts of goal and transformation functions, as presented by Wilschut *et al.* (2018*b*), are therefore woven into ESL.

Goal-function specifications denote the purpose of components with respect to other components within the system. Goal-function specifications can be extended with one or more subclauses that state additional design specifications that are subordinate to the main clause.

Listing 3 shows, for example, goal requirement g-mt-01 that states that component `motor-mt` shall provide `torque-kp` to component `pump-pm`, such that `nominal-torque-kn` is equal to 100 [N/m].

Listing 3: Example goal-requirement specification

```
goal-requirement
  g-mt-01: motor-mt shall provide torque-kp to pump-pm
  with subclause
  * s1: nominal-torque-kn shall be equal to 100 [ N/m]
```

Transformation specifications describe the internal conversion processes within a component. Listing 4 shows example transformation requirement `t-pm-01` that states that an instantiation of definition `Pump` internally shall convert `torque-kp` into `water-flow-qs`.

Listing 4: Example transformation requirement

```
define component Pump
 parameters
  torque-kp is a MechanicalEnergyFlow
  water-flow-qs is a LiquidMaterialFlow

 transformation-requirement
  t-pm-01: shall convert torque-kp into water-flow-qs
```

4.6. Design specifications

Design specifications denote bounds on the values of variables. These variables might represent flows that are used within goal and transformation specifications or properties of components.

Listing 5 shows example design specification `dr-st-01` that states that `storage-capacity-v` shall be at least 0.5m^3. The variable `storage-capacity-st` is a property of component `storage-tank-st`, which implies that `storage-capacity-v` directly relates to the design of component `storage-tank-st`. Note that one could make multiple instances of component definition `StorageTank`. Therefore, design specification `dr-st-01` is not written within component definition `StorageTank`. Otherwise, all instances of `StorageTank` shall have a storage capacity of at least at least 0.5 m^3.

Listing 5: Example design-requirement specification

```
world
  variable
    storage-capacity-v is a Volume

component
  storage-tank-st is a StorageThank with arguments
    * storage-capacity-v

design-requirement
  dr-st-01: storage-capacity-v shall be at least 0.5[ m^3]


define component StorageTank
  parameter
    storage-capacity-v is a Volume property
```

4.7. Behavior specifications

Static functionality can be described using goal and transformation specifications. Behavior specifications can be used to describe requirements regarding dynamic behavior. That is, behavior specifications can be used to describe when a system should do what. These statements are static. ESL does not simulate these statements.

Listing 6 shows example behavior requirement `b-motor-torque` that defines three cases. Case `motor-on` defines that `torque-kp` shall be at least 100 Nm when `control-signal-ce` is equal to True, that is, the motor shall be on. Case `motor-on` defines that `torque-kp` shall be equal 0 Nm when `control-signal-ce` is equal to False, that is, the motor shall be off. Case fallback defines that when no other case applies, that is, for none of the cases the when clauses evaluate to true, then `torque-kp` shall be equal 0 Nm, that is, the motor shall be off.

Listing 6: Example behavior-requirement specification

```
behavior-requirement
  b-motor-torque:
    case motor-on:
      when
        * c1: control-signal-ce is equal to True[ -]
      then
        * r1: torque-kp shall be at least 100[ Nm]
case motor-off:
  when
    * c1: control-signal-ce is equal to False[ –]
  then
```

```
    * r1: torque-kp shall be equal to 0 [ Nm]
  case fallback:
    when no other case applies
    then
      * r1: torque-kp shall be equal to 0 [ Nm]
```

4.8. Relations

The variables in the design of a system may depend on each other in a variety of ways. For example, the second law of Newton dictates a dependency between force, mass and acceleration; and the weight of a component equals to sum of the weight of its subcomponents. Similarly, the reliability of a component depends on the reliability of its subcomponents.

ESL, therefore, supports the specification of the existence of dependencies between variables by means of relations. The specification of the actual equations that mathematically describe those dependencies is beyond the scope of ESL as there exist many (domain-specific) programming and modeling languages for this purpose. The network of dependencies, however, can be used to automatically construct a multidisciplinary design optimization problem (Beernaert & Etman 2019).

4.9. Comments and tags

System specifications are often annotated with comments. ESL has two kinds of comments: code comments and annotation comments. Code comments are not part of the specification but are used to write down information that is relevant to the engineers who are creating and reading the specification. Annotation comments are part of the system specification and are attached as documentation to components, variables, needs, goal specifications, transformation specifications, design specifications and relations. That is, annotation comments can be used to specify additional arbitrary textual information that is not captured by the ESL elements themselves.

In addition, one can attach special tags to components, variables, needs, goal specifications, transformation specifications, design specifications and relatios using annotations comments. Tagging allows one to categorize annotations comments, for example, to tag a component or specification with the responsible stakeholder(s) or to indicate that status of a specification (e.g., draft or final). As such, comments and tags can be used to enrich ESL specifications.

Listing 7 shows the various forms of comments. Code comments are indicated using #, while annotation comments are indicated using # < . By using @ within an annotation comment, one can attach tags to ESL elements.

Listing 7: ESL code and annotations comments

```
world
  # Code comment.
  variable
    x is a real #< In line annotation comment.

```

```
comment
  x #< Annotation comment block for longer text.
  #< @TagName Tagged annotation comment.
```

5. Illustrative example

Since the appearance of the first version of ESL in 2018, it has been applied to create system specifications in a variety of industries such as infrastructure (Wilschut *et al.* 2018*a*), fusion engineering and big science (*Beernaert et al. 2022*, *2024a*,*b*), the maritime industry (Herremans *et al.* 2022) and high tech (Meeusen *et al.* 2019; Kools 2022).

This article focusses on the concepts and way-of-working of ESL, rather than on the application of ESL in an industrial setting. Hence, the usage of ESL is illustrated using an extended three-level version of the water storage system example used in our preceding paper (Wilschut *et al.* (2018*b*)) to explain the concepts of goal and transformation functions.

Figure 1 shows the four-level hierarchy of the water storage system specification. The `world` is the root of the specifications and contains two components: `water-storage-system-sts` and `water-source-ws`. Component `water-storage-system-sts` is the system of interest and is composed of `power-supply-ps`, `pump-module-pm`, `control-module-cm` and `storage-tank-st`. In turn, `pump-module-pm` is composed of `motor-mt` and `pump-pm`. Component `control-module-cm` is composed of `controller-ct` and `sensor-sp`.

5.1. System specification

The ESL specification of the water storage system is shown in Listing 8 through Listing 12. An ESL specification may be written in a single file or distributed over multiple files.

Typically, a *preamble.esl* file is created that contains all type, verb and relation definitions. Listing 8 shows the *preamble.esl* file of the water storage system example in which nine variable types and eight verb–preposition combinations are defined. Note that in the combination `accumulate yielding`, the word `yielding` is not a preposition but a second finite verb.

Next, a *world.esl* file is created that contains the `world` definition of the ESL specification. Listing 9 shows the `world` definition of the water storage example specification.

Listing 8: Type and verb definitions

```
define type
  MechanicalEnergyFlow is a real with unit Nm
  LiquidMaterialFlow is a real with unit l/s
  ElectricalEnergyFlow is a real with unit W
  ElectricalPotential is a real with unit Wh
  Volume is a real with unit m^3
  ControlSignal is a boolean
```

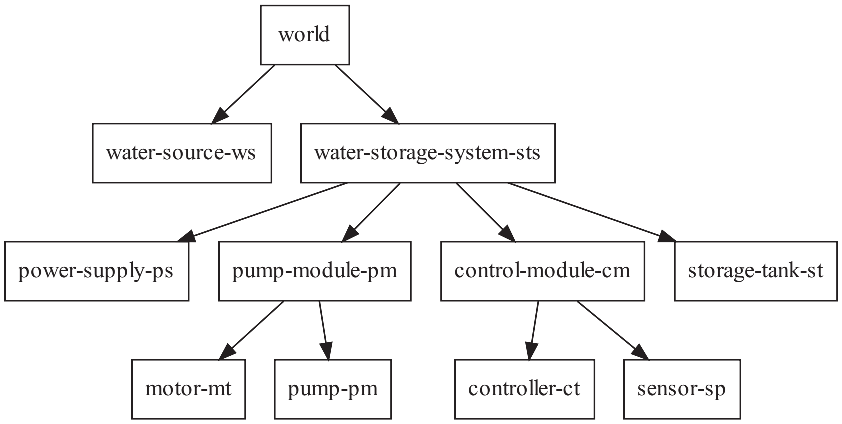

Figure 1. The four-level hierarchy defined within the water storage system specification. The world is the root of the specification, and the `water-strorage-system-sts` is the system of interest.

```
    StatusSignal is a boolean
    DataSignal is a real
    Pressure is a real with unit Bar

  define verb
    provide to
    provide for
    measure within
    send to
    convert into
    pump from
    accumulate yielding
    distribute over
```

Listing 9: World definition

```
world
  variables
    water-source-flow-qs is a LiquidMaterialFlow

  components
    water-source-ws is a WaterSource with arguments
      * water-source-flow-qs

    water-storage-system-sts is a WaterStorageSystem
    with arguments
      * water-source-flow-qs

  goal-requirement
```

```
    g-sts-01: water-storage-system-sts shall pump
    water-source-flow-qs from water-source-ws
```

Within the world definition, the components `water-source-ws` and `water-storage-system` are instantiated. Goal requirement `g-sts-01` states that the purpose of `water-storage-system-sts` is to pump `water-source-flow-qs` from `water-source-ws`. In general, the interactions of the system-of-interest with other systems (components) that exist within its environment are defined within the world. In this case, that interaction is quantified by variable `water-source-flow-qs`, which is defined to be a `Liquid-MaterialFlow` with unit l/s and is passed along as an argument to both components.

In practice, it is often convenient to create separate files for each component definition to allow for easy reuse. Here, the definitions of `WaterSource` and `WaterStorageSystem` are listed in Listing 10. An instance of `WaterSource` shall internally provide `water-source-volume-vs` as a source for `water-source-flow-qs`.

Listing 10: Water storage system level-1 component definitions

```
define component WaterSource
  parameter
    water-source-flow-qs is a LiquidMaterialFlow

  variable
    water-source-volume-vs is a Volume

  transformation-requirement
    t-ws-01: shall provide water-source-volume-vs for
    water-source-flow-qs


define component WaterStorageSystem
  parameter
    water-source-flow-qs is a LiquidMaterialFlow

variables
  pressure-wp is a Pressure
  control-signal-ce is a ControlSignal
  power-pe, power-pc is an ElectricalEnergyFlow
  water-volume-vs, storage-capacity-v is a Volume
  internal-water-flow-qi is a LiquidMaterialFlow

  transformation-requirement
    t-sts-01: shall accumulate water-source-flow-qs
    yielding water-volume-vs

  components
  power-supply-ps is a Battery with arguments
    * power-pe
    * power-pc

  pump-module-pm is a PumpModule with arguments
    * water-source-flow-qs
    * internal-water-flow-qi
    * control-signal-ce
    * power-pe

 control-module-cm is a ControlModule with arguments
    * power-pc
    * control-signal-ce
    * pressure-wp

  storage-tank-st is a StorageTank with arguments
    * internal-water-flow-qi
    * storage-capacity-v
    * pressure-wp
    * water-volume-vs

need
  n-cm-01: control-module-cm shall be IP68 compliant

goal-requirement
  g-ps-01: power-supply-ps shall provide power-pe to
     pump-module-pm
  g-ps-02: power-supply-ps shall provide power-pc to
      control-module-cm
  g-pm-01: pump-module-pm shall provide internal-
     water-flow-qi to…
    storage-tank-st
  g-cm-01: control-module-cm shall measure pressure-
     wp within storage-tank-st
  g-cm-02: control-module-cm shall send control-
     signal-ce to pump-module-pm

design-requirement
  dr-st-01: storage-capacity-v shall be at
     least 0.5[ m^3]
```

An instance of `WaterStorageSystem` shall accumulate `water-source-flow-qs` to yield stored `water-volume-vs`. To fulfill this function, `WaterStorageSystem` is composed of subcomponents `power-supply-ps`, `pump-module-pm`, `control-module-cm` and `storage-tank-st`. These components shall fulfill goal requirements `g-ps-01` through `g-cm-01`, which define the purpose of these components with respect to each other. For example, the purpose of `power-suppy-ps` is to provide `power-pe` to `pump-module-pm`.

Additionally, control-module-cm shall be IP68 compliant as stated by need n-cm-01 and storage-capacity-v shall be at least 0.5m^3. The variable storage-capacity-v is a property of storage-tank-st as can be seen in the definition of StorageTank in Listing 11 (line 70).

Listing 11 contains the definitions of Battery, ControlModule, PumpModule and StorageTank, which are the definitions of the subcomponents of water-strorage-system-sts. In each definition, one can find one or more transformation requirements that define the required internal flow conversions. These transformation requirements denote the desired and thus functional dependencies between the input, output and internal variables of a component.

Transformation-requirements t-bt-01 (line 10) and t-bt-02 (line 11) define, for example, that power-supply-ps, which is an instance of Battery, shall convert internal variable power-potential-pb into the internal flow power-pi and subsequently distribute power-pi over power-pe and power-pc, which are to be provided to pump-module-pm and control-module-cm, respectively, as stated by goal requirements g-ps-01 and g-ps-02 in Listing 10 (lines 52 and 53).

In general, inputs and outputs of a component are always represented by different variables. Note, for example, water-source-flow-qs and internal-water-flow-qi, where the first variable represents the water flow flowing from the water source to the pump module, while the latter represents the internal water flow flowing from the pump module to the storage tank. Hence, when identifying and specifying the goal requirements of a component, it is often helpful to (mentally) create a free-body diagram of the respective component in which all flows crossing the boundary of the component are indicated.

ControlModule contains the subcomponents controller-ct and sensor-st. PumpModule contains the subcomponents motor-mt and pump-pm. The definitions of these components are listed in Listing 12.

Again, each of the definitions in Listing 12 contains a transformation requirement that denotes the desired input–output relations. In addition, one can find behavior requirement b-motor-torque within the definition of ElectricMotor, which defines the behavior dependencies between control-signal-ce and torque-kp.

None of these definitions have subcomponents as we have reached the leafs of the decomposition tree shown in Figure 1. However, if desired, one could easily instantiate additional subcomponents within these definitions to add an additional decomposition layer to the specification.

Listing 11: Water storage system level-2 component definitions

```
1 define component Battery
2   parameters
3     power-pe, power-pc is an ElectricalEnergyFlow
4
5   variables
6     power-potential-pb is an ElectricalPotential
7     power-pi is an ElectricalEnergyFlow
8
9   transformation-requirement
```

```
  t-bt-01: shall convert power-potential-pb into
     power-pi
  t-bt-02: shall distribute power-pi over power-pe
     and power-pc


define component ControlModule
 parameter
  power-pc is an ElectricalEnergyFlow
  control-signal-ce is a ControlSignal
  pressure-wp is a Pressure

 variable
  status-signal-wl is a StatusSignal

 transformation-requirement
  t-cm-01: shall convert power-pc and pressure-wp
    into control-signal-ce

 components
  controller-ct is a Controller with arguments
   * power-pc
   * control-signal-ce
   * status-signal-wl
  sensor-sp is a PressureSensor with arguments
   * status-signal-wl
   * pressure-wp

 goal-requirement
  g-ps-01: sensor-sp shall send status-signal-wl
     to controller-ct


define component PumpModule
 parameters
  water-source-flow-qs, internal-water-flow-qi is a
     LiquidMaterialFlow
  control-signal-ce is a ControlSignal
  power-pe is a ElectricalEnergyFlow

 transformation-requirement
  t-pm-01: shall convert control-signal-ce,
     water-source-flow-qs, and power-pe into
     internal-water-flow-qi

 variables
  torque-kp, nominal-torque-kn is a Mechanical
     EnergyFlow

 components
  motor-mt is an ElectricMotor with arguments
    * control-signal-ce
    * power-pe
    * torque-kp

   pump-pm is a Pump with arguments
    * torque-kp
    * water-source-flow-qs
    * internal-water-flow-qi

 goal-requirements
  g-mt-01: motor-mt shall provide torque-kp to
    pump-pm with subclause
    * s1: nominal-torque-kn shall be at least 100 [ Nm]


define component StorageTank
 parameter
  water-flow-qi is a LiquidMaterialFlow
  storage-capacity-sc is a Volume property
  pressure-wp is a Pressure
  water-volume-vs is a Volume

 transformation-requirement
  t-st-01: shall accumulate water-flow-qi yielding
     water-volume-vs and pressure-wp
```

Listing 12: Water storage system level-3 component definitions

```
define component Controller
 parameters
  power-pc is an ElectricalEnergyFlow
  control-signal-ce is a ControlSignal
  status-signal-wl is a StatusSignal

transformation-requirement
t-cl-01: shall convert status-signal-wl and
   power-pc into control-signal-ce


define component PressureSensor
 parameters
  status-signal-wl is a StatusSignal
 pressure-wp is a Pressure

 transformation-requirement
```

```
    t-ps-01: shall convert pressure-wp into status-signal-wl


define component ElectricMotor
  parameters
    control-signal-ce is a ControlSignal
    power-pe is an ElectricalEnergyFlow
    torque-kp is a MechanicalEnergyFlow

  transformation-requirement
    t-em-01: shall convert control-signal-ce, power-pe into torque-kp

  behavior-requirement
    b-motor-torque:
      case motor-on:
        when
          * c1: control-signal-ce is equal to True [–]
        then
          * r1: torque-kp shall be at least 100 [Nm]
      case motor-off:
        when
          * c1: control-signal-ce is equal to False [–]
        then
          * r1: torque-kp shall be equal to 0 [Nm]
      case fallback:
        when no other case applies
        then
          * r1: torque-kp shall be equal to 0 [Nm]


define component Pump
  parameters
    torque-kp is a MechanicalEnergyFlow
    water-source-flow-qs, internal-water-flow-qi is a LiquidMaterialFlow

  transformation-requirement
    t-pm-01: shall convert torque-kp and water-source-flow-qs into internal-water-flow-qi
```

5.2. Architecture derivation and visualization

A key feature of ESL is the automated derivation of a dependency graph that represents the system architecture. In this section, the working principle of the dependency graph derivation mechanism is informally explained using

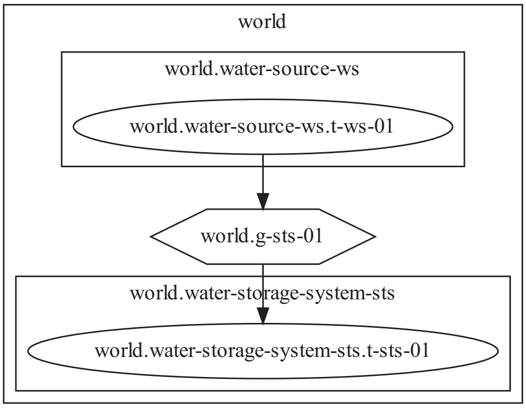
world
world.water-source-ws
world.water-source-ws.t-ws-01
world.g-sts-01
world.water-storage-system-sts
world.water-storage-system-sts.t-sts-01

Figure 2. Functional dependency diagram of the water storage system at decomposition level 1 (rectangle: component; hexagon: goal specification; ellipse: transformation specification).

architecture visualizations of the water storage system example. The system architecture is visualized in graph and matrix form at all three decomposition levels. The interested reader is referend to the reference manual (Ratio Innovations B.V. 2020*b*) for the in-depth explanation of mathematical derivation rules. Goal, transformation, behavior and design specifications are all used to derive dependencies. This section, however, focusses on explaining how functional dependencies are derived from goal and transformation specifications.

Figure 2 shows the generated functional dependency diagram of the water storage system at decomposition level 1. This diagram shows `world` that contains component `water-source-ws`, which contains transformation requirement `t-ws-01` and component `water-storage-system-sts` which contains transformation requirements `t-sts-01`. Goal requirement `g-sts-01` connects transformation requirements `t-ws-01` and `t-sts-01`.

These dependencies are derived by 'following' `water-source-flow-qs` through the system. That is, `water-source-flow-qs` is the output of transformation requirement `t-ws-01`, is transferred from `water-source-ws` to `water-storage-system-sts` as stated by goal-requirement `g-sts-01` and is the input to transformationrequirement `t-sts-01`.

Figure 3 shows a component-function-variable multidomain-matrix (CFV-MDM) of the water storage system at the first decomposition level. The CFV-MDM is composed of three DSMs and three domain mapping matrices (DMMs). The colors of the wedges within the matrix indicate the various labels that have been assigned to a dependency based on the specification.

The component DSM (rows, columns 1 and 2) shows the dependencies, often referred to as interfaces when the dependencies are functionally intended and designed for, between the components of the first decomposition level. A dot at position i,j indicates that component c_i has an interface with component c_j. This matrix is symmetric since if c_i has an interface with component c_j, then by definition c_j has an interface with component c_i. For example, Figure 3 shows that `water-storage-system-st` has a `LiquidMaterialFlow` interface with `water-source-ws`. This interface has been derived from goal requirement `g-`

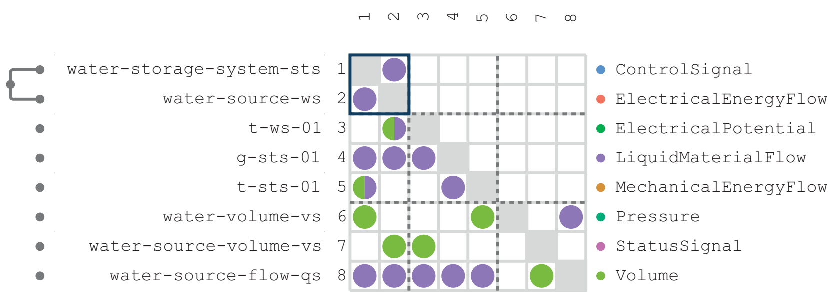

Figure 3. Component-function-variable multidomain-matrix of the water storage system at decomposition level 1.

`sts-01` that states that `water-storage-system-st` shall pump `water-source-flow-qs`, which is a `LiquidMaterialFlow`, from `water-source-ws`.

In practice, one should check the first DSM for missing or unexpected interfaces. A missing interface indicates an incomplete specification. An unexpected interface might indicate a specification error, such as a reference to an incorrect component.

The function DSM (rows, columns 3–5) shows the dependencies between the goal and transformation requirements. A mark at position i,j indicates that goal or transformation requirement r_i requires an input from goal or transformation requirement v_j. Figure 3 shows, for example, that transformation requirement `t-sts-01` requires an input from goal-requirement `g-sts-01`, which in turn requires an input from transformation requirement `t-ws-01`. This path is the same transformation-goal-transformation path as shown in Figure 2. When creating a specification, one should check the completeness of these functional paths. An incomplete path indicates that one or more goal and transformation requirements are missing.

The variable DSM (rows, columns 6–8) shows the dependencies between the set of variables that related to the components and requirements stated within the first decomposition level of the specification. A mark at position i,j indicates that variable v_i depends on variable v_j. In the detailed design phase, one could formulate a mathematical equation or create a model for each dependency within the variable DSM to obtain an optimization model.

The component-function DMM (rows 3–5, columns 1 and 2) shows the mapping from components to goal and transformation requirements. A mark at position i,j indicates that goal or transformation requirement f_i is being fulfilled by component c_j. A transformation requirement always maps to a single component, while a goal requirement always maps to two components. This DMM can, for example, be used during change management to get an indication of which set of goal and transformation requirements might be affect when changing the design of a component.

The component variable DMM (rows 6–8, columns 1 and 2) shows the mapping of components to variables. A mark at position i,j indicates that variable v_i is an input, output or property of component c_j. This matrix provides for each component an overview of all variables that are subjects to requirements.

The function variable DMM (rows 6–8, columns 3–5) shows the mapping of goal and transformation requirements to variables. A mark at position i,j indicates

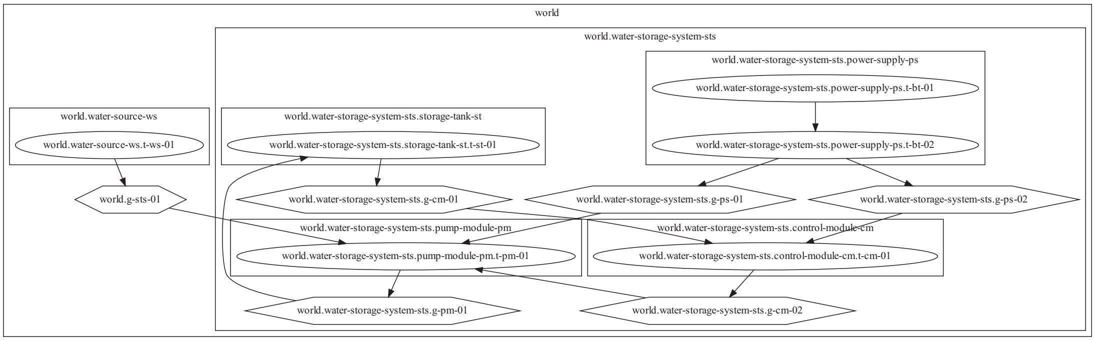
world
world.water-storage-system-sts
world.water-storage-system-sts.power-supply-ps
world.water-storage-system-sts.power-supply-ps.t-bt-01
world.water-storage-system-sts.power-supply-ps.t-bt-02
world.water-source-ws
world.water-source-ws.t-ws-01
world.water-storage-system-sts.storage-tank-st
world.water-storage-system-sts.storage-tank-st.t-st-01
world.g-sts-01
world.water-storage-system-sts.g-cm-01
world.water-storage-system-sts.g-ps-01
world.water-storage-system-sts.g-ps-02
world.water-storage-system-sts.pump-module-pm
world.water-storage-system-sts.pump-module-pm.t-pm-01
world.water-storage-system-sts.control-module-cm
world.water-storage-system-sts.control-module-cm.t-cm-01
world.water-storage-system-sts.g-pm-01
world.water-storage-system-sts.g-cm-02

Figure 4. Functional dependency diagram of the water storage system at decomposition level 2 (rectangle: component; hexagon: goal specification; ellipse: transformation specification).

that variable v_i is part of goal or transformation requirement r_j. This matrix provides an overview of which goal and transformation requirements are affected when changing the value of a variable and vice versa.

In Figure 4, the functional dependency diagram of the water storage system is shown at the second decomposition level. In this diagram, `water-storage-system-sts` is expanded to reveal subcomponents `power-supply-ps`, `control-module-cm`, `pump-module-pm`, `storage-tank-st` and the associated goal and transformation requirements.

Note that transformation requirement `t-sts-01` is not part of Figure 4. It has been substituted by the goal and transformation requirement to be fulfilled by the subcomponents of `water-storage-system-sts`. Goal-requirement `g-sts-01` now provides the input to transformation-requirement `t-pm-01`. The reason for this is that it is `pump-module-pm` that actually pumps `water-flow-qs` from `water-source-ws`.

Figure 5 shows the CFV-MDM at decomposition level 2. The component DSM, goal and transformation requirement DSM and variable DSM have all increased in size due to the expansions of `water-storage-system-sts`.

With rows 1–4 and columns 1–4, one can see all internal interfaces between the subcomponents of `water-storage-system-sts`. These interfaces are directly derived from goal requirements `g-ps-01` through `g-cm-02` listed within Listing 10 (rows 52–57).

Note that `pump-module-pm` has an external interface with `water-source-ws` (row 5, column 1). The reason for this can be seen at position (11, 1) of the MDM, where one can see that goal requirement `g-sts-01` has been assigned to `pump-module-pm`. That is, goal requirement `g-sts-01` has automatically migrated one level down as it is `pump-module-pm` that actually converts `water-source-flow-qs` into `internal-water-flow-qi`. In other words, it is `pump-module-pm` that pumps water. Hence, `pump-module-pm` has to have an interface with `water-source-ws`.

Within the function DSM, one can observe that the goal and transformation chains have significantly increased in length. One can even identify the loop `g-cm-02` → `t-pm-01` → `g-pm-01` → `t-st-01` → `g-cm-01` → `t-cm-01` → `g-cm-02`. This loop represents the pressure control feedback loop. That is, based on the measured pressure within `storage-tank-st`, `control-module-cm`

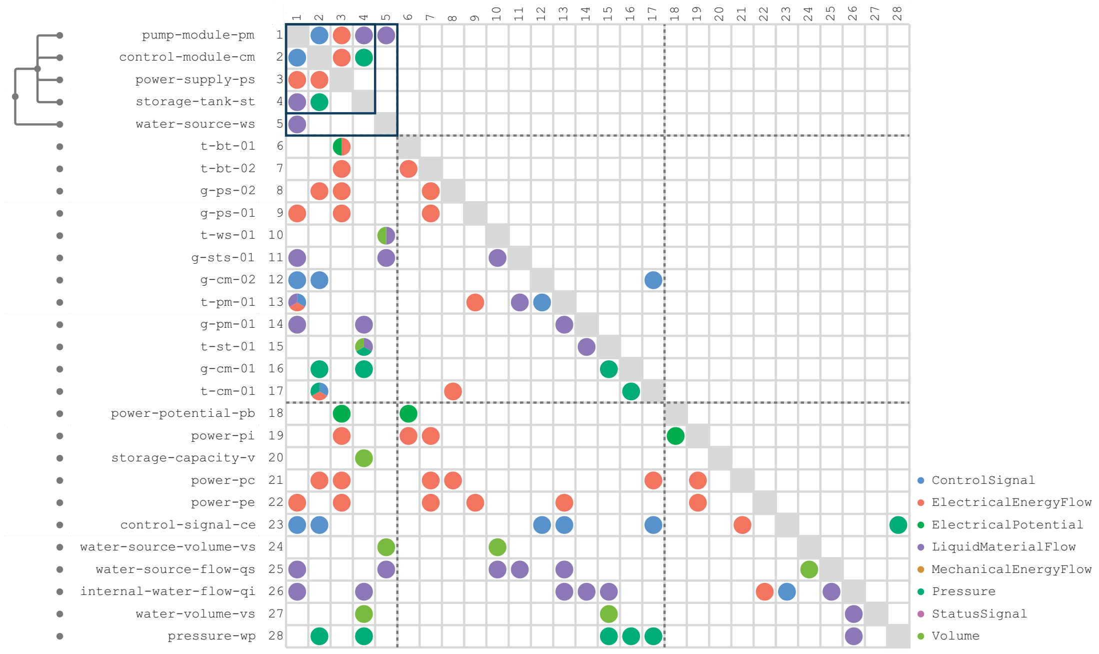

Figure 5. Component-function-variable multidomain-matrix of the water storage system at decomposition level 2.

shall determine an appropriate value for `control-signal-ce`. The behavior requirements that state the logic for determining this value have not been added to this example.

The control feedback loop is also visible in Figure 4, albeit a bit more difficult to identify. Hence, for real-life systems, it is often more convenient to use the matrix visualizations as they provide better scalability.

The variable DSM (rows, columns 18–28) is extended with all variables that are used within goal and transformation requirements at the second decomposition level.

In Figure 6, the functional dependency diagram of the water storage system is shown at the third decomposition level. In this diagram, the components `pump-module-pm` and `control-module-pm` are decomposed one level further the reveal their subcomponents and associated goal- and transformation-requirements.

Goal-requirement `g-sts-01` now provides input `water-source-flow-qs` to transformation requirement `t-pm-01` which is to be fulfilled by third-level component `pump-pm`.

Note that Figure 6 is already becoming relatively crowded, while the decomposition tree of this small example contains only seven leaf components. This is a frequently encountered problem when using graphical modeling languages. Hence, the choice for a textual format of ESL with component by component definition of the elements in the tree.

Figure 7 shows CFV-MDM at decomposition level 3, which provides a more detailed overview of the system architecture. The component DSM (rows, columns 1–7) that `water-source-ws` has an interface with `pump-pm`, that `power-`

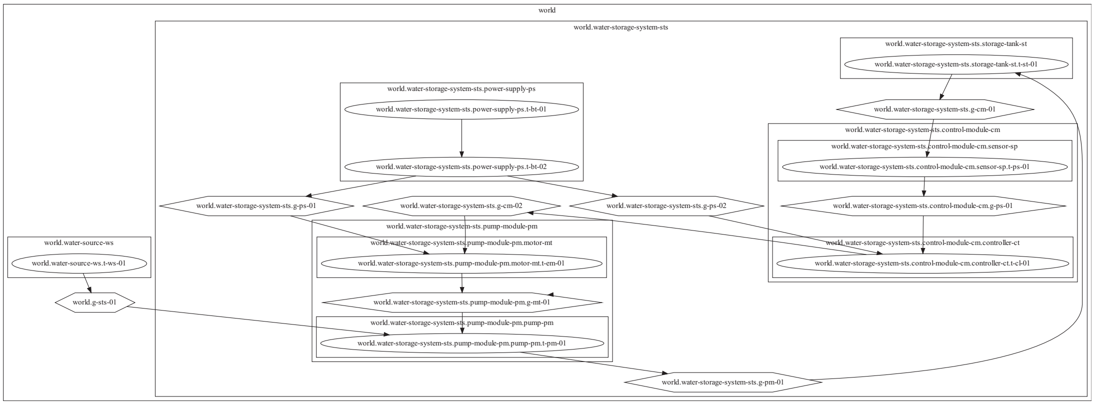
world
world.water-storage-system-sts
world.water-storage-system-sts.storage-tank-st
world.water-storage-system-sts.storage-tank-st.t-st-01
world.water-storage-system-sts.power-supply-ps
world.water-storage-system-sts.power-supply-ps.t-bt-01
world.water-storage-system-sts.power-supply-ps.t-bt-02
world.water-storage-system-sts.g-cm-01
world.water-storage-system-sts.control-module-cm
world.water-storage-system-sts.control-module-cm.sensor-sp
world.water-storage-system-sts.control-module-cm.sensor-sp.t-ps-01
world.water-storage-system-sts.g-ps-01
world.water-storage-system-sts.g-cm-02
world.water-storage-system-sts.g-ps-02
world.water-storage-system-sts.control-module-cm.g-ps-01
world.water-storage-system-sts.pump-module-pm
world.water-source-ws
world.water-source-ws.t-ws-01
world.water-storage-system-sts.pump-module-pm.motor-mt
world.water-storage-system-sts.pump-module-pm.motor-mt.t-em-01
world.water-storage-system-sts.control-module-cm.controller-ct
world.water-storage-system-sts.control-module-cm.controller-ct.t-cl-01
world.g-sts-01
world.water-storage-system-sts.pump-module-pm.g-mt-01
world.water-storage-system-sts.pump-module-pm.pump-pm
world.water-storage-system-sts.pump-module-pm.pump-pm.t-pm-01
world.water-storage-system-sts.g-pm-01

Figure 6. Functional dependency diagram of the water storage system at decomposition level 3 (rectangle: component; hexagon: goal specification; ellipse: transformation specification).

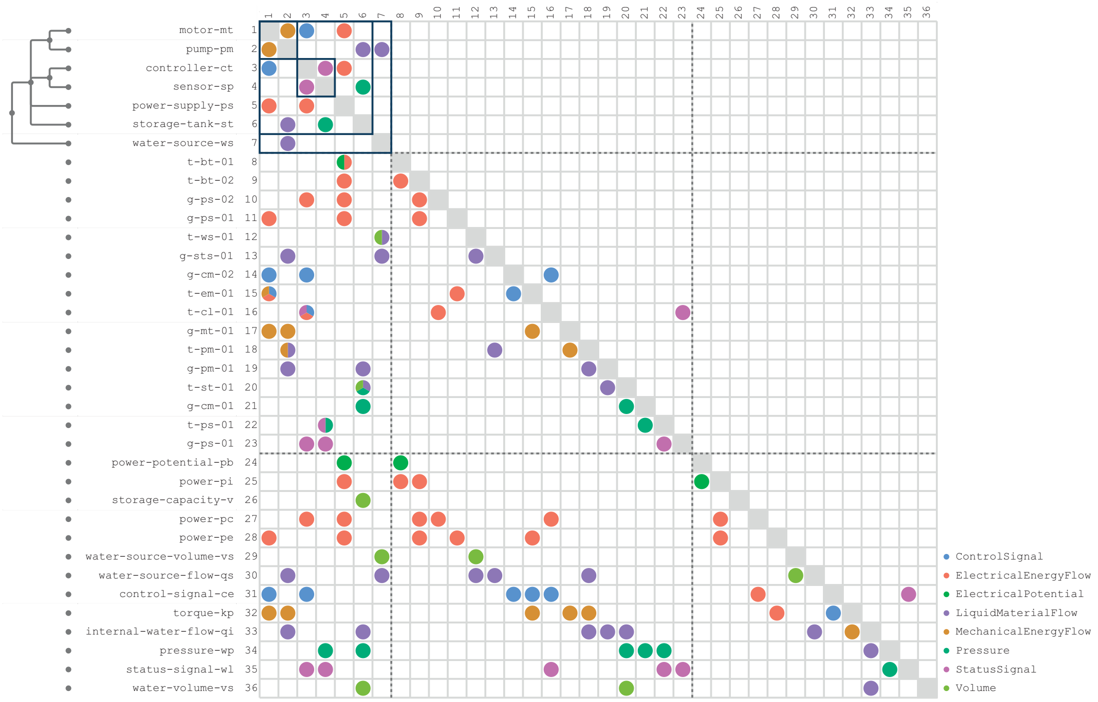

Figure 7. Component-function-variable multi-domain-matrix of the water storage system at decomposition level 3.

supply-ps has interfaces with motor-mt and controller-ct, and that storage-tank-st has interfaces with component pump-pm and sensor-sp. Hence, the component DSM contains components that are defined at three different levels in different branches of the decomposition tree, yet provides a complete and consistent overview of all interfaces between these components. These interfaces correspond directly to the stated goal and transformation requirements.

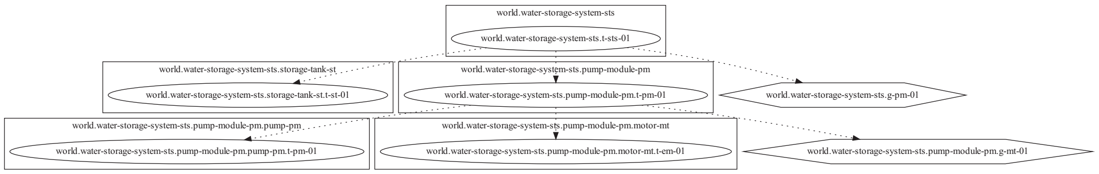
world.water-storage-system-sts
world.water-storage-system-sts.t-sts-01
world.water-storage-system-sts.storage-tank-st
world.water-storage-system-sts.storage-tank-st.t-st-01
world.water-storage-system-sts.pump-module-pm
world.water-storage-system-sts.pump-module-pm.t-pm-01
world.water-storage-system-sts.g-pm-01
world.water-storage-system-sts.pump-module-pm.pump-pm
world.water-storage-system-sts.pump-module-pm.pump-pm.t-pm-01
world.water-storage-system-sts.pump-module-pm.motor-mt
world.water-storage-system-sts.pump-module-pm.motor-mt.t-em-01
world.water-storage-system-sts.pump-module-pm.g-mt-01

Figure 8. Traceability diagram for transformation requirement `t-mp-01` of the water storage system.

Similarly, within the function DSM (rows, columns 8–23) and variable DSM (rows, columns 24–36), one can still find a complete and consistent network of dependencies, while the respective goal requirements, transformation requirements and variables are specified throughout the decomposition tree. Each dependency is directly and automatically derived based on mathematical rules.

In fact, the structured syntax and semantics allows for the automated derivation of traceability dependencies between transformation and goal requirements as well, which is often the primary motivation for creating a separate function decomposition. Figure 8 shows, for example, the traceability diagram for transformation requirement `t-sts-01` which is to be fulfilled by `water-storage-sts`. One decomposition level down, the requirement is actually being fulfilled by the sequence of transformation and goal requirements `t-pm-01`, `g-pm-01`, and `t-st-01`. In turn, transformation requirement `t-pm-01` is one-level down fulfilled by the sequence of goal and transformation requirements `t-em-01`, `g-mt-01` and `t-pm-01`.

6. Usage of the Elephant Specification Language

The earlier mentioned waterfall model (Royce 1987), spiral model (Boehm 1988), systems engineering V-model (Forsberg & Mooz 1991), onion model (Childers & Long 1994) and NASA's systems engineering engine (Hirshorn *et al.* 2017) all have in common that they describe the engineering design and realization process as an iterative process in which engineers model the system at various hierarchical levels during different design phases. However, the terminology in which the process, activities and associated system models are described often differs (Maier *et al.* 2017).

Figure 9 merges the onion model and the NASA SE model into a variant of the V-model. We take the design process as displayed in the left-hand side of Figure 9 as our reference for writing ESL specifications.

One starts at the top left at decomposition level 0 with requirements engineering (R) and system architecting (A) by specifying needs in the form of 'the system shall…', by specifying interactions with other components that exist within the environment the system shall operate in the form of goal requirements and by specifying internal transformations that the system will need to perform in the form of transformation requirements. Desired or required quantitative bounds on properties of the system can be added in the form of subclauses to goal and transformation requirements and design-requirements.

The interactions of the to-be-designed system with its environment, the required internal transformations and design-requirements are likely to depend on its working principle and embodiment. For example, a petrol car has a different

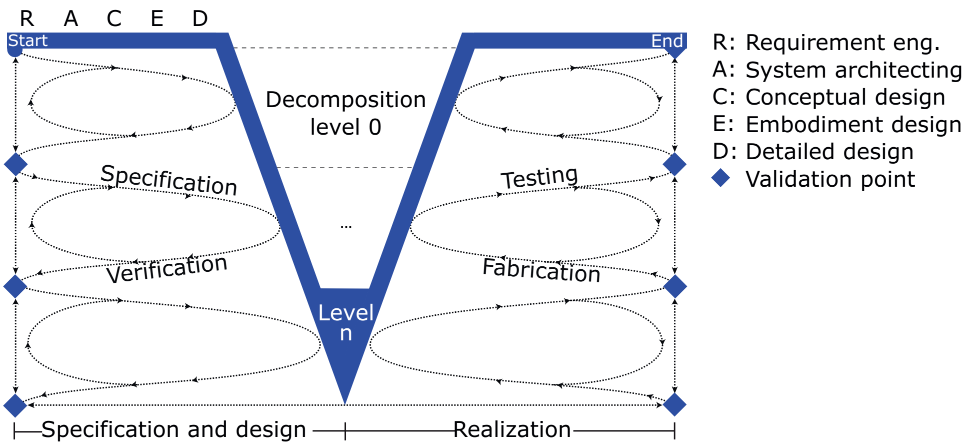

Figure 9. Adapted version of the V-model (due to Forsberg & Mooz 1991), indicating the iterative design processes at the various decomposition levels. Design decisions at decomposition level *i* are input to the design process of the subcomponents at decomposition level $i+1$. The developed ESL language neatly supports the specification and design process in the left-hand side of the V.

internal working principle than an electric car, and a gasoline car may be subject to different legislation than a diesel car. Hence, the conceptual design (C) determines what working-principles are best suited for the system. Subsequently, the embodiment design (E) determines how that working-principle could be realized. The detailed design (D) is needed to verify, or at least get an indication off whether the needs and requirements can be met with the selected working principle and embodiment.

Once a working principle and embodiment at level 0 have passed the verification stage, one can move to the first validation point on the left in Figure 9. If the validation is not successful, the stated requirements and chosen working principles and embodiments are likely to be incorrect or incomplete and one has to go through the R, A, C, E, D (RACED) cycle again until the validation is successful. This is visualized by the vertical arrows in Figure 9.

Once the validation is passed, one can move on to decomposition level 1, where one can start to specify the subcomponents of the system following the chosen working principle and embodiment. This neatly aligns with the component centered perspective of ESL, which takes the system decomposition as the central structure of the specification. The system decomposition at level *i* follows from the design decisions made at level $i-1$. For each of these subcomponents, one again follows the RACED cycle until one reaches the second validation point and one can progress to the third level, further decomposing the system.

This process continues until one reaches a sufficiently detailed specification and design to move on to the right side of the V-model. At each level and during each RACED activity, ESL and the generated dependency graph and visualization can be used for efficient requirements management and interface management. Additionally, one can generate PDF and Excel documents for different stakeholders from the same source using the document generation functionality integrated within the ESL tooling (Ratio Innovations B.V. 2022). This ensures the consistency among these documents. Of course, one will need to use other engineering software as well to do detailed simulations and create geometric models. At each level of the decomposition and throughout the development cycle, one can generate the architecture model and analyze it with architecture analysis tools such as those of Paparistodimou *et al.* (2020) to improve the architecture. Albers *et al.* (2019) demonstrated how MBSE can support modular design and evaluated their framework for five case studies, one of which about requirements engineering. The ESL tooling comes with integrated DSM analysis and clustering tools to support the multilevel architecture analysis.

In particular application cases, we experienced that it may also be effective to start lower in the decomposition tree. When applying ESL during the design of several advanced research setups for the Dutch Institute of Fundamental Energy Research (DIFFER; http://www.differ.nl/), we experienced that the early design process happened to be rather organic and that the system decomposition tree frequently changed as new (better) ideas came up, requirements changed, or concepts turned out not to be feasible. The research setups required the integration of very specific sets of off-the-shelf diagnostic tools to study highly specific physical phenomena. For these diagnostic tools, there exist only a few suppliers in the world. Furthermore, another requirement was to leverage experience obtained with previous setups as much as possible. Consequently, the desired systems are unique and new-to-the-world. Yet many of the components of these systems are off-the-shelf. This resulted in a rather bottom-up design process in which the challenge was to realize a system in which the physical phenomena could be realized, controlled and studied with the required diagnostic tools.

ESL can support both a top-down, bottom-up or mixed engineering design process. ESL components can represent off-the-shelf components, conceptual ‘blobs’ which are in the very early design phase, or anything in between. All can exist within the same specification allowing the specification to evolve over time as more information about the system becomes available. Through the use of a single decomposition tree and by employing the goal and transformation statements in a hierarchical tree, ESL can automatically generate and maintain the mappings within and between all levels and branches of the decomposition tree as explained in Section 5.2.

Hence, ESL can be used during the design of both mature and new-to-the world systems. Beernaert *et al.* (2024*b*), for example, have used ESL throughout the design process of an optical plasma diagnostic system for nuclear fusion power plant ITER, which is currently being constructed in France.

Additionally, different branches of the decomposition tree may have different numbers of levels. This provides flexibility in modeling the various parts of the system at different levels of granularity. In general, we advise to stop decomposing a system once one reaches off-the-shelf components or components that can be designed and realized by a relatively small design team. The granularity of leaf-components in the decomposition tree depends therefore on the particular design challenge. In the specification of a pulsed-laser deposition research cluster designed for DIFFER, for example, the leaf components ranged from individual pressure sensors at decomposition level 3 to a full X-ray photoelectron spectroscopy system at decomposition level 1, both being off-the-shelf components. Albeit that the latter is a hundred-thousand times more expensive than the prior. For more guidelines on how to use ESL in practice, the reader is referred to the ESL specification 101 pages (Ratio Innovations B.V. 2023).

ESL has shown to be a flexible language allowing one to create specifications and derive architecture models for both new and (partially) existing systems following a top-down, bottom-up or rather organic process that can evolve over time. This evolution of ESL specifications can be easily managed using conventional version control software such as GIT and SVN due to the text based nature of ESL. That is, one can create different branches to explore different concepts, generate different architectures and compare discuss them, and merge them if desired.

The semantics and use of ESL has commonalities with IDEF0, a long existing standard for functional modeling (see, e.g., the book by Buede 2009). Both build upon functions, flows and decomposition. The fundamental difference is that IDEF0 takes a (transformation) function centric approach, while ESL is component-centric. ESL associates the transformation functions to the components and uses goal functions to specify the flows between components.

In ESL, decomposition of a component implies that the transformation functions embodied by the component are represented in greater detail by subcomponents and their transformation and goal functions. So, in ESL, function decomposition takes place through the transformation functions, similar to IDEF0.

ESL may look at first glance oriented on modeling physical architectures. However, by adopting the method of use of ESL, as explained in this section, and by taking care that the choice of nouns for the components is sufficiently solution free, ESL actually enables a combined approach to functional and physical architecture modeling. The allocation of goal and transformation functions naturally follows design decisions on working principles and embodiments through all levels of the system decomposition and may be revised and detailed as the system architecting progresses.

What is more, other types of requirements, such as performance and design requirements, can be intuitively associated with the various functions, flows, variables and components, providing for the desired bridge between requirements engineering and systems architecting.

7. Concluding remarks

Writing 'high-quality' system specifications is essential to the success of a design project (Buede (2009)). In this article, we present a novel textual specification language, named ESL, which enables the structured specification of a system's functions, behavior and design in terms of needs, requirements and constraints at various levels of granularity following the systems engineering V-model.

Components are the basic building blocks of an ESL specification. Needs, (function/behavior/design) requirements and (function/behavior/design) constraints are specified within component definitions. ESL distinguishes between goal-function and transformation-function requirements and constraints. Goal functions denote the purpose of components with respect to other components within the system. Transformation functions describe the internal flow conversion processes in a component. Design specifications denote bounds on variables that describe the design of a system.

A compiler has been developed that checks the consistency of ESL specifications and derives dependencies between components, variables, needs, goal specifications, transformation specifications, design specifications, relations and combinations thereof across all decomposition levels of the specification, following a predefined set of mathematical rules. This provides a major added value compared to existing architecture modeling tools where the assignment or derivation of such dependencies typically has to be ad hoc implemented and checked for correctness and consistency, which is labor intensive and error prone.

Dependencies between specification elements can be visualized at each decomposition level in graph or multidomain-matrix (MDM) form. This is illustrated for a small example problem. The general method of use of ESL closely aligns with the commonly used SE V-model. This has been exemplified by presenting a variant of the V-model that stresses the iterative decomposition-based design and development process. The system decomposition is the central tree in writing the ESL specifications, which follows the design concept and embodiment decisions. The graphs and MDMs clearly show the dependencies between components, goal functions, variables and combinations thereof. By analyzing the network of dependencies, one can gain insight into the (functional) system architecture.

A specification language such as ESL has the potential to reduce ambiguity and inconsistencies in system specifications. Moreover, by automatically deriving all dependencies between a system's components, functions and variables, engineers can quickly increase their understanding of the specified system and optimize the system architecture, effectively blending requirements engineering with systems architecting. This is the main added value of the proposed language.

The ESL compiler and all associated visualization and analysis tooling has been made open-source and freely available within an online repository (https://pypi.org/project/raesl/) to allow for easy and wide spread adoption and further development among systems engineers and architects.

Acknowledgements

We would like to thank Walter Nieman, Erik-Jan Houwing, Maria Angenent and Robert de Roos for their feedback and support for this work. We would also like to thank the anonymous reviewers for their comments and suggestions, which have improved the article.

Financial support

Financial support for this research by Rijkswaterstaat, part of the Dutch Ministry of Infrastructure and Water Management, is gratefully acknowledged.

References

Albers, A., **Bursac, N.**, **Scherer, H.**, **Brik, C.**, **Powelske, J.** & **Musschik, S.** 2019 Model-based systems engineering in modular design. *Design Science* **5**, e7.

Arora, C., **Sabetzadeh, M.**, **Briand, L. C.** & **Zimmer, F.** 2014. Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns. In *2014 IEEE 4th International Workshop on Requirements Patterns (RePa)*, pp. 1–8. IEEE.

Beernaert, T. F., **de Baar, M. R.**, **Etman, L. F. P.**, **Classen, I. G. J.** & **de Bock, M.** 2024a Managing the complexity of plasma physics in control systems engineering. *Fusion Engineering and Design* **203**, 114436.

Beernaert, T. F. & **Etman, L. F. P.** 2019 Multi-level decomposed systems design: Converting a requirement specification into an optimization problem. In *Proceedings of the Design Society: International Conference on Engineering Design*, volume **1**, pp. 3691–3700. Cambridge University Press.

Beernaert, T. F., Etman, L. F. P., De Bock, M., De Baar, M. & **Classen, I.** 2022 Challenges of big-science: A matrix-based interface model to manage technical integration risks in multi-organizational engineering projects. In *Proceedings of the 24th International DSM Conference (DSM 2022)*, pp. 1–10. The Design Society.

Beernaert, T. F., Verlaan, A. L., De Bock., M., Moser, L., Etman, L. F. P., Classena, I. G. J. & **De Baar, M. R.** 2024b Multi-level architecture modelling and analysis: The case for model-based systems engineering of fusion diagnostics. *Fusion Engineering and Design* **205**, 114571.

Boehm, B. W. 1988 A spiral model of software development and enhancement. *Computer* **21** (5), 61–72.

Buede, D. M. 2009 *The Engineering Design of Systems: Models and Methods*, 2nd ed. John Wiley & Sons.

Caldwell, B. W., Thomas, J. E., Sen, C., Mocko, G. M. & **Summers, J. D.** 2012 The effects of language and pruning on function structure interpretability. *Journal of Mechanical Design* **134** (6), 061001.

Campo, K. X., Teper, T., Eaton, C. E., Shipman, A. M., Bhatia, G. & **Mesmer, B.** 2023 Model-based systems engineering: Evaluating perceived value, metrics, and evidence through literature. *Systems Engineering* **26** (1), 104–129.

Cascini, G., Fantoni, G. & **Montagna, F.** 2013 Situating needs and requirements in the FBS framework. *Design Studies* **34** (5), 636–662.

Chen, X., Chen, C.-H., Leong, K. F. & **Jiang, X.** 2013 An ontology learning system for customer needs representation in product development. *The International Journal of Advanced Manufacturing Technology* **67**, 441–453.

Childers, S. R. & **Long, J. E.** 1994 A concurrent methodology for the system engineering design process. In *INCOSE International Symposium*, pp. 226–231. Wiley Online Library.

Clark, P., Harrison, P., Jenkins, T., Thompson, J. A. & **Wojcik, R. H.** 2005 Acquiring and using world knowledge using a restricted subset of english. In *FLAIRS Conference*, pp. 506–511.

Clarkson, P. J., Simons, C. & **Eckert, C.** 2004 Predicting change propagation in complex design. *Journal of Mechanical Design* **126** (5), 788–797.

Cohen, E. 1990 *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer Science & Business Media.

Crilly, N. 2013 Function propagation through nested systems. *Design Studies* **34** (2), 216–242.

De Gea, J. M. C., Nicolás, J., Alemán, J. L. F., Toval, A., Ebert, C. & **Vizcaíno, A.** 2012 Requirements engineering tools: Capabilities, survey and assessment. *Information and Software Technology* **54** (10), 1142–1157.

De Saqui-Sannes, P., Vingerhoeds, R. A., Garion, C. & **Thirioux, X.** 2022 A taxonomy of MBSE approaches by languages, tools and methods. *IEEE Access* **10**, 120936–120950.

Deng, Y.-M. 2002 Function and behavior representation in conceptual mechanical design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **16** (5), 343–362.

Dermeval, D., Vilela, J., Bittencourt, I. I., Castro, J., Isotani, S., Brito, P. K. & **Silva, A.** 2015 Applications of ontologies in requirements engineering: A systematic review of the literature. *Requirements Engineering* **21** (4), 405–437.

Dori, D., **Reinhartz-Berger, I.** & **Sturm, A.** 2003 OPCAT – A bimodal case tool for object-process based system development. In *Proceedings of the 5th International Conference on Enterprise Information Systems*, pp. 286–291. IEEE.

Dym, C. L. & **Brown, D. C.** 2012 *Engineering Design: Representation and Reasoning*, 2nd ed. Cambridge University Press.

Eckert, C. M., **Stacey, M. K.** & **Clarkson, P. J.** 2003 The spiral of applied research: A methodological view on integrated design research. In *ICED 13 International conference on engineering design*, pp. 1–10. The Design Society.

Eggert, R. 2005 *Engineering Design*. Pearson/Prentice Hall.

Eisenbart, B., **Blessing, L.** & **Gericke, K.** 2012 Functional modelling perspectives across disciplines: A literature review. In *Proceedings of 12th International Design Conference*. The Design Society.

Eppinger, S. & **Ulrich, K.** 2015 *Product Design and Development*. McGraw-Hill Higher Education.

Eppinger, S. D. & **Browning, T. R.** 2012 *Design Structure Matrix Methods and Applications*. MIT Press.

Eppinger, S. D., **Joglekar, N. R.**, **Olechowski, A.** & **Teo, T.** 2014 Improving the systems engineering process with multilevel analysis of interactions. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **28** (4), 323–337.

Erden, M. S., **Komoto, H.**, **van Beek, T. J.**, **D'Amelio, V.**, **Echavarria, E.** & **Tomiyama, T.** 2008 A review of function modeling: Approaches and applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **22** (2), 147–169.

Ericson, A., **Müller, P.**, **Larsson, T. C.** & **Stark, R.** 2009 Product-service systems from customer needs to requirements in early development phases. In *Proceedings of the 1st CIRP Industrial Product-Service Systems Conference*, p. 62. Cranfield University Press.

Estefan, J. A. 2007 Survey of model-based systems engineering (MBSE) methodologies. *Incose MBSE Focus Group* **25** (8), 1–12.

Feiler, P., **Delange, J.** & **Wrage, L.** 2016 A requirement specification language for AADL. Technical report, Carnegie Mellon University.

Fernandes, J. M. & **Machado, R. J.** 2016 *Requirements in Engineering Projects*. Lecture Notes in Management and Industrial Engineering. Springer International Publishing.

Forsberg, K. & **Mooz, H.** 1991 The relationship of system engineering to the project cycle. In *INCOSE International Symposium*, pp. 57–65. Wiley Online Library.

Friedenthal, S., **Moore, A.** & **Steiner, R.** 2014 *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann.

Fuchs, N., **Kaljurand, K.** & **Kuhn, T.** 2008 Attempto controlled English for knowledge representation. In *Reasoning Web*, pp. 104–124. Springer.

Ghosh, S., **Elenius, D.**, **Li, W.**, **Lincoln, P.**, **Shankar, N.** & **Steiner, W.** 2016 Arsenal: Automatic requirements specification extraction from natural language. In *8th International Symposium on NASA Formal Methods*, volume **9890**, pp. 41–46. ACM Digital Library.

Giffin, M., **de Weck, O.**, **Bounova, G.**, **Keller, R.**, **Eckert, C.** & **Clarkson, P. J.** 2009 Change propagation analysis in complex technical systems. *Journal of Mechanical Design* **131** (8), 081001.

Glinz, M 2007 On non-functional requirements. In *15th IEEE International Requirements Engineering Conference*, pp. 21–26. IEEE.

Goel, A. K., **Rugaber, S.** & **Vattam, S.** 2009 Structure, behavior, and function of complex systems: The structure, behavior, and function modeling language. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **23** (1), 23–35.

Grady, J. O. 2014 *System Requirements Analysis*. Academic Press.

Herremans, B. R., **Wilschut, T.**, **Deul, M.**, **Vink, B.** & **Brouwer, R.** 2022 Modular parametric architecture design for ferry bilge systems. In *DS 121: Proceedings of the 24th International DSM Conference (DSM 2022), Eindhoven, The Netherlands, October, 11–13, 2022*, pp. 11–20. Eindhoven University of Technology.

Hirshorn, S. R., **Voss, L. D.** & **Bromley, L. K.** 2017 Nasa systems engineering handbook. Technical report, NASA.

Hirtz, J., **Stone, R. B.**, **McAdams, D. A.**, **Szykman, S.** & **Wood, K. L.** 2002 A functional basis for engineering design: Reconciling and evolving previous efforts. *Research in Engineering Design* **13** (2), 65–82.

Hooks, I. 1994 Writing good requirements. In *INCOSE International Symposium*, volume **4**, pp. 1247–1253. Wiley Online Library.

Hull, E., **Jackson, K.** & **Dick, J.** 2017 *Requirements Engineering*, 2nd ed. Springer.

INCOSE 2023 *INCOSE Systems Engineering Handbook*, 5th ed. John Wiley & Sons Inc.

Jarratt, T. A. W., **Eckert, C. M.**, **Caldwell, N. H. M.** & **Clarkson, P. J.** 2011 Engineering change: An overview and perspective on the literature. *Research in Engineering Design* **22** (2), 103–124.

Jiao, J. & **Chen, C.-H.** 2006 Customer requirement management in product development: A review of research issues. *Concurrent Engineering* **14** (3), 173–185.

Jiao, J., **Simpson, T. W.** & **Siddique, Z.** 2007 Product family design and platform-based product development: A state-of-the-art review. *Journal of Intelligent Manufacturing* **18** (1), 5–29.

Koelsch, G. 2016 *Requirements Writing for System Engineering*. Apress.

Komoto, H. & **Tomiyama, T.** 2012 A framework for computer-aided conceptual design and its application to system architecting of mechatronics products. *Computer-Aided Design* **44** (10), 931–946.

Kools, L. 2022 A system architecture margin model for design change decision-making in mechatronic system development. Master's thesis, Department of Mechanical Engineering, Eindhoven University of Technology.

Kuhn, T. 2014 A survey and classification of controlled natural languages. *Computational Linguistics* **40** (1), 121–170.

Lough, K. G., **Stone, R.** & **Tumer, I. Y.** 2009 The risk in early design method. *Journal of Engineering Design* **20** (2), 155–173.

Madni, A. M. & **Sievers, M.** 2018 Model-based systems engineering: Motivation, current status, and research opportunities. *Systems Engineering* **21** (3), 172–190.

Mahmud, N., **Seceleanu, C.** & **Ljungkrantz, O.** 2017 Specification and semantic analysis of embedded systems requirements: From description logic to temporal logic. In *International Conference on Software Engineering and Formal Methods*, pp. 332–348. Springer.

Maier, J. F., **Eckert, C. M.** & **Clarkson, P. J.** 2017 Model granularity in engineering design – concepts and framework. *Design Science* **3**, e1.

Mavin, A., **Wilkinson, P.**, **Harwood, A.** & **Novak, M.** 2009 Easy approach to requirements syntax (EARS). In *17th IEEE International Requirements Engineering Conference*, pp. 317–322. IEEE.

Meeusen, K. A., **Etman, L. F. P.**, **Wilschut, T.**, **Peijnenburg, A. T. A.** & **Segers, H.** 2019 Evaluation of conceptual design choices using dependency structure matrix methods. Master's thesis, Eindhoven University of Technology (CST 2019.026).

Mustafa, A., **Kadir, W. M. N.** & **Ibrahim, N.** 2017 Automated natural language requirements analysis using general architecture for text engineering (GATE) framework.

Journal of Telecommunication, Electronic and Computer Engineering (JTEC) **9** (3–4), 97–101.

Oxford English Oxford 2009 *Oxford English Dictionary*. Oxford University Press.

Pahl, G. & **Beitz, W.** 2007 *Engineering Design: A Systematic Approach*. Springer Science & Business Media.

Paparistodimou, G., **Duffy, A.**, **Whitfield, R. I.**, **Knight, P.** & **Robb, M.** 2020 A network tool to analyse and improve robustness of system architectures. *Design Science* **6**, e8.

Ratio Innovations B.V. 2020a ESL enhancement proposals. https://guide.ratio-case.nl/governance/.

Ratio Innovations B.V. 2020b ESL reference manual. https://guide.ratio-case.nl/reference/.

Ratio Innovations B.V. 2020c ESL user manual. https://guide.ratio-case.nl/tutorials/.

Ratio Innovations B.V. 2022 ESL user manual. https://raesl.ratio-case.nl/usage/doc.html.

Ratio Innovations B.V. 2023 ESL specification 101. https://guide.ratio-case.nl/how-to-guides/ESL/101/.

Royce, W. W. 1987 Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pp. 328–338. IEEE Computer Society Press.

Sosa, M. E., **Eppinger, S. D.** & **Rowles, C. M.** 2007 Are your engineers talking to one another when they should? *Harvard Business Review* **85** (11), 133–142.

Steward, D. V. 1981 The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management* **28** (3), 71–74.

Stone, R. B. & **Wood, K. L.** 2000 Development of a functional basis for design. *Journal of Mechanical Design* **122** (4), 359–370.

Suh, N. P. 1998 Axiomatic design theory for systems. *Research in Engineering Design* **10** (4), 189–209.

Tilstra, A. H., **Seepersad, C. C.** & **Wood, K. L.** 2012 A high-definition design structure matrix (HDDSM) for the quantitative assessment of product architecture. *Journal of Engineering Design* **23** (10–11), 767–789.

Tomiyama, T., **van Beek, T. J.**, **Alvarez Cabrera, A. A.**, **Komoto, H.** & **D'Amelio, V.** 2013 Making function modeling practically usable. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **27**(Special Issue 3), 301–309.

Tosserams, S., **Hofkamp, A. T.**, **Etman, L. F. P.** & **Rooda, J. E.** 2010 A specification language for problem partitioning in decomposition-based design optimization. *Structural and Multidisciplinary Optimization* **42** (5), 707–723.

Ulrich, K. 1995 The role of product architecture in the manufacturing firm. *Research Policy* **24** (3), 419–440.

Umeda, Y., **Ishii, M.**, **Yoshioka, M.**, **Shimomura, Y.** & **Tomiyama, T.** 1996 Supporting conceptual design based on the function-behavior-state modeler. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **10** (4), 275–288.

van Vliet, H. 2005 *Software Engineering: Principles and Practice*. Wiley.

Wilking, F., **Horber, D.**, **Goetz, S.** & **Wartzack, S.** 2024 Utilization of system models in model-based systems engineering: Definition, classes and research directions based on a systematic literature review. *Design Science* **10**, e6.

Wilschut, T., **Etman, L. F. P.**, **Rooda, J. E.** & **Vogel, J. A.** 2018a Multi-level function specification and architecture analysis using ESL: A lock renovation pilot study. In *30th International Conference on Design Theory and Methodology, International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume **51845**, p. V007T06A038. American Society of Mechanical Engineers.

Wilschut, T., **Etman, L. F. P.**, **Rooda, J. E.** & **Vogel, J. A.** 2018b Generation of a function-component-parameter multi-domain matrix from structured textual function specifications. *Research in Engineering Design* **29** (4), 531–546.

Woodcock, J. & **Davies, J.** 1996 *Using Z: Specification, Refinement, and Proof*. Prentice Hall Englewood Cliffs.