BRIDGING THE GAP BETWEEN REQUIREMENTS ENGINEERING AND SYSTEMS ARCHITECTING

New methods and software tools have been designed to support the transition from document-driven to model-based systems engineering. Though bringing improvements, these methods and tools bring along several practical challenges. To resolve the issues, Ratio Computer Aided Systems Engineering continues the development of the open-source Elephant Specification Language (ESL), for which the foundations were laid at Eindhoven University of Technology. ESL is a language to write highly structured system specifications from which system architecture models are automatically derived. It has been designed from an engineering perspective rather than an information management perspective, with the ultimate goal of bridging the gap between requirements engineering and systems architecting.

TIM WILSCHUT

Introduction

In recent years, several methods and software tools have emerged to support the transition from document-driven systems engineering to model-based systems engineering. Though being an improvement from fully document-driven engineering, these methods and tools bring along several practical challenges.

Firstly, most of these methods and tools work like databases in which pieces of information, such as requirements, are manually labelled and linked to other pieces of information, such as elements of the product breakdown structure (PBS), to keep track of all the information (i.e. what relates to what) that is being produced during the course of a development project. This manual labelling and linking of requirements to breakdown structure elements results in a heavy administrative workload for systems engineers and architects.

With stringent deadlines it cannot be avoided that corners are cut here and there. Moreover, to create these links one requires a deep understanding and an overview of the entire system. As the complexity of systems has been increasing in recent decades, it is impossible for a person to have such a complete understanding and overview. Moreover, continuous development and many unknowns complicate matters even further.

Secondly, the quality and consistency in labelling and linking completely relies on 'good behaviour' of those who create the labels and links. There are no means to, for example, formally check whether a requirement really relates to the elements it has been linked to. Instead, one usually resorts to expert reviews, which quickly lead to lengthy discussions.

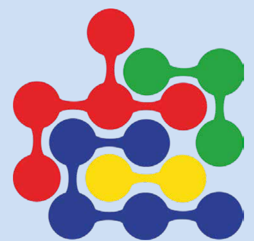
AUTHOR'S NOTE

Tim Wilschut is a co-founder of Ratio Computer Aided Systems Engineering, located in Eindhoven (NL), and hybrid lecturer at Eindhoven University of Technology (TU/e). He studied Mechanical Engineering at TU/e, where he also obtained his Ph.D.; his thesis was titled "System specification and design structuring methods for a lock product platform" [1]. Building on this work and the collaboration with the TU/e High-Tech Systems Center, Ratio develops systems engineering methods and tools. Wilschut does so together with Ratio's other co-founder, Tiemen Schuijbroek, since 2018, when they had finished their Ph.D. and M.Sc. thesis, respectively.

t.wilschut@ratio-case.nl
t.j.l.schuijbroek@ratio-case.nl

Ratio

Ratio Computer Aided Systems Engineering is a company that specialises in the development of methods and tools for requirements engineering and the modelling, analysis, and design of system architectures and product portfolios. Ratio has experience with modelling and analysing a wide variety of systems, ranging from locks and bridges to nuclear fusion power plants. Recently, Ratio set up a partnership with Quootz to allow customers to directly apply Ratio's product-portfolio analysis methods within the Quootz product configurator software.



WWW.RATIO-CASE.NL
WWW.QUOOTZ.NL

Thirdly, most tools focus on information management, not on the quality of the information being managed. As such, these tools simplify the management of large volumes of information. The information itself, however, may be just as vague and ambiguous as in a fully document-driven approach.
These combined challenges often result in a linking structure that is inconsistent and incomplete, does not provide a model of the systems architecture, and therefore has little value in everyday engineering practice.

Therefore, systems engineers and architects often resort to graphical modelling tools to create systems architecture models manually. In turn, these models must be kept up-to-date and consistent across the board as well as with their related requirement specifications, which increases the administrative workload even further and results in a gap between requirements engineering and systems architecting.

Six blind men and the elephant

When creating systems architecture models, systems engineers and architects will often find themselves in lengthy discussions on what the systems architecture really is. These discussions often resemble the parable of the six blind men that went to see an elephant. As the story goes, they explored the elephant by touch and disputed long and loud about what an elephant is, until the prince came out. "Be quiet now," he said, "because you are all in the right and you are all in the wrong." In real life, however, there is usually no prince to be found.

To resolve these issues, the open-source Elephant Specification Language (ESL) [1] [2] was created. ESL is a simple, highly structured formal language for defining a PBS and all functional, behavioural, and design requirements in a consistent and concise manner. As it is natural-language-based, it is readable by any engineer. Yet, it is sufficiently structured to allow the ESL compiler to automatically derive dependencies (links) between functional requirements, behavioural requirements, design requirements, and elements of the PBS based on mathematical rules. This network of dependencies defines the systems architecture.

Automated dependency derivation reduces the risk of human error drastically as dependencies cannot be forgotten. Moreover, it reduces the human workload tremendously. For example, at the Dutch Institute For Fundamental Energy Research (DIFFER) [3], ESL is being used to create the conceptual design of a pulsed-laser-deposition research cluster. At the time of writing, the research cluster specification defines a PBS comprising three layers containing 161 elements. These elements are subject to nearly a thousand requirements between which over 13,000 dependencies are automatically derived in less than a second, which allows for efficient systems-architecting modelling and analysis.

The text-based format allows one to easily manage ESL files using version control software such as Git and SVN, which has been used for change management in software development for decades.

A break with convention

ESL has been designed from an engineering perspective rather than an information management perspective. In other words, it has been designed to aid systems engineers and architects in designing the system by allowing one to automatically generate systems architecture models from the requirement specifications. Perhaps counterintuitively, it therefore deliberately deviates from several classical systems engineering concepts.

One decomposition to rule them all

Firstly, in classical systems engineering it is often advocated to use separate system, function, and requirement decomposition trees of which the elements are linked to another. In practice, however, systems engineers will often experience that it is very hard to create separate function and requirement trees and that when they manage to do so these structures provide little value in designing the actual system. If you have ever found yourself jumping through near poetic hoops to describe the function of something as simple as a button without being allowed to call it a button, you will be glad to leave these discussions behind.

The reason for this is quite clear; in practice it is often simply impossible to create separate function and requirement trees [4]. For example, the function 'measure temperature' might contribute to the functions 'control position' and 'control temperature'. Consequently, it is impossible to assign a single 'parent function' to the function 'measure temperature'. This also holds for requirements.

Moreover, many functions and requirements originate from design decisions rather than other functions and requirements. For example, if one decides that an actuator is to be a hydraulic actuator, the functions 'filter oil', 'store oil', 'pump oil' and requirements with respect to the type of oil appear. While choosing a spindle actuator would yield a completely different set of functions and requirements.

Hence, ESL only requires one to define the product decomposition (= PBS). All functions and requirements are formulated in terms of flows between PBS elements and properties of PBS elements. Dependencies between the functions, requirements, flows, properties and PBS elements are automatically derived by the ESL compiler.

Systems, modules, components, parts, tomàto, tomáto
Secondly, in classical systems engineering it is often advocated to give different levels of a PBS a different name. For example, the system layer, the module layer, and the part layer. In practice, systems engineers and architects often struggle with squeezing their system into such a structure. It is usually much more natural to create a decomposition in which different branches of the tree have an arbitrary number of levels.

Besides, it is relative whether something is to be considered a system, module, or part. For example, when looking at a factory one could view a machine within that factory as a component, while the supplier of the machine will view it as a system. In other words, tomàto, tomáto, it does not matter what one calls a layer of the decomposition. In essence, all elements at each layer represent a component of a system.

Therefore, in ESL all elements within a PBS are referred to as components. A component may contain other components, which in turn may contain components themselves, etcetera. This allows a user to define a decomposition tree with an arbitrary number of branches, each of which may have an arbitrary number of levels.

Blibs, blabs, blobs and off-the-shelf components
Thirdly, in classical systems engineering it is often advocated to model the systems architecture on different levels of abstraction, such as a conceptual, functional, technological and physical level. Each of these levels is successively more detailed and concrete. The consistency among these models needs to be ensured by systems engineers and architects, which contributes significantly to their administrative workload.

Moreover, it is a non-trivial task as different parts of the system are usually in different stages of development. 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.

ESL does not distinguish between components at different levels of abstraction. In fact, ESL components that represent a conceptual blib, a functional blab, a technological blob, or a physical off-the-shelf component can all exist within the same specification. This allows engineers to describe different parts of the systems at different levels of abstraction and different levels of granularity all within the same specification. One can simply let the specification evolve as more information on the system design becomes available.

Two flavours of function
Finally, in classical systems engineering function specifications are often written in unconstrained natural language. In ESL one can define goal-functions and transformation-functions following a fixed grammar:
- Goal-functions describe the purpose of one component with respect to another component via simple sentences. For example, the goal function '*the power-supply shall provide power to the electric-motor*' defines the functional purpose of the *power-supply* with respect to the *electric-motor*. In fact, it defines a functional dependency between these two components that is quantified by the variable *power* that flows from one to another.
- Transformation-functions describe the transformation of flows within components. For example, the transformation-function '*the electric-motor shall convert power into torque*' defines what the *electric-motor* shall do internally and defines dependencies between the inputs and outputs of a component.

These two flavours of function enforce users to be concise regarding the functional flows that flow through the system and define them as variables. These variables are the basis for detailed analysis models. In fact, these two flavours of function are a key feature of ESL, allowing for the automated dependency derivation throughout the PBS [5]. Therefore, this enables ESL to bridge the gap between requirements engineering and systems architecting.

The single source of truth
As mentioned before, ESL files can be easily managed using conventional version control software to create a single source of truth while allowing different exploratory branches of the specification to diverge and finally merge back into the main specification in an organised fashion. In practice, however, it is unlikely that every engineer will be familiar with ESL in the near future. This is where the generated PDF and Excel output comes in. ESL files can automatically be converted into nicely formatted documents, which – once more – guarantees their consistency.

More important and innovative, however, is that ESL specifications are automatically converted into a Python Graph object, which allows for easy manipulation, analysis, and visualisation of the specifications. Standard functions are available for visualising the system decomposition and systems architecture in various graph and matrix formats. These visualisations are guaranteed to be consistent with the written specifications and consistent among each other, as they are all generated from the same source. This provides systems engineers and architects with effective means to discuss and increase general understanding of the systems architecture.

ESL example

Listing 1 shows as an example the ESL specification of a *pump* (line 6) and a *drive-mechanism* (line 8). The purpose of the *drive-mechanism* is to provide at least 50 Nm of *torque* to the *pump* as stated by goal-requirement *g-dm-01* (line 13). The *pump* is a *CentrifugalPump* (line 18) that internally shall convert *torque* into *water-flow* as specified by transformation-requirement *t-cp-01* (line 26). The *drive-mechanism* is an *ElectricalDriveMechanism* (line 29) that internally shall convert a *power-potential* into *power* (transformation-requirement *t-dm-01*, line 38).

Listing 1. Example of an ESL specification.

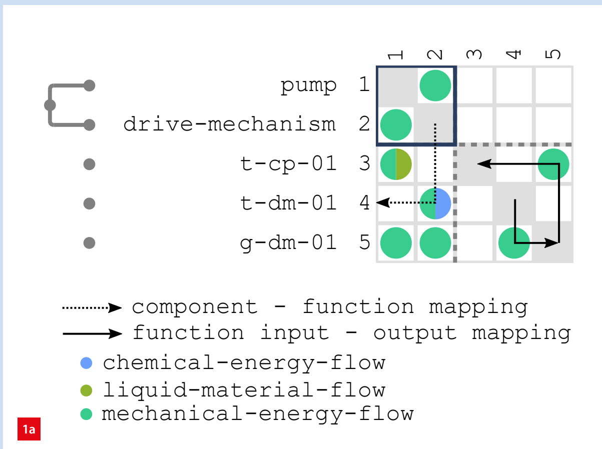
```
 1 world
 2   variables
 3     torque is a mechanical-energy-flow
 4
 5   components
 6     pump is a CentrifugalPump with arguments
 7       * torque
 8     drive-mechanism is an ElectricalDriveMechanism ...
 9       with arguments
10       * torque
11
12   goal-requirements
13     g-dm-01: drive-mechanism shall provide torque ...
14     to pump with subclauses
15       * s-01: torque shall be at least 50 Nm
16
17
18 define component CentrifugalPump
19   parameters
20     torque is a mechanical-energy-flow
21
22   variable
23     water-flow is a liquid-material-flow
24
25   transformation-requirements
26     t-cp-01: shall convert torque into water-flow
27
28
29 define component ElectricalDriveMechanism
30   parameters
31     torque is a mechanical-energy-flow
32
33   variables
34     power is an electrical-energy-flow
35     power-potential is a chemical-energy-flow
36
37   transformation-requirements
38     t-dm-01: shall convert power-potential into torque
39
40   components
41     power-source is a Battery with arguments
42       * power
43       * power-potential
44     motor is a BrushlessMotor with arguments
45       * power
46       * torque
47
48   goal-requirement
49     g-ps-01: power-source shall provide power to motor
50
51
52 define component BrushlessMotor
53   parameters
54     power is a electrical-energy-flow
55     torque is a mechanical-energy-flow
56
57   variables
58     conversion-efficiency is an efficiency
59
60   transformation-requirements
61     t-bm-01: shall convert power into torque ...
62     with subclauses
63       * s-01: conversion-efficiency shall be at ...
64         least 80 [%]
65
66
67 define component Battery
68   parameters
69     power-out is an electrical-energy-flow
70     power-potential is a chemical-energy-flow
71
72   transformation-requirement
73     t-ps-01: shall convert power-potential ...
74     into power-out
75
76 define type
77   mechanical-energy-flow is a real with unit Nm
78   electrical-energy-flow is a real with unit W
79   chemical-energy-flow is a real with unit Wh
80   liquid-material-flow is a real with unit l/s
81   efficiency is a real with unit %
82
83 define verb
84   provide to
85   convert into
86   send to
```

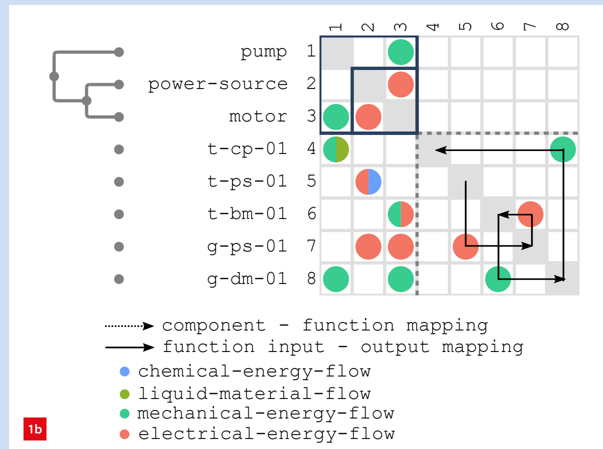
The *drive-mechanism* is composed of a *power-source* (line 41), which is a *Battery* (line 67), and a *motor* (line 44), which is a *BrushlessMotor* (line 52). The purpose of the *power-source* is to provide *power* to the *motor* as stated by goal-requirement *g-ps-01* (line 49). Internally, the *motor* shall convert *power* into *torque* with an efficiency of at least 80% as defined by transformation-requirement *t-bm-01* (line 61).

From this ESL specification, the component-function multi-domain-matrix (CF-MDM) models shown in Figure 1 are automatically generated. MDM models are a simple and compact means to visualise and analyse system architectures [6]. The left CF-MDM in Figure 1 shows the dependencies between components (rows, cols (columns) 1-2), the mapping of components to functions (rows 3-5, cols 1-2) and the dependencies between goal- and transformation-requirements (rows, cols 3-5) at the first decomposition level of the specification. The right CF-MDM shows exactly the same image, but now at the second decomposition. That is, the *drive-mechanism* has been decomposed into its sub-components *power-source* and *motor*.

Note that in Figure 1a components *pump* and *drive-mechanism* have a mechanical-energy-flow dependency (row 2, col 1), while in the right figure components *pump* and *motor* have a mechanical-energy-flow dependency (row 3, col 1). Moreover, note that in Figure 1a components *pump* and *drive-mechanism* relate to goal-requirement *g-dm-1* (row 5, cols 1, 2), while in Figure 1b components *pump* and *motor* relate to goal-requirement *g-dm-1* (row 8, cols 1, 3). This migration of dependencies and goal-requirements is automatically performed by the ESL compiler ensuring the consistency of the dependency structure throughout the decomposition tree.

Similarly, in Figure 1a one can identify the goal-function chain *t-dm-01* ➛ *g-dm-01* ➛ *t-cp-01* indicating the functional transformations and transfer of *chemical-energy* into *mechanical-energy* into a *water-flow*. In Figure 1b one





The component-function multi-domain-matrix generated from the ESL specification of Listing 1.
(a) First decomposition level.
(b) Second decomposition level.

can identify the same path of functional transformation and transfer of flows. That is, one can identify the path *t-ps-01* → *g-ps-01* → *t-bm-01* → *g-dm-01* → *t-cp-01*, which is longer than the path within Figure 1a, as the ESL compiler has automatically replaced the level-1 transformation-requirement *t-dm-01* with the level-2 transformation- and goal-requirements *t-ps-01*, *g-ps-01*, and *t-bm-01* that are performed by the sub-components of *motor*. This again ensures the consistency of the linking structure throughout the system decomposition tree.

For a more detailed ESL example and a more thorough explanation, please check out the ESL user manual [2].

The resulting dependency network can be used for interface management, change management, modularity analysis, failure mode and effects analysis, risk analysis and many other applications.

Clustering, sequencing, and comparison algorithms are available to assist systems engineers and architects in, for example, improving the modularity of the system, carrying out change management, and comparing architecture alternatives. Finally, the derived dependency graph can be exported in conventional data formats such as JSON, YAML, and XML, which allows one to import the data into other software of choice for further analysis and visualisation.

ESL and the supporting tooling have many additional features, which are not discussed within this article, such as clustering algorithms for modularity optimisation. For a more detailed ESL example and a more thorough explanation, please check out the ESL user manual [2].

REFERENCES

[1] T. Wilschut, "System Specification and design structuring methods for a lock product platform", Ph.D. thesis, Eindhoven University of Technology, 2018.
[2] *docs.ratio-case.nl/manuals/esl_manual*
[3] *www.differ.nl*
[4] N. Crilly, "Function propagation through nested systems", *Design Studies*, 34 (2), pp. 216-242, 2013.
[5] T. Wilschut, L.F.P. Etman, J.E. Rooda, and J.A. Vogel, "Automated generation of a function-component-parameter multi-domain matrix from textual function specifications", *Research in Engineering Design*, 29 (4), pp. 531-546, 2018.
[6] S.D. Eppinger, and T.R. Browning, *Design structure matrix methods and applications*, MIT press, Cambridge, MA, USA, 2012.
[7] *www.gitlab.com/ratio-case-os/python*

The bright future

Ratio is determined to continue the development and improvement of ESL and supporting tooling to fully integrate requirements engineering with systems architecting. That said, ESL and all supporting tooling [7] is open source under a GNU GPL V3 license. Hence, Ratio invites anyone interested to use the tooling and contribute to the development.

T.WILSCHUT@RATIO-CASE.NL