Explaining Embedded Software Modelling Decisions

Jelena Marincic
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
The Netherlands
Email: j.marincic@utwente.nl

Angelika Mader
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
The Netherlands
Email: a.h.mader@utwente.nl

Roel Wieringa
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
The Netherlands
Email: r.j.wieringa@utwente.nl

***Abstract*—As today's devices, gadgets and machines become more intelligent, the complexity of embedded software controlling them grows enormously. To deal with this complexity, embedded software is designed using model-based paradigms. The process of modelling is a combination of formal and creative, design steps. Because of the partially non-formal character of modelling, the relation between a model and the system cannot be expressed mathematically. Therefore, the modeller's justification that the model represents the system adequately can only be non-formal. In this paper we discuss the nature of non-formal modelling steps and pin-point those that create a 'link' between the model and the system. We propose steps to structure the explanation and justification of non-fomal modelling decisions. This in turn should enhance confidence that the non-formal, physical world surrounding the embedded system is adequately represented in the model.**

***Keywords*-embedded software; modelling process;**

I. Introduction

With the increasing complexity of today's computer-controlled systems, the role of models in software engineering in mastering this complexity is growing. A model describes a system on an abstraction level higher than that of a code, which in turn provides insight and facilitates communication. Models may be constructed as part of automated code generation and system analysis, which increases software design efficiency and lowers the number of errors.

Numerous modelling techniques and tools support different steps of software design, testing and verification. In this article, we focus on modelling of embedded systems, for the purpose of formal verification, using model-checking. But, rather than focusing on computational aspects, we will look at modelling as a design activity.

No matter whether we model software only or the whole system, the software environment has to be represented in the model. Modelling software environment bridges the physical world and the mathematical world, as illustrated in Figure 1. In the physical world, there is a software embedded in a system consisting of mechanical, electrical and other parts. The goal of formal verification is to find out whether the system satisfies the requirement of interest. Therefore the model has to mimic all (and only) the system components and aspects relevant to the requirement.

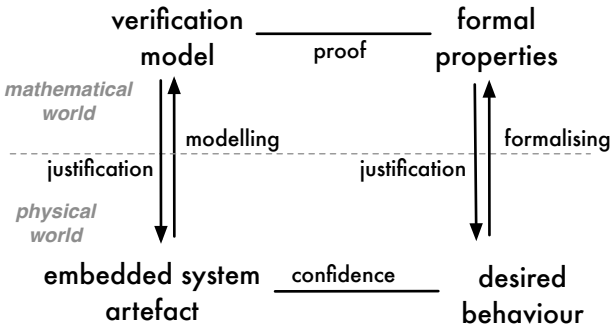

Figure 1. Relationships between the model and the system.

The model is a formal object and property checking, transformation, and any other manipulation are mathematical activities. However, model construction, a design activity in its nature, is a combination of formal and non-formal steps. Choosing a particular solution pattern for the modelling problem or deciding on criteria to decompose a system are examples of non-formal design decisions. They are intertwined with mathematical transformations and operations.

The (partially) non-formal nature of model construction steps prevents formal expression of the model-system relationship. Consequently, to justify the model's adequacy, or to just explain the model to the stakeholders, the modeller has to justify the modelling decisions non-formally, relating parts of the system with parts of the model.

Non-formal modelling decisions are mostly not documented and they also often stay implicit. This state of affairs should be improved for several reasons. One is to improve the justification of the claim "the model has been proven to have property P, so the modelled system also has property P". Another is to to improve our capability to teach modelling to others (to students or to other engineers).

In this article we propose a taxonomy of non-formal modelling decisions as a first step towards understanding these decisions. Knowing what these decisions are, could at the same time prescribe what decisions the modeller has to address when explaining or justifying the model to others.

We focus on the explanation of an already existing model.

The model is designed through a discovery process, which is not a straightforward course of actions, but a rather curvy path of exploration of possible solutions, with many dead-ends. In contrast, justification or explanation describes model construction as a structured, straightforward, monotonic process [18]. To be able to understand the model, it is not necessary to go through all the discovery steps in a chronological order; it is sufficient to outline some of the construction steps that demonstrate which components of the model represent which components of the system and give the rationale of a chosen solution. In this article we propose a structure of such an explanation.

We view modelling as a design activity and as an engineering activity. As a design activity, modelling is a problem-solving activity. As an engineering activity, modelling deploys steps that are also used in science when representing physical phenomena with a model or a theory.

The characterisation we present combines elements addressed by design science and philosophy of science (references will be given later). A body of work in these two areas refers to sciences other than computer science and engineering disciplines other than software engineering. We compare and relate the concepts we found in the literature with the steps used in software-intensive systems modelling. This paper is continuation of our previous work [15] in which two of the authors co-authored a paper on modelling decisions. There, modelling decisions are classified in form of questions that the modeller has to answer when constructing a model. In this paper we relate some of those steps with what is now known about modelling and abstraction in other branches of science and engineering. We also distinguish between the 'low-level' steps that result in concrete elements of the model and 'high-level steps' that influence the model as a solution, but are not 'tangible' in form of concrete model elements.

The rest of the paper is organised as follows. Section II looks at the problems characteristic to embedded systems modelling. These problems are related to the fact that the modeller has to represent part of the software environment in the software model. Section III pin-points low level modelling steps - those that result in concrete model components and Section IV adresses high-level modelling steps - those that explain why certain modelling decisions are made. In Section V we will discuss how our analysis can be useful for explanation and justification of modelling decisions.

II. Why is Modelling Difficult?

The challenges of embedded software modelling emerge from (1) the necessity to describe physical world when designing or verifying software, without knowing in advance what parts exactly have to be described, (2) heterogeneity of the physical environment designed by other domain experts, and (3) lack of proof that the model represents the software and the physical world adequately. The latter is inherent for all modelling pursuits, not just embedded software modelling.

A. The Physical Aspect

Embedded software controls the rest of the system (a *plant* as we call it) that consists of mechanical, electrical, and other parts, like for example chemical components. Most of the software modelling efforts focus on modelling the software only. But, no matter whether we design only a software specification or we describe both the software and the plant together, the plant has to be represented in the model.

The necessity to describe the physical plant is not surprising, knowing that software is a solution to a problem which, generally speaking, can be formulated as a need to change something in the plant, which surrounds the software [10]. Even though responsible for the plant behaviour, the software can directly access only actuators and sensors. To be able to 'calculate', or 'reason' about the plant parts connected directly or indirectly to the actuators and sensors, there has to be in internal representation of the plant within the software [12]. It relates signals observed on the sensors with other monitored values in the plant, and values sent to the actuators with the behaviour of the plant components.

Some models represent the plant separately from the software, for the system requirements verification. The plant model has to show the connection between the requirements on the software interface (actuators and sensors) and the overall system requirements that refer to the plant behaviour.

One of the problems in representing the plant is that the modeller has to extract what information, what knowledge about the plant is relevant as the plant representation. There is no automated process that selects the causally related events and phenomena that relate the software behaviour and the plant behaviour. Instead, it is left to the insight and experience of the modeller to identify the relevant elements. This is error-prone and difficult to learn.

B. Heterogeneity of Modelled Domains

Another problem is transforming the relevant knowledge into a suitable form. Software belongs to a discrete domain, thus its descriptions (models) are suited to be described with discrete models, such as automata, discrete message passing, or state machines, to name just a few. On the other hand, most of the plant processes are continuous. They are designed by different engineering disciplines that focus on mechanical, electrical and other non-software aspects. To address, design, validate, reason about these aspects, different mathematical frameworks, laws and theories are much more suitable than those used for the software models.

For example, the rollers that move paper sheets in an inserter (a machine that automatically folds papers and puts them in envelopes) are designed by mechanical engineers. In the domain of mechanical engineering, these rollers are represented with differential equations containing their size,

rotational and tangent speed, and their stiffness. Mechanical engineers calculate the rollers' stiffness and the pressure they apply to the paper so that the paper moves along without slipping. The software modeller then has to decide which of these details are relevant to be represented in the model – is it the rollers speed in relation to the motor's current? is it the rollers spatial distribution? and so on. There are no obvious answers to these questions. Moreover, this knowledge has to be described with a discrete software model.

The problem exists even when the processes are discrete or discretised by domain experts, for example by control engineers. Modern control theory is based on discrete mathematics that takes into account aspects of signals that are not relevant to the software (such as stability and observability). The signals are represented in a mathematical framework suitable to treat these aspects. Consequently, this framework resides in orthogonally different mathematical domains than that of the software [11]. Besides, some plant descriptions are not mathematical – there are diagrams, blue-prints, natural language descriptions of the plant which parts the modeller also has to incorporate into the model.

C. *"All models are wrong, but some models are useful."*

Modelling an embedded system has to bridge between the physical world and the mathematical world, as illustrated in Figure 1. In formal verification, the goal is to find out whether the system satisfies the requirement we are interested in. Therefore the model has to mimic all the system components and aspects relevant to the requirement of interest. The requirement is encoded into an expression about a model property. In other modelling techniques, the property is not necessarily checked automatically, so we do not have a formal proof for any statement about the model.

In all these cases, we do not have a formal proof that the model is adequate. A model is a representation of a system, an abstraction that mimics a set of system aspects, processes and parts. A model simulates, imitates an aspect of interest and, as Simon [20] explained, we can look at one level of abstraction and learn about the system by analysing the model, while not knowing much about other levels and representing them with rough approximations.

As the quote [3] in the title of this subsection suggests, all models are wrong, but this does not matter, as long as the conclusion we draw about the system, by analysing the model is correct. The problem here is - how do we show that the model is adequate, that, even though it is wrong, represents all the relevant system aspects. How do we know that the modeller did not miss something important or over-approximated some of the phenomena of the system?

The answer is that we do not and canot have a formal proof for the model's adequacy. One approach is to test the model and show that its behaviour matches the system behaviour. This is not a formal proof of the model's adequacy, but it raises confidence in it. Another approach is to make an informal or semi-formal argument that demonstrates the model's adequacy. Surely the significant part of this argument is the rationale of modelling decisions and steps. In the absence of a formal proof, showing how the model is constructed is relevant in the model's adequacy argument.

III. Taxonomy of Modelling Steps

In this section we relate embedded system modelling steps to modelling steps performed in other engineering disciplines and natural sciences. The steps we will discuss are: abstraction, idealization, approximation, decomposition and localization, establishing analogies, and establishing causal relationships. Philosophers of science discuss these activities as they are undertaken in science and classical engineering disciplines. Their analysis is part of their search for answers to philosophical questions, such as: "Are scientific theories telling the truth?" and "What does it mean for a model to represent a part of the world?".

Even though our goals are not philosophical, we can use the answers found by philosophers to answer our more pragmatic question - how do we know that the model represents the system adequately? The activities identified by philosophers are the very same activities that, when explained on a concrete model, uncover how the model was constructed. These steps are the closest we can get to finding the system-model relationship, given the lack of some formal, mathematical relation.

We will use literature that explored mostly physics and fluid mechanics, focusing on comparison between the modelling activities in science and engineering. To the best of our knowledge, a compaarable analysis has not been done for embedded systems modelling.

One may ask: Why do we need to analyse these steps exclusively for software engineering? Why don't we just take over the analysis already performed for other disciplines? Our answer is that other engineering disciplines often use directly the models and theories from science and, if necessary, adapt them [2]. In embedded software engineering, as we explained in the previous section, we cannot directly use control laws, statical equations and other models as parts of software discrete models. Instead, we have to non-formally interpret and combine parts of different models into software engineering models.

Another difference between modelling in other branches of science and engineering and modelling embedded systems is that in science in general, the model is a stepping stone from the observable world to a theory. Science strives for generality and theories apply not only to a single system but to some phenomena in general. Some models in science are idealized in order to explain an isolated mechanism in nature, but they do not necessarily provide accurate numerical results. A theory that explains a mechanism holds for an idealized model, but when applied to a real phenomena, it may turn not to be completely accurate. [4]

In embedded system modelling, by contrast, we are not interested in generality, our model has to represent only the system under development. The results we get from verification models are not numbers, they are 'yes' or 'no' answers to questions about the system properties. We cannot afford a wrong answer there, the model has to be adequate, accurate 'enough' to give us the correct answer.

A. Abstractions

Abstraction is leaving out, omitting describing some of the system components, properties, processess, aspects. Underneath every abstraction lies, often implicit, reasoning why omitting certain information about the system still results in a model that adequately represents the system. Take for example, a verification model of the software embedded in the inserter we mentioned in Sect. II. The software controls transport of papers and envelopes from their designated feeders through the machine modules, to the exit. The software also controls movement of numerous mechanical parts that fold a paper, open an envelope, and insert the folded paper into the envelope. But, in the inserting module, the moisturising of the envelope flap with a stroke of a brush is controlled mechanically. The moisturising takes place simulataneously with other software controlled actions without interfering with them in any way. Therefore, the flap moisturizing is invisible for the software, and consequently irrelevant for the software model.

Philosophy of science literature lists the reasons why abstractions are performed. One is the relevance, and the example we gave falls into this category. The property under investigation, the purpose of the model, the desired preciseness of calculation or predictive accuracy do allow to ignore many parameters [5] . Another reason is pragmatic, as the number of potentially relevant factors for the property we are interested in, is extremely high, which would make constructing a model impractical [5].

In modelling for verification, we start dealing with this complexity in the early phase of the model design, while defininig the modelling problem. The modelling problem is decomposed into sub-problems by decomposing the requirement of interest into sub-requirements for which we can design models without having a state explosion or other problems related to computational complexity. If, for example, the requirement is a safe control of a car, the requirement will be decomposed into the following requirements: "When the car is on ice, if the driver hits the brake, it will be overruled by the control", "The cruise control gives the control to the driver as soon as he presses the brake or gas pedal", "Air-bag is blown if collision is detected". By decomposing the safety requirement into sub-requirements the models are less complex and easier managable.

There are no firmly established criteria on what should the 'filter of abstraction' capture and what simply does not count [22]. An analysis [22] of criteria for abstracting away aspects, parts, phenomena of the modelled target, found the following criteria in the contemporary literature: relevance vs. irrelevance, simplicity vs. complexity, manageability vs. unmanageability, tractability vs. intractability, and the author adds his own, related to the cognitive value of the model, information vs. noise. The same analysis also categorizes the abstractions done in science to those that (1) omit relevant factors for the sake of getting to partial truths that are in some way useful [4]; (2) the abstractions that omit only irrelevant factors, so even if they were incorporated in the model, they would not change the modelling result; (3) the abstractions made for the sake of simplicity, aimed at reaching high cognitive value, rather than providing calculation or prediction. The latter criterion is implicitly present in best practices of modelling in software engineering, where for example layout of diagrams has to be 'clean' and readable.

In embedded system formal verification, we want the models that give truthful predictions, confirmations and disputes. Does that mean that we cannot afford the abstractions that, when put into the model, would change verification result? By decomposing the problem into computationally tractable and managable problems, we eliminate partially the problem of having to abstract away for pragmatic reasons. However, there are many modelling assumptions that are not part of the model. These assumptions are critical for the model's adequacy because, if they are not fulfilled, the verification result may not hold. For example, we can say our model describes the system that operates on temperatures between -10 and +50C, but nowhere in our model we represent temperature values.

In many cases, abstractions are made for the goal of hiding information [6], likr for example when using encapsulation. But, modellers also completely omit certain aspects of the system, like in the cases in which they abstract away the plant description and prove only the requirements at the software interface. They implicitly assume that the behaviour on the software interface will result in the desired behaviour of the whole system. The plant components are not irrelevant, but they are (or should be) addressed separately from the model.

When describing the plant, the modeller chooses what aspects to desribe and what to leave out. The plant is described by different domain experts, such as mechanical and electrical engineers. For software verification only some of the descriptions given by different experts is relevant. For a mechanical engineer, the stiffness of a material and other mechanical properties are relevant, the dimensions play a role, and their statics. The modeller of the plant for the purpose of software verification needs to know the spatial distribution of some of the elements, how the mechanical elements are connected, and how movement of one influences the movement of the other components.

B. Idealizations

Unlike abstractions that omit the truth by not describing parts of the system, idealizations introduce distortions or, we can say, false assumptions. The modeller describes certain system aspects as if they were different than they are. "Idealizations change properties of objects, perfecting them in certain ways, not all of which even approximate to reality." [22]. Point masses in calculating positions of planets rotating around the sun is a paradigm example of an idealization in physics. There is no such thing as a point mass, but in the point-mass models of Newton's and Kepler's theories they are assumed to exist, and they help understand the movement of planets and calculate their positions.

In software modelling, we often idealize events as instantaneous, like opening and closing a valve, for example. Another example is representation of sensors as dimensionless points in the system. Rollers that move papers along a printer are made of soft rubber. On their contact with paper they flatten temporarily which changes their round shape and tangent speed with which paper is moved. However, for some calculations they can be idealised to perfectly round-shaped rollers without compromising the overal result.

Cartwright [4] recognizes two different types of idealizations. One type are idealizations that describe the phenomena in a way that can never be achieved in reality, like for example point masses in physics. These are useful to explain and understand the phenomena, but they never give accurate predictions. The second type are idealizations that can be approximated in reality. For example, if we assume vacuum for the law that calculates pendulum angle and velocity, we could have this in reality, by putting the pendulum in a very low pressure room. The theories based on these idealizations approximate the truth. The second type of idealizations allows for de-idealization [17], which is relaxing ideal conditions and bringing them closer and closer to reality. In science, they are performed with the goal to simplify theories and make them computationally tractable [21]. Verification models are designed to provide accurate result, without a 'second chance' to de-idealize.

A special type of the truth distortion is assumption of a worst-case scenario. The modeller assumes extreme conditions, such as extreme temperatures or pressures. These are at the same time the strongest mathematical conditions on the model, and if the property holds under these conditions, it will guaranteed holds in less strong conditions. Also, to examine the property of interest, the modeller sometimes has to invent elements that do not exist in the system, but are necessary to observe what happens in the model.

C. Approximations

Approximation is assigning a numerical value that is not the exact one, but within an acceptable error boundary. If for example the prediction we want to make with the model is of an order of minutes or hours, than we may choose to round all the values that are below seconds. For example, in model-checking language Uppaal[1]time passage is represented with time units which have to be normalised to the lowest common denominator of all the time values. If all the processes take time in order of minutes, than one time unit is interpreted as one minute. But if there is a process that lasts order of seconds, all the times have to be converted into seconds, and in the model described with time units that represent seconds. This increases the state space, and complexity of the model. So, it may be convenient, if the property of interest allows, to approximate the duration of that one single process to zero.

Laymon [14] defines a special kind of idealization, which is a *transformational approximation*. It is a substitution of a term or an expression in a mathematical expression that represents a theory or a law, with another term or an expression, under the assumption that the overall function will stay stable. Transformational approximations do not apply to discrete, state-based models we are focusing on. But, there are other transformations that the modeller invents to have a model that is simpler, smaller, easier to manage, easier to comprehend. For example, the structure of the model maybe a structure that is not present in the system, but that shares the relevant properties with the system.

D. Abstractions and Idealizations Relationship

In many cases both idealizations and abstractions are performed at the same time, or even more, they are closely linked. We mentioned an example of dimensionless sensors in the model, but it is difficult to say whether we just abstracted away the dimension or idealize sensors in form of points.

When explaining the model, we do not necessarily separate abstractions, idealizations and approximations from each other. A modelling decision the modeller finds relevant to address explicitly can be a mixture of these. Still, by knowing the distinction between these decisions we have three ways to justify a difference between the model and the modelled system. In the case of an abstraction, we left out the elements that do not influence the property we want to prove (or we address them separately). In an idealization, we added things to make computations in the model easier, without compromising the results interpretation back in the modelled system. In an approximation, we argue that the model is a good-enough approximation of what is the case in the real-world; it will produce a slightly wrong prediction but that prediction is still good enough for our purposes, e.g. within safety margins.

E. Finding Analogies

Analogies represent one phenomena with phenomena of a different nature. In physics, a typical example of analogical

[1]http://www.uppaal.com

model is a model of light based on an analogy to water or sound wave. Analogies are based on similarities of different phenomena. An analogical model incorporated into a computer-based system possesses phenomena and properties which have no counterpart in the subject [13]. If we go back again to the example of the inserter machine, modelling exact layout of its feeders and rollers that move papers might result in a model that is too complex. Instead, it is possible to model a machine with a different, easier to model layout of the feeders, as long as the properties of interest described in the model remain unchanged. When looking at the model only, without knowing the system layout, one would assume the layout of the inserter whose relevant behaviour is analogous to the inserter behaviour.

F. Decomposition (and localization)

Decomposition is a way to deal with complexity. Bechtel et al. [1] distinguish between decomposition – giving structure to observed system or phenomena, and localization – tracing what part is responsible for a certain function. They define decomposition and localization as strategies for discovering mechanisms for articulating structure of the phenomena. We look at decomposition and localization under the umbrella term of decomposition.

Decomposition, as we define it, consists of a top-down and a bottom-up type of process. In the bottom-up process, the modeller assigns the structure to the system by recognizing parts that have the same characteristics. In the top-down process the modeller isolates different components and discovers what each one does. After performing a combination of these two types of processes, the modeller is able to show different structures and parts responsible for different functions. Together with decomposition there is a formal or non-formal, implicit or explicit argument of recomposition.

One might argue that man-made, engineered artefacts reveal their parts which makes their decomposition easier than the decomposition of phenomena in the nature. However, today's computer-controlled systems are so complex, that there are many possible structures that the modeller has to explore. These systems do not have a single unique decomposition. For verification and simulation models, the modeller has to do the following decompositions.

- Decomposition of the modelling problem. The problem is almost always expressed in vague terms, and even if it is not, it may happen that the requirement has to be decomposed into sub-requirements.
- Decomposition of the system. The system is built by different engineers, it has different components, functions, performs different processes; all these have to be explored to understand the modelling problem.
- Decomposition of the model. Related to different structures of the system are different perspectives from which we can view the system. They usually come from different engineering disciplines.
- Level of abstraction with which software and the plant are represented is about abstractions, but it can be argued that how we establish hierarchies is decomposition, too.

Decomposition of the model and decopmosition of the system do not necessarily coincide. If they do, we have isomorphism between the system and the model components. If not, we argue implicitly or explicitly that the two structures are equivalent regarding the property and behaviour of interest. We may have a partial isomorphism if only some of the model components are isomorphic to the system copmonents; or, if there is an isomorphism on some of the abstraction levels, but not on all of them.

The structure of the model depends often on the structure of the system, but it also depends on the modelling method chosen. If we chose an object-oriented method, we will 'see' the system as composition of objects, which will be represented in the model, together with their internal structure, functions they perform, etc.

G. Representing causal relationships.

Causal relationships show how the behaviour on the software interface causes the plant as a whole to behave as required. The software can interact directly only with actuators and sensors, whereas the system requirement refers to the plant parts that are usually on some distance from the sensors and actuators. Therefore, the task of the modeller, and of the software designer, is to establish causal relationships between the system components that result in the overall system behaviour. For example, the relationship between warming up food in a microwave oven, and the sensors and actuators of the embedded software in the oven is: the motor rotates the plate holder, the magnetic element emits the energy when the software sends the signal to the switch, the sensor senses the temperature, and the timer measures the time. Usually this reasoning is performed informally and stays implicit. Seater et al. explored how these relationships can be made explicit [19].

When we model the system as a whole, we do not necessarily separate the software and the plant, so these relationships are formally described with the model. When describing software only, there is, within the software specification, a representation of (a part of) the plant and causal relationships [12]. Variables or structures keep track of sensors' values and the order of events and based on this data they determine the state of the components that are not directly connected to the software.

While decompositions and components that are the result of abstraction and idealization focus either on static picture of the system, or if they describe dynamic aspects like processes, they show then their static represantation in the model. Causal relationships cannot be pinpointed in the model, they are not documented as such, but can be shown by analysing multiple model components model.

IV. Modelling Process

The steps we identified in the previous section are those that the modeller performs *after* making the decision what parts of the system to represent in the model. These steps show *how* the selected elements are represented in the model. But, these steps do not explain *why* a system element is chosen to be represented in the model. In explanation of the modelling decisions we have to have a wider picture of the modelling problem. We will identify the rest of the model explanation elements through analysis of modelling as a design process.

Modelling is a conceptual design activity. In software engineering, one of the commonly used paradigms to describe software design is that of problem analysis, solution design and solution validation [8]. Different structures for the software development process, such as the V-model, the incremental model, iterative cycle models, all contain these three aspects as phases of software design. Applied on modelling, these three activities are: modelling problem analysis, model design and validation or justification of the model. In the rest of this section we will examine the processes and steps within each of these three activities. We will not suggest their chronological order, given that our goal is neither to describe evolutionary nor incremental character of the model design process. Also we will not investigate relations between all the processes and steps.

A. Problem Analysis

Like most of the design problems in software engineering, a modelling problem is almost always ill-structured. This is the term used by Simon [20] to characterise problems that are not well-defined. Translated to modelling, this means that the modelling problem, the system and the system requirement are often formulated in vague terms. It is the task of the modeller to refine the problem definition. Figure 2 shows our classification of the problems that that the modeller has to solve while designing a model.[2] This classification is based on the analysis of the case studies we performed, and on the analysis of software engineering processes in general [10].

1) Analysing the Modelling Problem: There is no single "correct" model of a system that can describe the system for different purposes. The model and the purpose for which the model is designed are inseparable and together determine how the model will be designed. A number of modelling decisions, such as the choice of an abstraction, depend on the purpose of the model. Therefore, an explicit statement of the purpose is therefore a prerequisite for both, construction and justification of a model.

In case of verification models, the purpose is determined by the property to prove. For a simulation model, it has to be defined what are the phenomena that will be observed. Identification of all these may not be known in detail beforehand. Also, formalising a property or making precise what is observed is part of the modelling process. Therefore, the statement of the purpose of a model may be constructed incrementally.

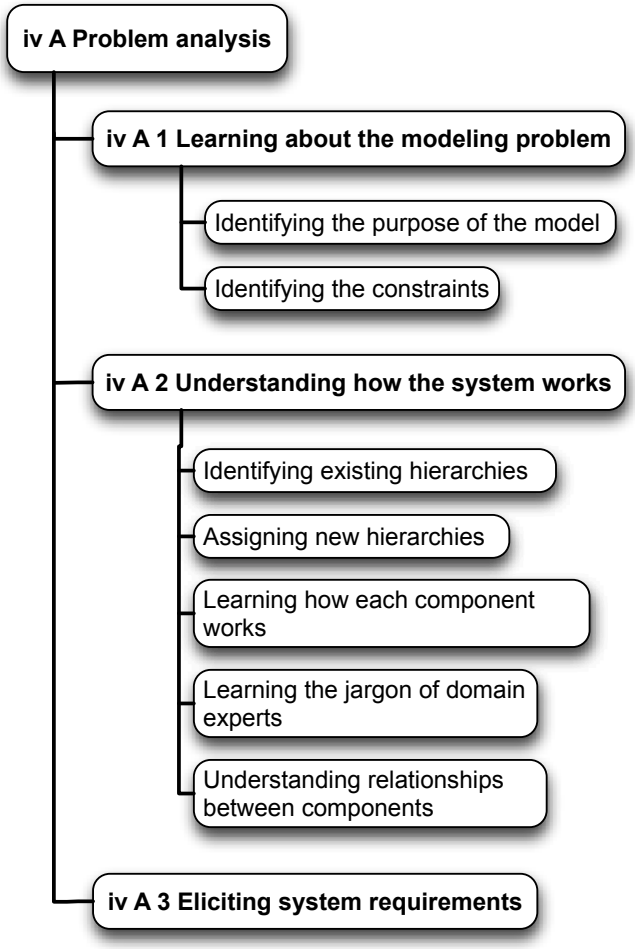

Figure 2. Activities that are part of problem analysis during model design.

There can be a number of limitations posed by stakeholders on the model itself. They can concern the choice of modelling language, the decomposition of the model, the resources available to build the model (in case of the model is implemented with hardware elements) and many others. This, too, influences the modelling decisions and narrows down the solution space. An explanation of practical constraints clarifies why certain modelling choices prevailed over the other which are as good or which would lead to a more simple, understandable or easier to manage model (or whatever is the quality criteria of a stakeholder's interest).

[2]The activities on this and other diagrams are numbered to correspond the numbering of the sections discussing them.

2) Understanding how the system works: To be able to determine what to model, the modeller needs to understand how the system works, what the system requirements are and what the stakeholders want with the model. The modeller is not necessarily someone from the organization that designs the embedded system, therefore not someone who knows the system. Very often, the modeller starts with the system and the modelling problem as black boxes and unveils them gradually, by learning about the system. To be able to understand how the system works, the modeller has access to technical documentation of the system, the system stakeholders, and sometimes the system itself. In an industrial organization, the stakeholders have their own jargon, so the modeller also learns the 'language' of domain experts, or languages, acknowledging that different domain experts sometimes use the same words for different meanings (and different words for the same meaning).

In this process, the modeller learns about the existing system decompositions. In practice, these decompositions are often mixed, made according to different criteria. Even though they are not 'perfect' they serve purposes of the organization. But the decomposition is not a tangible artefact, it is a concept, and it is what people assign to complex systems. Therefore, the modeller can also define her own decompositions while learning about the system.

The modeller also focuses on individual components and modules and learns how they work. Of course, it is not possible to get into all the details, but it is necessary to find out how these components contribute to the overall system behaviour. The modeller also needs to establish how different components interact among themselves. In relation to this process, we identify two additional elements of the model explanation. They are: explanation why a chosen abstraction level is accurate enough and an explanation or justification that the model structure is adequate. In the latter case, it can happen that the structure of the model does not coincide with any of the existing system structures. In this case it might be necessary to justify that the model is equivalent to the model that would have the same structure.

3) Eliciting system requirements: When modelling is used for design, the modeller has the role of the software specification designer as well. Inherent to software specification is eliciting system requirements, and this is also done incrementally.

B. Model Construction

We distinguish two main aspects of model construction (see Fig. 3). One is the decision *what* aspects, parts, properties, phenomena etc. of the system will be represented in the model. The other are the concrete modelling steps, and the decisions that have as a direct result a part or a segment or an element of the model constructed. These decisions are about *how* the system will be modelled and we have already talked about them in Sect III.

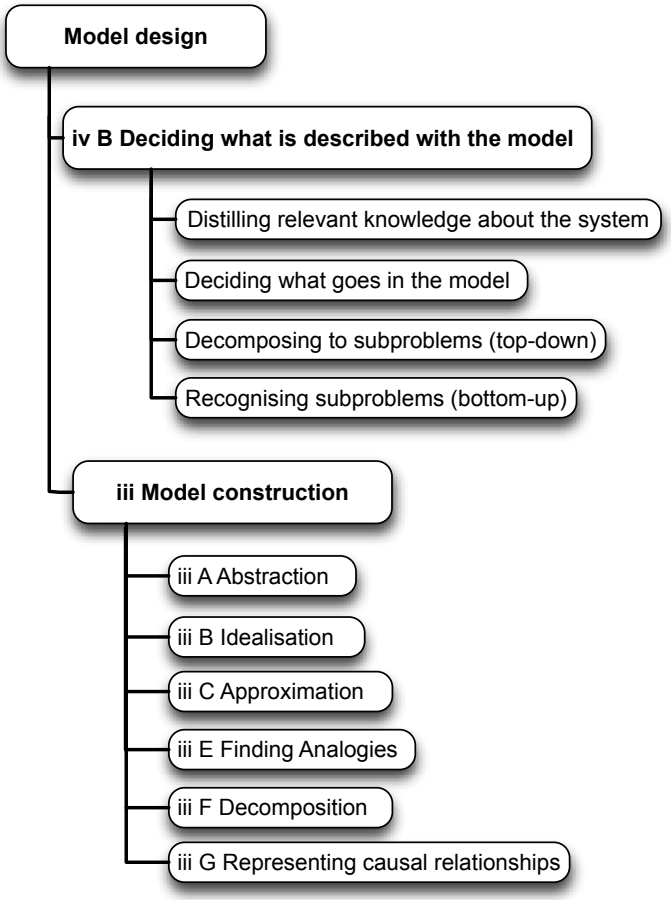

Figure 3. Model construction steps and decisions.

We classify the decision on what to model as part of the solution design, rather than the problem analysis because it is up to an extent the modeller's decision what will go in the model.

The modeller will get enormous amount of information about the system and about the modelling problem, and after resolving possible contradictions, learning how the system works and what the stakeholders really want with the model, she will distil the parts that are relevant for the requirement to analyse. It can happen though that some of the relevant information about the system is not represented in the model, but is left as a modelling assumption, a condition under which the model is adequate.

There is a large number of assumptions that have to hold in order to have the model that represents the system adequately. They usually stay implicit, but as we showed in our earlier work, a number of them can be extracted in form of lists [16]. A list of assumptions can be part of an explanation of when the model is adequate.

Another aspect of the solution invention is that the mod-

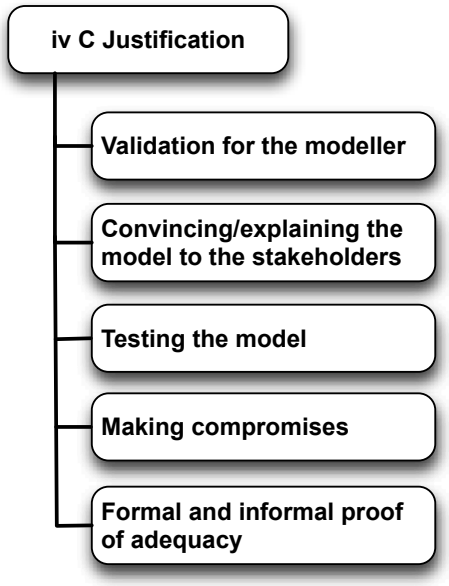

Figure 4. Aspects of justification that a model is adequate.

eller does both top-down decomposition of the problems and bottom-up recognition of previously solved problems. Top-down and bottom-up thinking always come together in design. Cognitive science showed that a designer recognizes 'templates' - previously solved problems, and in case of modelling the modeller will recognize previously solved problems and will map them to her own 'pattern library' that resides in her long-term memory [7].

Among the decisions that concern the model components in an indirect way is the decomposition into the model sub-components. There is always a mapping between the model and the system architecture, but it is the modeller's decision what structure will be represented in the model. Assigning hierarchies and decomposing is a top-down strategy to deal with complexity.

C. Model Justification

Showing that a model is adequate can be done in a non-formal or semi-formal argument. The argument will, among possible other elements, contain the justification of the modelling steps that we analysed as part of problem analysis and model construction. So, the justification steps will be: justification of the problem decomposition, justification of an abstraction etc. Most often, justification is not structured, but is an implicit part of the communication between the modeller and the stakeholders.

Fig. 4 does not show the structure of the justification argument, but it emphasises the aspects of model justification. We distinguish between the justification that the modeller performs for herself and the justification that she possibly performs for the stakeholders. Sometimes, the model has to be explained to the stakeholders in order to convince them that the model is adequate.

Testing the model is a way to validate the model and it can be used as means to both convince the modeller herself and the stakeholders.

Sometimes, the model is not adequate but it is considered good enough for an initial analysis. The compromise is made when the model is representative enough.

A justification argument can also have formal elements. For example, if we are using model-checking we can use the technique to validate the model itself, by inventing different testing queries that will show whether the modeller designed the model she intended to.

V. Discussion and Future Work

In this article we propose ingredients of an embedded system model explanation. They are the explanation of modelling decisions which results can be shown directly in the model (these decisions we discussed in Sect. III) and they are explanation of the decisions that are part of the modelling process but cannot be pointed directly in the model.

A full understanding of the modeller's reasoning while modelling would require studies in cognitive science, which we did not do. But, for the purpose of explanation of the model, and to teach others modelling, the modeller has to address concrete decisions, not the cognitive process behind them. The goal of our analysis was to identify what these decisions are, so that they can be used as a checklist of important modelling aspects to mention in a model justification, explanation or when teaching others how to model. One can build and use this checklist even when the cognitive processes are not quite understood.

The modelling steps and processes we discuss also do not show incremental model growth. To show that the model is adequate or to just explain the model to others, it is not necessary to document the decisions in form of a diary, as the modeller ran into the need of making these decisions; only the justification of the decisions is needed.

Some of the steps are a combination of the individual steps we identified. For example, sometimes it is not easy or possible to separate a decomposition and an abstraction step. Also, addressing all the modelling steps would not be practical. We leave to the modeller which steps to explain, once the model is designed.

Ideally, we would want to formalize all the modeling steps and design a tool and a language that would design a model on a press of a button. But, the difficulty with modeling is that there are always non-formal steps that precede automation and formalization. We can extend the degree of automation by designing domain specific languages, and patterns, but it is not possible to formalize the decision to use a certain pattern, or a decision how to idealize some phenomena in the system.

Some of the non-formal steps are made explicit by defining modelling as a structured process, or collecting best practices in standards that prescribe how to make better models. Our structure is a contribution in structuring the part of the modeling process that evaluates the model's adequacy. Some authors, such as [9] formalize the process itself, in order to explicitly document certain classes of design decisions. We could map our own classification into their problem-oriented classes of design decisions, which would not formalize the decision itself, but would formalize the classes of validation arguments. For the goal of this paper this embedding would not add so much, but we plan it for future work.

Finally, our last remark is - how do we know that we covered all the relevant aspects of modelling? Designing taxonomies or making classifications are the first steps in understanding unknown or not fully understood phenomena. It may happen that the model is missing some steps. Possibly there are steps that we did not cover in our analysis, as non-formal modelling aspect is a very broad area. Still, the steps we did identify contribute to making modelling decisions explicit which raises the confidence in the model's adequacy.

For future work we plan to compare modelling decisions made by control and software engineers. The design models made by these two groups of experts are often made in isolation, using different mathematical frameworks. We plan to investigate what knowledge these experts need to share so that from the early stages of design, decisions in both domains can be optimised.

REFERENCES

[1] W. Bechtel and R. Richardson. *Discovering Complexity: Decomposition and Localization as Strategies in Scientific Research*. The MIT Press, 2010.

[2] M. Boon. How science is applied in technology. *International Studies in the Philosophy of Science*, 20(1):27–47, 2006.

[3] G. E. P. Box. Robustness in the strategy of scientific model building. *MRC technical summary report*, 1979.

[4] N. Cartwright. *How the laws of physics lie*. Clarendon Press, 1983.

[5] A. Chakravartty. The semantic or model-theoretic view of theories and scientific realism. *Synthese*, 127(3):325–345, june 2001.

[6] T. Colburn and G. Shute. Abstraction in computer science. *Minds and Machines*, 17(2):169–184, July 2007.

[7] N. Cross. Design cognition: results from protocol and other empirical studies of design activity. In C. Eastman, W. McCracken, and W. Newstetter, editors, *Design Knowing and Learning: Cognition in Design Education*, pages 79–103. Elsevier, 2001.

[8] S. Dasgupta. *Design theory and computer science: processes and methodology of computer systems design*. Cambridge University Press, New York, NY, USA, 1991.

[9] J. Hall and L. Rapanotti. Software engineering as the design theoretic transformation of software problems. *Innovations in Systems and Software Engineering*, pages 1–19. 10.1007/s11334-011-0171-2.

[10] J. G. Hall, L. Rapanotti, and M. Jackson. Problem oriented software engineering: A design-theoretic framework for software engineering. *sefm*, 0:15–24, 2007.

[11] T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[12] M. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley, 2000.

[13] M. Jackson. Some notes on models and modelling. In A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 68–81. Springer Berlin / Heidelberg, 2009.

[14] R. Laymon. Computer simulations, idealizations and approximations. *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association*, 1990:519–534, 1990.

[15] A. H. Mader, H. Wupper, M. Boon, and J. Marincic. A taxonomy of modelling decisions for embedded systems verification. Technical Report TR-CTIT-08-37, Enschede, May 2008.

[16] J. Marincic, A. H. Mader, and R. J. Wieringa. Classifying assumptions made during requirements verification of embedded systems. In *Requirements Engineering: Foundation for Software Quality, REFSQ 2008, Montpellier, France*, volume 5025, pages 141–146, London, 2008. Springer Verlag.

[17] E. McMullin. Galilean idealization. *Studies in History and Philosophy of Science*, 16:247–273, 1985.

[18] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Software Eng.*, 12(2):251–257, 1986.

[19] R. Seater, D. Jackson, and R. Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, 12(2):77–102, 2007.

[20] H. A. Simon. *The Sciences of the Artificial*. The MIT Press, 1996.

[21] M. Weisberg. Three kinds of idealization. *Journal of Philosophy*, 104(12):639–659, 2007.

[22] J. Woods and A. Rosales. Virtuous distortion model-based reasoning in science and technology. In *Model-Based Reasoning in Science and Technology: Abduction, Logic, and Computational Discovery*, volume 314 of *Studies in Computational Intelligence*, pages 3–30. Springer Berlin / Heidelberg, 2010.