Expert Systems

Article

DOI: 10.1111/j.1468-0394.2012.00631.x

Reusing knowledge in embedded systems modelling

Jelena Marincic,[1] Angelika Mader,[2] Roel Wieringa[3] and Yan Lucas[4]

(1) Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands
Email: J.Marincic@utwente.nl
(2) Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands
Email: A.H.Mader@utwente.nl
(3) Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands
Email: R.J.Wieringa@utwente.nl
(4) Neopost Technologies, Drachten, The Netherlands
Email: Y.Lucas@neopost.com

Abstract: *Model-based design is a promising technique to improve the quality of software and the efficiency of the software development process. We are investigating how to efficiently model embedded software and its environment to verify the requirements for the system controlled by the software. The software environment consists of mechanical, electrical and other parts; modelling it involves learning how these parts work, deciding what is relevant to model and how to model it. It is not possible to fully automate these steps. There are general guidelines, but given that every modelling problem differs, much is left to the modeller's own preference, background and experience. Still, when the next generation of a system is designed, the new system will have common elements with its previous version. Therefore, lessons learned from the current model could inform future models. We propose a framework for identifying the non-formal elements of knowledge, insights and a model itself, which can support modelling of the next system generation. We will present the application of our framework on an action research case – modelling mechanical parts of a paper-inserting machine.*

Keywords: model-based design, plant modelling, reuse

1. Introduction

Embedded software is part of a larger, composite system that contains mechanical, electrical and other parts in addition to software parts. We call the parts of the composite system outside the embedded software the *plant*. The purpose of the embedded software is to control the plant, so that the composite system exhibits a desired behaviour. The problem investigated in this paper is how to model the embedded software *and* the plant so that we can prove, or convincingly argue, that the embedded software controls the plant correctly.

Plant modelling cannot be fully formalized and automated because the process is complex and unpredictable, which stems from the following two issues:

(1) 'Elements of the plant that need to be represented in the model are not known in advance'. Modellers go through a process of discovery to determine what elements should go into the model. They engage in an iterative learning process while talking to other engineers (domain experts), examining existing documentation and designing the first versions of the model. In this process they also need to check whether they understand correctly how the plant works.
(2) 'Non-formal parts of the modelling process leave too large a scope for possible models'. The modelling activity is a combination of formal and non-formal steps. The latter leave modellers with many, possibly equally suitable, solutions. Given the number of unknowns and the partially non-formal nature of the modelling process, the experience of the modeller is very important to make this process efficient.

Domain knowledge acquisition and other non-formal steps of model design, by their nature, are such that they cannot be solved with a mathematical formula, an expert system, or a modelling language. Different languages and approaches offer varying guidelines on how to build a model, but because every problem is different, many decisions remain open and are left to the modeller. What were good modelling decisions and what the designer needed to know about the plant is often known *a posteriori*.

Still, when the next generation of a system is designed, the new system will have common elements with its previous version. Therefore, the plant-modelling problems will also have common elements. Lessons learned from the first model could inform future models. To narrow the gap between general guidelines and concrete plant-modelling steps, we propose to focus on a class of modelling problems and make explicit some of the knowledge, insights and modelling steps in form of explicit guidelines that can be used in a similar modelling problem in the future.

Suppose, we have a model of a plant designed from scratch. Some time has been spent to design the model that suits its purpose in a given context and more time has been spent to validate the model's adequacy. The question is how the knowledge and experience invested in this process could be reused when a similar model is built, one which is the next generation of the system, but where the organizational context remains the same. The old model cannot be directly reused because system requirements and some of the system parts will change. What can be reused are modelling knowledge and some of the design decisions.

Currently, the problem in modelling is that this knowledge transfer rarely takes place, if ever. Usually, only the model is delivered to its users and the knowledge accumulated during modelling is lost. This is particularly problematic nowadays, when work is outsourced to temporarily hired consultants or to companies at long distances: implicit knowledge is immediately lost when the contract is complete. This knowledge transfer is critical for two reasons: to shorten the time for modelling and to preserve uniformity of modelling decisions.

To assist with this necessary knowledge retention, we propose a framework that specifies what elements of a model should be extracted for reuse. Recording some of the modelling decisions and reusing them makes the future models less dependant on a modeller's personal style. To identify elements of our framework, we have to solve two problems:

(1) What are the non-formal elements of a modelling method that can be reused?
(2) How to acquire this knowledge if it has not already been recorded during modelling?

We will solve these problems using a concrete example from an action research study, and we will then make some generalizations. Our generalization can serve as conceptual framework to extract reusable modelling elements in different cases. Before we present the case and the framework, we will review the work of others in the area of modelling methods, reuse-oriented software design and problem orientation. In the remaining sections, we will show what elements from these different approaches we combined to create a solution optimized for the plant-modelling problem.

This paper is the extended version of the paper (Marincic *et al.*, 2010) in which we briefly presented the action case in which we analyzed modelling as a leap from a non-formal world to the formal one. In this paper, we will review the work of others that deals with modelling and will discuss to what extent they address methods focused on the reuse of plant description for the purpose of software verification. We will also present our action case in more detail, and will extend the analysis of reusable modelling decisions.

2. Related work

2.1. Object-oriented method

One widely accepted modelling and design approach is object orientation (OO). Besides languages and tools, it offers methodological support for modelling. The methodology provides guidelines for dealing with complexity and recognizing different structures during classification and abstraction (Booch *et al.*, 2007). There are many examples that illustrate good modelling practices within OO. Furthermore, phases in the modelling process are distinguished, such as analysis, system design and object design; general guidelines for structuring the process further are given (Rumbaugh *et al.*, 1991). These steps are not presented in the form of a cookbook, given that 'the design of [complex] systems involves an incremental and iterative process' (Booch *et al.*, 2007). The process itself is supported by the Unified Modelling Language (UML - Unified Modeling Language, 2011).

Some of the elements of the OO approach can be used as methodological support for methods other than the OO method. However, the steps and best practices of the OO methodology are focused on the *software*, not the plant modelling.

The OO approach is designed to be reusable. After all, the idea of design patterns in software engineering was conceived as an OO programming method. Patterns document design solutions for recurring problems in software design. A pattern describes a problem and its context, the elements of the solution and the trade-offs between applying different patterns (Gamma *et al.*, 1995). In their collection of common software design problems, Coad *et al.* (1995) present patterns together with problem solving strategies. Design patterns are not only solutions that need to be instantiated, they also explicitly represent (a part of) experts experience.

Some parts of the modelling knowledge and some of the model elements extracted for reuse in modelling the next system generation may be represented as patterns. However, making decisions on what to reuse and how to extract reusable model elements for the next system generation is not supported by design patterns and pattern languages.

2.1.1. SysML As a software design and modelling tool, UML has its limitations when it comes to modelling the components that surround software. The OMG's Systems Modeling Language (SysML - Open Source Specification Project, 2011) overcomes these limitations. SysML introduced new diagrams, modified some of the existing ones and abandoned the diagrams that are useful for software but not system modelling. As UML's extension, SysML is based on the OO concept, and it can be used in combination with other approaches.

Researchers in the simulation modelling community see the future role of SysML similar to the role of UML models in software design. Huang *et al.* (2007) propose using SysML as a language for developing a system description that will serve as basis for the automatic generation of simulation models. Paredis and Johnson (2008) recognize SysML as a framework to integrate different domain-specific models; they developed a graph-transformational approach to translate SysML models into corresponding simulation models. However, SysML offers no methodological guidance for system analysis; its role is to serve as a precise notation for system engineers.

2.2. Reuse-based software engineering

Apart from design patterns there other techniques have been developed for software reuse. A software reuse technique that aims to increase development efficiency in the long term is product-line engineering. It exploits commonalities between the related set of products, investing initially in defining what these commonalities are (Mili *et al.*, 2002). An important

concept in this approach is variability – it addresses the differences between similar products in an established common platform (Pohl *et al.*, 2005).

Product-line engineering addresses software reuse, not the plant model reuse. Also, it requires commitment of the organization to reuse from the very start of the development. Commonalities of the systems, their software, requirements specifications, design decisions, test data and architectures are established in advance. When looking at reusability of the plant models we do not assume this commitment, but we focus on extracting knowledge from an already existing system and reusing it in the future.

Both practice and research acknowledge the importance of organizational and economical, besides technical factors (Mili *et al.*, 2002). We, too, take into account that a wide solution space for a modelling problem is restricted by pragmatic constraints, some of which come from organizational factors. However, we focus on a very small part of these factors that influence directly modelling decisions.

2.3. Capturing, reusing and managing architectural knowledge

Members of the software architecture community recognize architectural knowledge as a valuable asset for building and maintaining quality systems. They have been developing methods for preserving knowledge and for capturing the rationale behind architecture design decisions.

Architecture design decisions and plant modelling both consist of formal and non-formal steps; also, in both of the activities, some of the decisions and part of the knowledge that the designers accumulate stay implicit. Therefore, the insights on types of architectural knowledge (Kruchten *et al.*, 2006) are a useful starting point for our own analysis of practical issues of plant modelling knowledge reuse.

If we would replace the word 'architecture' with the phrase 'plant model' in the literature on architecture decisions, we would get questions and problems relevant for plant modelling. However, a thorough analysis of types of architecture decisions, like, for example, the ontology proposed by Kruchten (2004), reveals the concepts and issues different than those in the plant modelling. This is because the perspective of architecture decisions differs too much from plant modelling decisions. Therefore, we cannot take over the solutions offered by the architecture community.

2.4. Design rationale

Design rationale is 'justification behind decisions' (Dutoit *et al.*, 2006). The techniques to elicit and document the design rationale offer argumentation structure, categorizations and ontologies. The arguments in one of the approaches, IBIS (Issue-Based Information Systems), address issues and arguments, pros and cons of positions. QOC (Questions, Options, and Criteria) addresses, as its name suggests, questions, options and criteria, but also assessments, arguments and decisions (Dutoit *et al.*, 2006). In the area of system and software architecture, ontologies are proposed (Kruchten, 2004). Our focus is slightly different; we look into what kind of concrete modelling decisions need to be explained. For example, if we cannot map the model components to the system components, i.e. if there is no isomorphism relationship between a model and system component, we recommend to explain why the two structures are equivalent regarding the property and behavior of interest.

2.5. Problem orientation

The problem frames technique (Jackson, 2000) offers a framework to analyze the system requirement and the plant while designing software specification. The modelling problems we are looking at all fall into the category of a 'required behaviour' problem frame. The first steps in decomposing the problem into domains coincide with the system decomposition we look at in our work, but we are focused on modelling steps, rather than on requirements analysis for software design.

Colombo *et al.* (2007) propose and evaluate using SysML as a language for the problem frames technique for requirement analysis. The problem frames technique compensates SysML's lack of methodological support for domain description. At the same time, SysML offers the adequate linguistic support that the problem frames technique misses. This approach structures domain modelling and problem analysis for the software solution design. Given that it combines SysML and problem frames, this approach could offer a notation for the elements we identify, but the methodological support we offer is of a slightly different scope (plant modelling) than the one for the problem frames (plant and requirements analysis for software design).

Based on the problem-oriented perspective, Hall *et al.* (2008) developed a problem-oriented approach (POE) for software development. They address the problem that Turski (1986) described as a gap between software (a formal entity, for which we can derive a correctness proof) and the application domain (the non-formal world, which cannot necessarily be described in a single linguistic system and cannot have its properties mathematically proven). This, in essence, is the same problem we start from, the only difference is that we deal with models, not software.

The POE approach offers conceptual framework based on the problem frames technique. It classifies software design activities and relevant entities. The conceptual framework is formally described to provide clarity and preciseness, but it also represents domain descriptions and the outcome of design activities in a language or notation (Hall *et al.*, 2008). This way, both non-formal and formal steps of software design are supported with a formal framework which defines design steps such as different types of problem transformations and elements of the problem domain.

The POE framework accommodates recording problem progression in a tree-like structure. Decisions that lead to the solution as well as dead-ends can be recorded, in formal or informal language or notation. The framework can be used for justification of design decisions, as a design-rationale-storing tool (Nkwocha *et al.*, 2010). Another application is in safety-oriented software engineering, where it can structure building of an assurance case simultaneously with design activities (Hall *et al.*, 2007).

Our framework is non-formal and it does not support recording progression from the problem to the solution; instead it structures the analysis after a model has been constructed. Also, even though software design and software

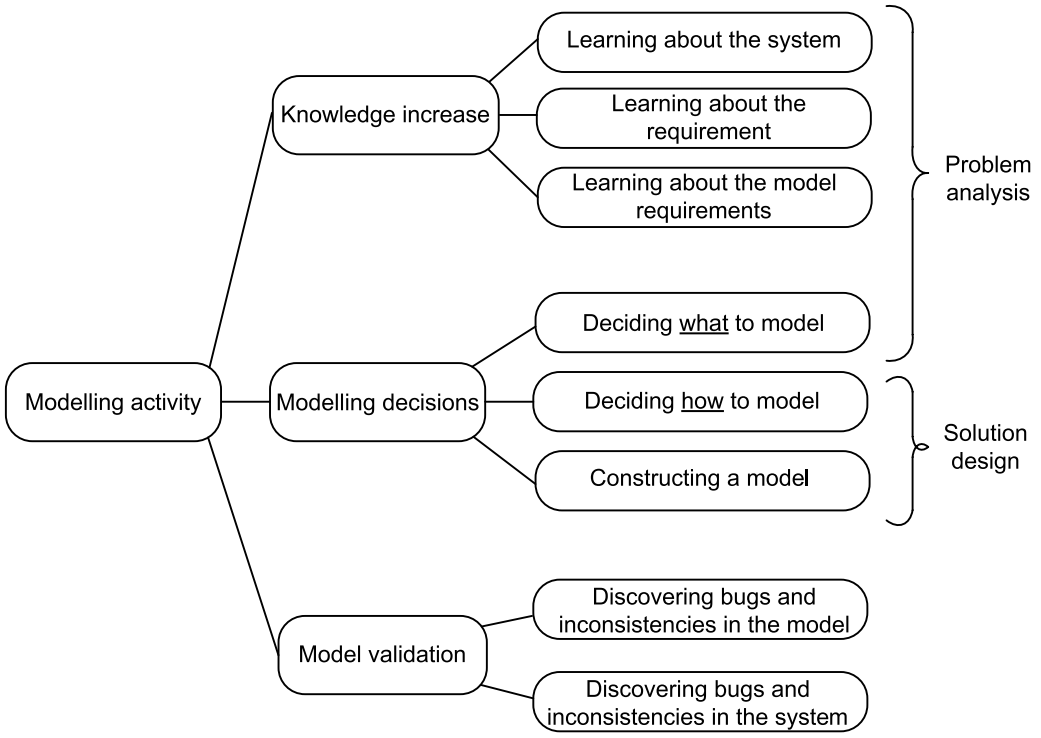


Figure 1: *A classification of modelling activity to learning, modelling and model validation activity.*

modelling have decisions that overlap, we focus only on modelling of a plant.

3. A framework for reusing modelling elements

To explore what kind of non-formal modelling decisions are reusable, we first have to define what non-formal modelling decisions are. Therefore, before we present our framework, we will discuss non-formal aspects of modelling process.

3.1. Modelling process

Modelling is an intellectual process that allows a modeller to understand a system, moving from the stage of exploring a problem in general to designing the solution. During problem analysis, knowledge about the plant and the modelling problem is acquired. At this stage, the modeller has already created her own decompositions of the system, abstractions and initial ideas about the model. These derive both from the information provided by domain experts and the modeller's previous experience, education and personal preferences. The modeller does not model the system, *per se*, but rather the knowledge acquired about the system.

During the process of learning, modellers first recognize known problems for which they know a solution (Cross, 2001). If the problem cannot be matched to known solutions, the solution space must be further explored. The problem can be decomposed in many ways into known problems, and many solutions to known problems can be discovered.

Figure 1 shows the classification of activities in modelling. It contains the same elements as Hall and Rapanotti's (2008) design cycle. While Hall and Rapanotti (2008) place an order on these activities, we examine and further classify each process separately, focusing exclusively on modelling that is required for software testing and verification.

In a previous work, two authors (Mader and Marincic) classified the modelling decisions identified in Figure 1 in the form of questions that a modeller must answer while designing a model (Mader *et al.*, 2008). These questions identify classes of non-formal modelling decisions in form of the following questions:

What is the purpose of the model?
What is the quality criteria for the model?
What is the object of modelling?
What is the structure being modelled?
What is the mathematical domain of the model?
What idealizations and simplifications are applied?
What are the pragmatic factors that influence modelling decisions?

3.2. Reusable modelling decisions

We used the classifications described in sub-section 3.1 to analyze a concrete model and to establish what modelling decisions can be reused. Based on this analysis, we propose a general framework for reusability analysis of modelling decisions. It can be seen as a method to extract reusable modelling decisions, knowledge and experience. The framework elements are as follows:

Reusable modelling steps and knowledge
- Reusable decompositions
  - System decomposition(s) represented in a model
  - Model decomposition
- Reusable solution specifications
- Reusable knowledge about modelling plant processes

Requirements for the model

- The system requirements to analyze with the model
- The purpose of the model
- Pragmatic limitations

Reusable modelling strategies

- Model elements that do not follow the structure
- Model elements that appear only in the model and not in the system
- Dependence on the software's behaviour
- Modelling assumptions

Modelling Experience

- Alternative modelling decisions
- Solutions that did not work

Model stakeholders

3.2.1. Reusable modelling steps and knowledge When deciding what modelling decisions and knowledge to extract from the current model, the modeller does not know how the next system generation will look like. It is the modeller's subjective estimation, based on the information received from the model stakeholders, what modelling decisions are worth the effort of preserving for future reuse.

If the system component, modelling language and the model requirements do not change, the part of the model describing the component can be directly reused in a future model. But, if any of the above elements change, the model component can no longer be directly reused. However, this does not preclude reusing some modelling decisions. They are as follows: the structures assigned to the system and the model (decompositions), solution patterns and the knowledge about the processes performed by components.

Reusable decompositions. In textbooks, we find examples of neatly organized diagrams representing physical components, functions, processes, etc. In practice, many decompositions exist at the same time without having one of them established as *the* decomposition that everyone in the company refers to. System decompositions in an organization are reflected in the directory tree of a shared project repository, roles assigned to engineers in different teams that they form and in the different abstraction levels they use when designing or analyzing a system. Often, a diagram mixes different decompositions as well as different abstraction levels.

The modeller can assign her own structures or adopt existing ones. Often, the model structure follows the system decomposition. Even if the individual components change, the high-level system and model decompositions may be reusable. If the modelling language allows for hierarchical structures, then the model can follow both the system hierarchy and the decompositions on different hierarchy levels.

Reusable solution specifications. For recurring modelling problems, solutions can be saved using generic, language-independent solutions. Many modelling problems for mechatronic systems are solved using a state-based paradigm. If a modelling language changes in the future, but not the state-based paradigm, then the statecharts can be used to document the solution. The modeller should bear in mind that different state-based languages have different semantics (Harel & Naamad, 1996).

Reusable knowledge about modelling the plant processes. Many commercially produced mechatronic systems rely on changes in the components to stay competitive in the market. Mechanical components are designed to be smaller, or of a different shape and layout. In cases of systems that manipulate products, often the processes that these components perform stay the same, because they are a part of the overall system requirements that define how the products should be manipulated. Knowledge about the processes, as well as the general knowledge about how components interact may be relevant in the future system generations.

3.2.2. Requirements for the model What parts of the system will be modelled and how, is determined by the following factors: the requirements of the system that are examined or analyzed with the model, the purpose that the model has to fulfill, the quality requirements for the model and pragmatic limitations that the stakeholders impose.

System requirements to examine/analyze with the model. In formal verification, the system requirements are defined explicitly (and formally). When using other modelling techniques, the requirement is not formally defined; the modeller or the model users examine whether the model behaves as expected and whether it possesses certain properties. When designing plant simulation models for testing purposes, verifiers define non-formally what requirements they want to test. In all these cases, the requirement usually refers to the plant elements that are not directly connected to the software. It is the modeller's task to find causal relationships between the signals on sensors and actuators and the plant elements behaviour that leads to the desired behaviour for the plant elements specified with the requirement.

Purpose. The obvious purpose of a verification model is to examine whether the system satisfies its requirements. But, the modeller also must examine in a larger context the purpose of the model to determine what parts of the software and the plant to model and with what degree of accuracy. If the model's purpose is to show that the requirement is fulfilled even when certain faults are present, than we have models made for fault tolerance requirements. If the model is designed to support testing, like the model from our case study, then care has to be taken that the plant model that has to be designed is at least somewhat independent from the software's behaviour.

Sometimes a model used for one purpose can be easily adapted or used for another purpose. Knowing the model's purpose prevents false assumptions that a model adequate for one purpose is automatically adequate for another one.

Pragmatic limitations. Pragmatic limitations such as hardware or software resources, processing power, time or knowledge available in the company, narrow down the solution space. Some of these limitations can lead to suboptimal modelling decisions. We propose to document them modelling

decisions so that the model can be improved in the future if some of the limiting factors change.

3.2.3. Reusable modelling strategies Modelling strategies provide a rationale for certain classes of modelling decisions. The classes of modelling decisions that we recommend to explain are as follows:

Model components that do not follow the system structure. A model represents its target system adequately if it shares the relevant properties with the system or, we can say, if it mimics the system. This is achieved by having some kind of similarity between the model and the system. It can be an isomorphism, partial isomorphism or homomorphism between system and model components. In practice, model structure can diverge from the identified system structure. Also, there can be additional elements in the model that cannot be traced back to any of the system components. An additional argument for the model's adequacy may be necessary to explain the modeller's decisions to the model stakeholders.

Model elements that are adequate under certain assumptions about the software. Ideally, the model of a plant would show the plant independent of the software, with all its possible behaviours. However, in practice, this would make the model too complex. So, compromises are made, and the model is designed with many assumptions about the software behaviour. For future reuse, these assumptions should be recorded: what seems to be an obvious assumption in the present may not be in the future. Also, while some assumptions may be obvious within any given organization, this internalized cultural knowledge will be lost if the modelling task is given to a new person or outsourced to another organization.

Model elements that are adequate based on the assumptions about the other parts of the system. Assumptions are made not only on the software, but also on the behaviour of the surrounding components and the system environment. Some of these conditions should be recorded and checked to see if they still hold for the components of the next system generation.

Model elements that are not optimal. These elements may have been taken over from an old model that had been made for a different purpose, or for the requirements that were dropped. Some of the modelling solutions may be simpler or better, so we propose to evaluate the solutions that derive from the old requirements.

3.2.4. Reusable modelling experience

Solutions that did not work. Documenting some of the 'dead-ends' in the search for the solution for the modelling problem can save time in the future. A modeler must subjectively decide which of the 'dead-ends' are worth documenting, though those that are not obvious deserve special attention.

Alternative modelling decisions. These are the decisions that were considered but later abandoned. Different modelling choices mean that some trade-offs are made, so recording what are the advantages and disadvantages of these alternative modelling solutions would save time when the next generation system is modelled.

3.2.5. Model stakeholders Assessing a model for future reuse requires understanding the modelling problem, modelling decisions and the pragmatic constraints on the model. Moreover, it is necessary to get the domain experts' estimation of changes in future system generations. To gather all the necessary information, the analyst will have to talk to all the stakeholders of the model. The modeller inside an organization most probably knows who the stakeholders are, but for an external modeller this is an important piece of knowledge.

In Section 4, we will show what these decisions are using a concrete example.

4. Case study

4.1. The goal of the case study and its relevance for our research questions

The goal of the case study was to find out what non-formal modelling decisions can be re-used. We started with our high-level classification of modelling decisions and used it in the model analysis. We adapted the elements of our classification into a framework for capturing re-usable modelling decisions.

Our task was to analyze a plant model made for a system that was under development at a time. More precisely, our task was to, based on the existing model, design a handbook that would assist modelling the plant for the next system generation. To gather the necessary data, we interviewed the designers of the plant model and the model's stakeholders: the project manager, the head of the software department, mechanical engineers, the system integrator and test engineers. To elicit the modellers' knowledge, we asked for the rationale of their modelling decisions as well as the history of the model development. We also had a hands-on experience with the model.

We performed the case study in a company (Neopost-Technologies, 2011) that develops, produces and distributes different types of mailroom equipment and document systems. The users of document systems are companies that send a lot of paper mail on a daily basis, such as insurance companies, post offices, and banks. One such machine, the inserter, is shown in Figure 2. The inserter automatically folds paper sheets and inserts them into envelopes.

The inserter is a typical example of a system whose mechanical elements' connection and synchronization are relevant for the software verification. Our research is aimed at these types of the machines, and the case studies we performed include a lab-made sorter of blocks and a printer. These are all the devices manipulating products (papers, blocks), highly modularized, controlled by real-time embedded software. Also, both the printer and the inserter are designed and produced in large companies that have

Figure 2: *The inserter folds paper sheets and inserts them in envelopes.*

subsidiaries in different countries, and that have large software development departments.

The company in our case has a matrix organizational structure. There is a division to the departments responsible for mechanical, electrical and software parts, respectively, as well as the system engineering department. Orthogonal to this division, teams are assigned to different projects. The head of the project is responsible for managing different product development phases, whereas the head of a department is responsible for providing relevant expertise to different projects.

One of the emergent benefits of the simulation model we analyzed is that it brought together different domain experts in the early phases of the development. Their communication is crucial in these early stages, and shortens the time to integrate components once they are designed and produced. Compared to integration time of previous systems, where a simulation model did not exist, integration time of a current system was significantly reduced which saved additional costs and prevented significant delays.

4.2. System description

The inserter works as follows. The operator places documents (paper sheets) and envelopes in appropriate feeders. Through the user interface, he specifies options (recipes) for manipulating documents. A recipe defines how many documents will be inserted in each envelope, whether the paper sheets will be folded, and if so, how; a document can be folded in half or into thirds (in Z or C shape). An example of a recipe is: 'Select three documents from the first feeder, fold them in half and insert them into an envelope'. Recipes are combined into scenarios, which automate further the inserter's work.

The diagram in Figure 3 shows the inserter's modules. Each module has its own function, which is to perform a certain process on a piece of paper or an envelope. The arrows in the diagram represent the flow of the documents and envelopes through the system. In the feeder, rollers (small plastic wheels) pull out the first document on the pile. The rollers and a transport belt move the document forward and bring it to the collator. The collator collects the documents that belong to the same envelope and collates them. After that, the documents are moved to the folding position. A stroke of a long, thin, sharp-edged arm folds the documents. In the meantime, the rollers of the envelope feeder pull out the top envelope. The rollers along the envelope transportation

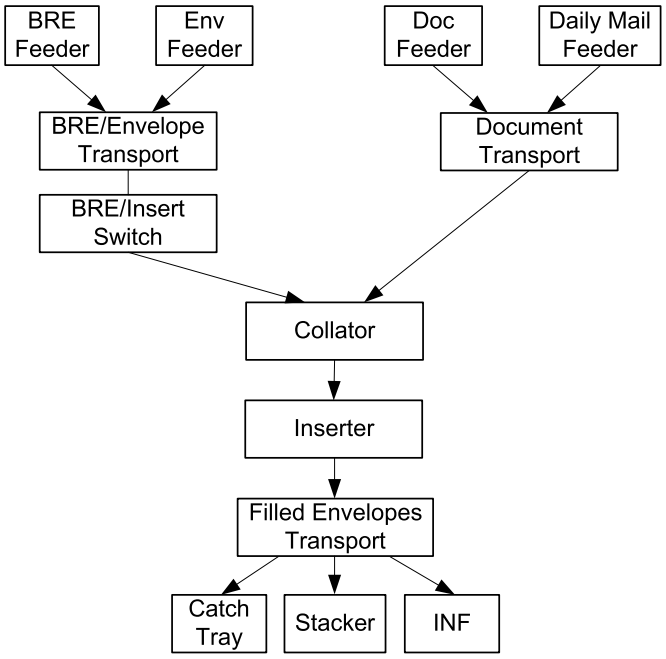


Figure 3: *Diagram of the mechanical plant parts and document and envelope paths. (BRE stands for Business Reply Envelope feeder.)*

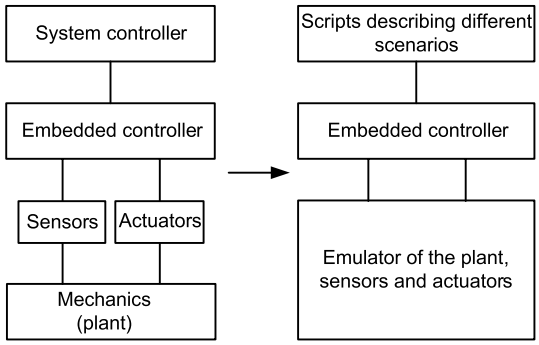


Figure 4: *Inserter's high-level architecture and the concept of the emulator.*

path bring the envelope to the flap moistener. The flap is first turned up and then moistened with a stroke of a brush. When the envelope takes the position in the inserting module, its upper side is slightly lifted, to facilitate document insertion. Folded documents are then inserted in the envelope, the flap is closed and the rollers move the envelope with documents towards the exit.

Every process performed on documents and envelopes is the result of controlled movements of mechatronic parts. Rollers, folding arm, brush and many other system parts are connected to actuators that move them. Along the system, sensors are placed to signal the presence or absence of material in front of them. Based on sensor readings, the control software sends signals to actuators.

The high-level system architecture is shown on the left side of Figure 4. The System Controller executes operator requests and recipes, forwarding them to the Embedded Controller. The Embedded Controller controls the plant.

4.3. Plant and control integration

When a new generation of the inserter is designed, some mechanical parts, or even whole modules, are re-used from the old inserter generation. Some parts, conversely, are completely re-designed. Radical design results in significant improvements of functionality or performance, which is important for competitiveness in the market. At the same time, it brings unanticipated problems and issues (Vincenti, 1990). Some problems are related to the integration of the control software and the plant. If the integration comes at a later stage of the project, solving these problems might need parts, modules or even concepts to be reworked, which causes delays.

The plant and the controller together deliver the inserter's desired behaviour. It is therefore important to enable their concurrent development. When the control software and plant are designed at the same time, mechanical and software engineers communicate during early development stage, and problems related to plant and control software integration arise early enough to be resolved on time.

The problem is that in the early development stages the plant exists only in sketches and computer-aided design (CAD) drawings. Interaction with the plant via sensors and actuators is the essence of the control software; therefore, without the plant, control software cannot be tested.

4.4. The plant model

To enable concurrent engineering of the plant and the control software from early development phases, the company designed the plant simulator (emulator) to test the control software. Figure 4 shows the comparison of the inserter architecture and the architecture of the testing setup. The emulator ($M1$) is a digital signal model of the plant, sensors and actuators (P). Digital signals represent plant processes, such as document transport and paper folding, and the behaviour of the sensors that detect documents. The signals that emulate plant processes are functions of the signals that the software sends to the actuators. The signals that emulate the sensor behaviour are functions of the signals that emulate plant processes. These functions are specified with LabView (LabView, 2011) diagrams (Φ) and they define the emulator behaviour.

We analyzed the LabView model. The programmable hardware (emulator) is just the execution of the specification, in a way similar to the computer hardware's execution of a computer program. The model could have been just as well implemented as a simulator, describing the plant with software; for our research question this would not make any difference.

To specify the digital circuits behaviour, a high-level specification language, called G-language, is used, in LabView tool. This diagram is a sort of data flow diagram. The LabView specification of the emulator specifies the behaviour of digital circuits. At the same time, it represents the flow of papers and envelopes through the inserter, movement of different physical parts and sensors' reaction to papers and envelopes. LabView compiles the diagram-based specification to the program for the hardware. We consider this transformation correct, and only address the graphical model of the system.

LabView allows to specify models using state machines, via its state machine pattern. The plant modeller did not use it, but designed his own state machine pattern, specialized for the problem in hand. The modeller first informally specified state machines (the model components) and then translated them to LabView diagrams. (Appendix A shows the feeder state machine and LabView diagram).

Designing the plant for the first time took significant time and effort. The modeller spent a lot of time learning about the plant, trying out different modelling solutions and finding the optimal one. He did not document his insights and knowledge progression, due to time-pressure. The model-based testing improved communication between departments and shortened the integration time, so it will be used in the future.

How the new generation of inserters will look like, when it will be developed, which parts will be incrementally and which radically designed, is not known yet. The modeller will most probably be another software architect either from inside or outside the company. Finally, the tools and languages used now, may not be the choice next time.

The handbook we will present supports modelling of the next inserter generation. It is based on the analysis of the existing inserter model and the estimation of the changes in the next system generation. As such, the handbook is useful for a concrete case of inserter modelling, or a family of inserters. Possibly, the handbook is useful for a state-based modelling of similar machines, such as printers. But, some of the printers' components and processes are completely different than an inserter's components and maybe modelling them has its own characteristics not covered by the handbook.

Our aim, however, is not to deliver a universal modelling handbook, but to show how the framework we described in Section 3 can be used as a method to design a handbook for modelling a highly decomposable mechatronic system.

The inserter modelling handbook proposes the current model elements suitable for re-use, part of the knowledge and experience gathered during designing the current model and principles followed in modelling the current model. We represented the re-usable model elements with state machines, as this is the modelling paradigm followed in the company.

4.5. The handbook – reusable modelling steps and knowledge

4.5.1. High-level system and model decomposition The inserter simulation model has to follow the system structure. It is often the case that a structure of a model follows a modelled system decomposition. If the modelling language allows for hierarchical structures, then the model can follow the system hierarchy as well as the decompositions on different hierarchy levels.

The diagram in Figure 3 is one of many diagrams (decompositions) used in communication between engineers in the company. The feeder refers to a physical component in which papers are initially stored, but when talking about a collator, it is seen as collating function in one occasion, and as a set of mechanical parts in the other. In the same diagram, there is a block called 'Transport', which refers to the row of rollers that move documents or envelopes, from the feeder to the collator. But, the rollers that move documents between the collator and inserter, between the folder and inserter, etc. are

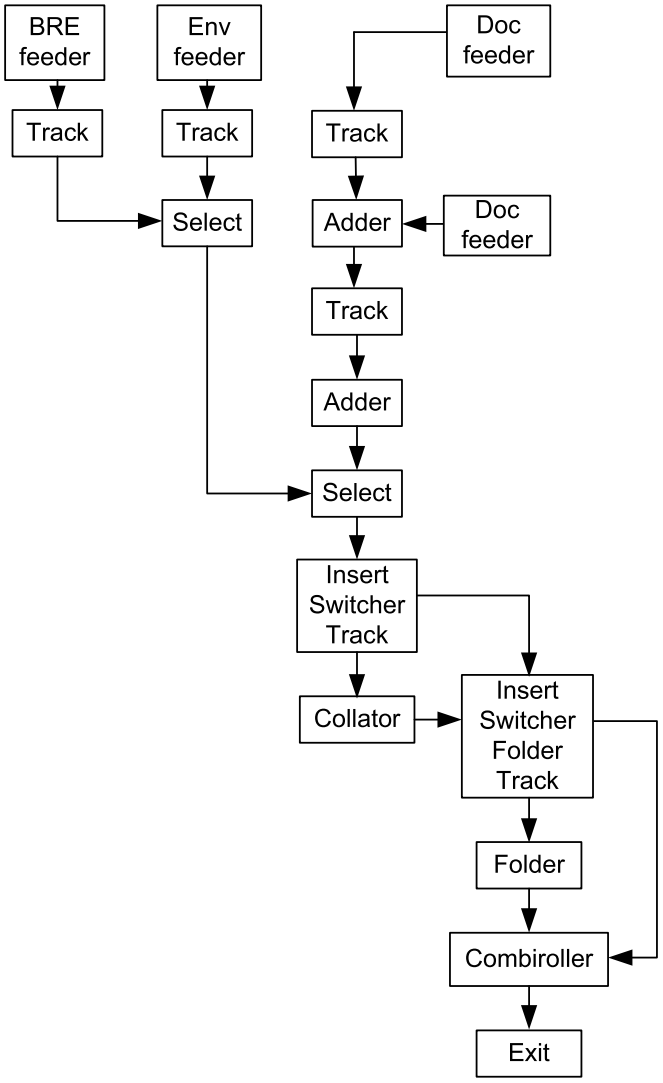


Figure 5: *Diagram of the mechanical plant parts and document and envelope paths in the model.*

considered to be elements of a lower abstraction level and therefore not explicitly represented in the diagram.

The plant modeller had to find a suitable system decomposition to be represented in the model, and it is shown in Figure 5. The transport path in this high-level model decomposition stops at the collator. In the actual inserter, the transport path continues all the way to the exit, but in the model its remaining parts are encapsulated in the collator, inserter and folder. How the papers are manipulated within these four components may change in the future, whereas the part of the path, from the feeders to the selector is unlikely to change. Therefore, the high-level structure of the model is the recommended structure for the future models.

4.5.2. Reusable solution specifications – paper transport and paper path The paper path is the least likely component to change. The current model of the path is designed in LabView, but it is based on the state machines that the modeller first informally designed and then translated them into the data flow (LabView) model. We chose to document these state machines (and not the Lab-View diagrams) and present them as reusable modelling solutions, for two reasons. First, LabView may not be the modelling language when the next

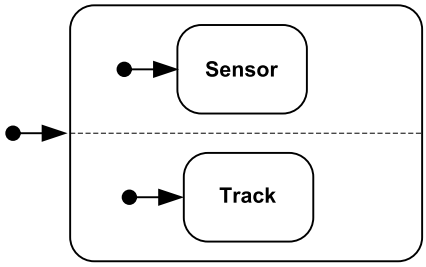


Figure 6: *A segment of the document path contains a sensor at the beginning of the segment.*

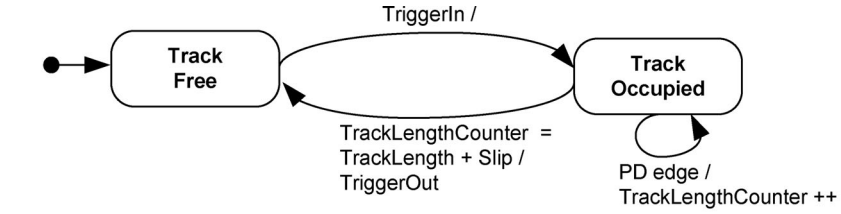
inserter generation is modelled. Second, the state-based approach is well adopted in the company and both mechanical and software engineers use them in communication about certain system aspects.

The path (model) consists of the following components:

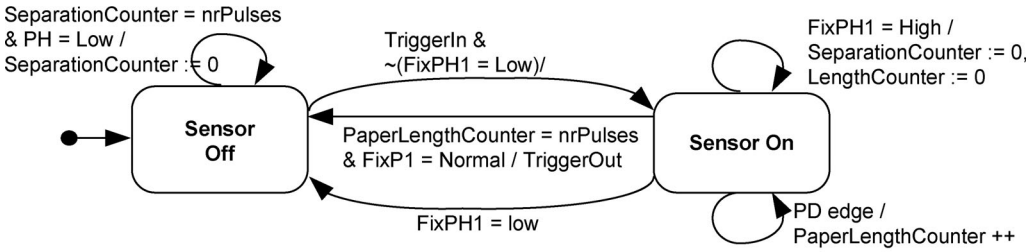
- Track models a piece of the paper path not longer than the length of the paper.
  - Simple track
  - Track with a slip – models slip of the rollers — rollers are rolling, but the paper is not moving
  - Feeder – this is a special type of a track, that on trigger reads the parameters of the model user. These parameters define the length and thickness of the paper in the feeder.
- Sensor – every track begins at the spot where the sensor is. The sensor senses the presence of the material in front of it. In the model, if the sensor breaks, it stays blocked in its current state. Depending on what the state is, it continues to represent either presence or absence of the material in front of it, no matter whether actually there is something in front of it or not (it stays stuck showing the last correct value).
- Adder – paper moving from one feeder will be joined by the paper coming out from another feeder. The way they join assumes that the software will make sure that they overlap on the two-thirds of their length or more.
- Switch – models the point on the path where there are two possible ways to take. A switch works like a railway switch – it moves the track towards one path or the other.
- Selector – selector is the place where two tracks merge into one.

Together with every diagram, we provided the vocabulary that defines what each state means, event, action and variable represents. For example, the variable 'length' refers to the length of a single paper or two papers joined together, or it is an envelope length. We also documented a rationale for some of the design decisions and modelling assumptions made while modelling. For example, one of the assumptions is the minimal length of an overlap between two A4 papers coming from the two feeders. Some of the diagrams are shown in Figures 6, 7(a), 7(b), 8 and 9.

Every state machine came with a short description of the system components and aspects they represent. Also, we documented properties of documents (papers, envelopes, etc.) that are relevant for the plant modelling. These include thicknesses, lengths, types of envelops, etc.



(a) Track state machine



(b) Sensor state machine

Figure 7: *State machines of a sensor and a track combined together to represent a path segment.*

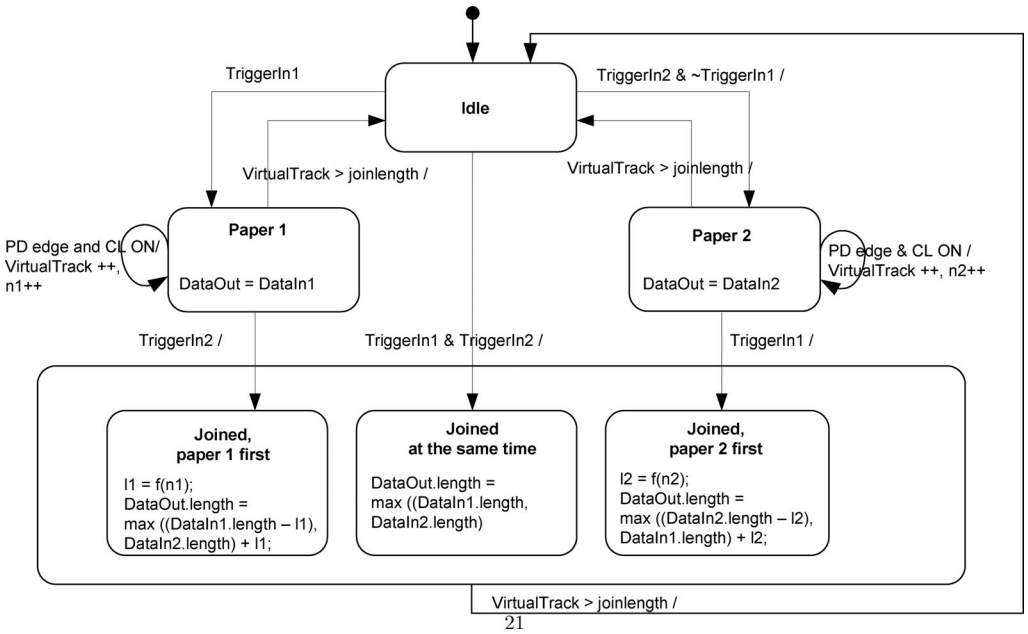


Figure 8: *Adder represents the part of the document segment path where two documents merge.*

Adding designations and definitions is important, as it makes explicit what the model represents (Jackson, 1995). Without them, different modellers could interpret the model differently.

4.5.3. Reusable knowledge about modelling the plant processes The paper path will most likely stay the same, but the collator, folder and inserter components may change. In fact, it is desirable that some of these components change to make the inserter smaller, faster or of a different shape. Creative solutions and radical changes introduced by mechanical engineers into new generations of inserters are essential for their competitiveness in the market.

If the physical components change, the high-level processes that they perform will stay the same – inserting papers into envelopes consist of collating, folding, inserting, flap moisturising, regardless the changes of the physical parts that execute these processes. The sub-processes, however, may change, given that the high-level processes are executed by different physical parts. Therefore, we did not generalize the model components that describe the inserter's physical components and sub-processes, but we extracted the questions that the modeller had to answer when modelling them. These questions mark the modeller's search for the solution. They are shown in Figure 10.

4.6. The handbook – requirements for the model

We recorded the main pragmatic factors that limited the solution space. They influenced modelling decisions significantly, and it might be that in the future, some of these conditions could no longer be present.

For example, the modelling language was chosen for the following reasons: engineers were already familiar with it, it provided visualization, and graphical modelling kept the focus on the plant (C-like simulation language made software engineers think in terms of the software, not the plant.)

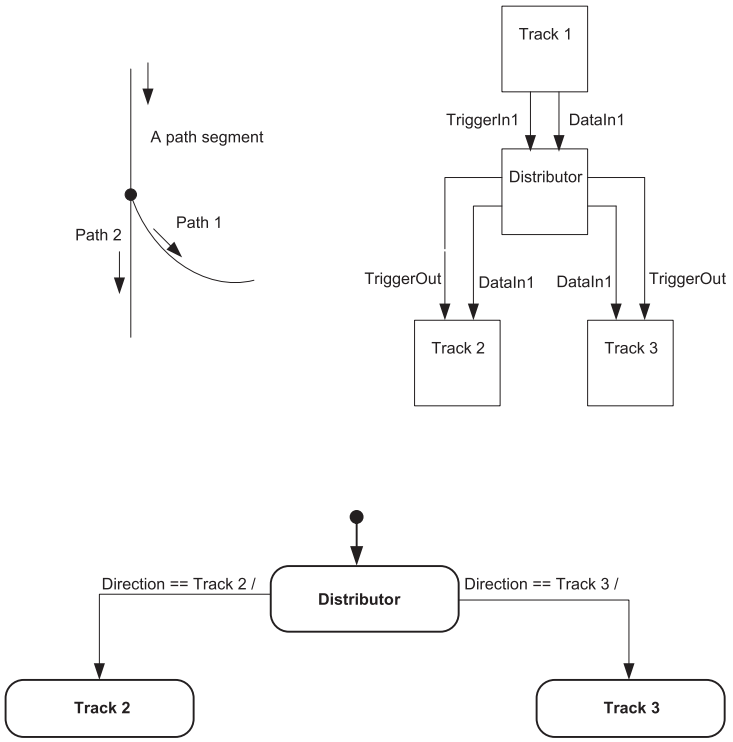


Figure 9: *A part of the path where it splits into two different paths. The state machine describing the ditributor's function is part of the state machine describing movement along the paper and envelope path.*

Questions to guide learning about new system parts
Functional and physical decomposition What is the system functional decomposition? Which physical part performs which function? Does one mechanical part perform two functions at the same time?
Components and sub-components positions upon arrival and leave of a document Is the initial and end position of a component relevant? If so, what is it? What are other relevant properties of each physical component upon document arrival and leave? How is a document moved from one component to another? Is it a joint activity of two components, or only one of them?
Speeds and speed ratios Is the speed ratio of two mechanical components relevant? Is the component speed relevant and is it constant?
Timing Can certain movements be modelled as zero-time events?
Clutches Is each component moved by a different clutch, or one clutch moves more than one component?
Software dependence Are there sub-functions performed without assistance of the software? What are the relevant phenomena that need to be described, but software cannot observe them?

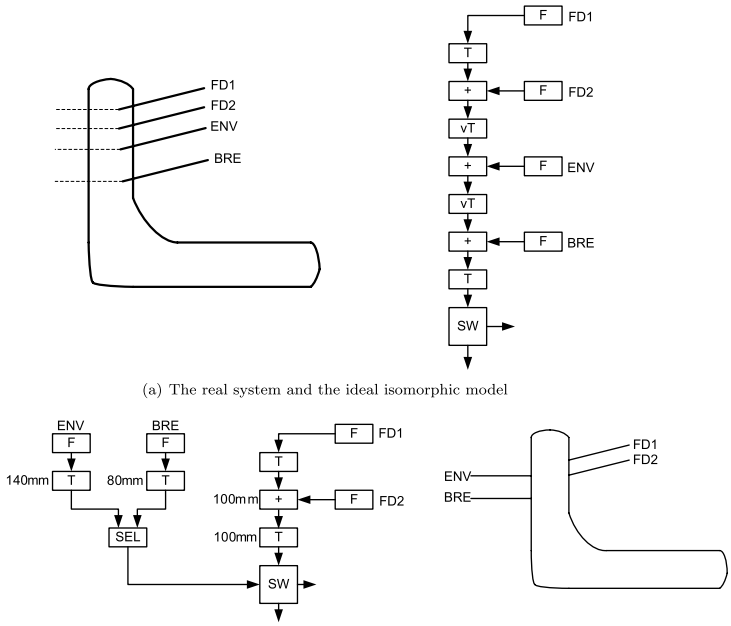
Figure 10: *Questions to answer when learning about the new solutions in the inserter design.*

Another important factor were limited hardware resources which resulted in a number of sub-optimal solutions. We recorded them, as in the future the modeller will have more resources at hand.

4.7. The handbook – reusable modelling strategies

We documented creative solutions and modelling 'tricks' the modeller invented to prevent the model becoming too complex. The complexity was determined by two factors. The first was the number of hardware circuits that implemented the emulator specification. The requirement was to design an emulator that would not require purchase of additional hardware elements. The second factor was the cognitive complexity. It was the modeller's subjective assessment of how complex the model can become and still remain intellectually manageable.

Model elements that do not follow the structure. The starting idea was to design a model that follows the structure



(a) The real system and the ideal isomorphic model

(b) The real model and the system that the model would represent if it were isomorphic

Figure 11: *Diverging from isomorphic structure to save hardware elements.*

of the system as closely as possible. This way the design decisions can be justified by the isomorphism between the model and the system. However, sometimes due to the language or other pragmatic constraints, it was not possible to achieve this. We looked for these model elements because they required creativeness to solve problems and to model the system differently. We also recorded the justification of these decisions, an argument how a model component that does not follow the 'ideal' structure still represents the system correctly.

An example of such deviation is the layout of the feeders in the model. The distribution of the feeders in the real system is shown on the left side of the Figure 11(a). The natural way to design the solution would be the structure shown the right side of the Figure 11(a). This model would follow the layout of the feeders but, due to very short distances between papers, virtual tracks would have to be introduced in the model, so that the movement of each paper can be described. Isomorphism would not be achieved and these virtual tracks would require usage of additional hardware elements.

Instead, the structure shown in Figure 11(b) is used. This structure suggests the layout of the feeders shown on the right side of the Figure 11(b), which is not the structure of the feeders in the modelled inserter. However, given that the overlap between an envelope and a paper sheet is not possible (the control does not allow it), the whole component adequately describes the movement of the documents.

Model elements that appear only in the model and not in the system. Some of the model elements do not represent any physical part or a process in the system. Still, they are introduced in the model for various reasons. Sometimes a modelling language has constraints that require adding additional elements, or the system property needs to be expressed in terms of the model components that do not have their match in the system. Inventing these solutions requires creativity.

In the emulator specification, one such element is a sensor. Every segment in the model has a sensor placed at the beginning of the track, whereas in the plant, sensors are not positioned at all these places. But, for modelling the document transport with the desired accuracy, placing 'sensors' at the beginning of each and every segment was necessary.

System elements not present in the model. A model is an abstraction of a system, which means that many system elements will not be described with the model. For example, in the emulator, the component that moistens envelope flaps is not represented, as this process is controlled mechanically, and therefore invisible to the control software.

A special case of a system element that is not present in the model is the motor. It is present in the model, but 'in person' rather than being modelled. Modelling the motor would require too many hardware elements, so it was cheaper to place the actual motor at the modelling setup. In the modelling setup, the motor does not have any load, so its current is scaled down.

Dependence on software behaviour. Ideally, a plant model is independent from the software specification, but in practice this would result in a too complex, too big, and incomprehensible model. What assumptions on the software to make to simplify the plant model is the modeller's decision.

In the case of the emulator, the modeller used the fact that the control forces the feeders' rollers to move the papers in such a way that the papers either overlap at least two-thirds of their length or do not overlap at all. Without these assumptions, the paper path would have to be described with many more elements, and this would require more hardware, and possibly would result in a model too complex to grasp.

Stakeholders	Goals (G) and Constraints (C)
Head of System Engineering Dept.	G: Shorten time to market by shortening integration time. G: The model has to be cheap in time, cost and human resources. G: Make modelling faster for future projects. G: Make better models in future projects.
Head of Software Engineering Dept.	G: Shorten time to market by shortening integration time. G: The model has to be cheap in time, cost and human resources; G: Model has to be adequate - to represent the plant so that the results obtained from it are meaningful.
Project leader	G: Finish each phase on time as planned. G: Design the model as soon as possible.
Modeller	G:Make a model of the plant with the tools suitable for the chosen kind of model. C: Modelling some parts would take too many hardware resources.
Software Developers	G: Develop the software and test it. C: Not experienced in language used for programming hardware.
Validation and Verification Engineer	G: Test and verify the software. C: Not experienced in language used for programming hardware.
Mechanical Dept.	G: Deliver the mechanical part. C: Not concerned about the models used for software verification. C: They have not made the plant yet.

Figure 12: *Stakeholders, their goals and the constraints they give on the model.*

4.8. The handbook – record of solution exploration (reusable modelling experience)

The modeller tried out some solutions that turned out not to work. To save time next time, we recorded the solution ideas and explanation as to why it did not work. For example, choosing the path segment length with respect to the paper length took many tries until the optimal solution was found.

There were equally good solutions, and the modeller chose one of them. We documented those other solutions, in case it turns out they suit the problem more next time. For example, in the part that represents inserter and folder, some of the elements belong to both modules, so in the model, they could just be represented in either of modules.

Not all the model parts were optimally designed. Some of the modelling decisions were taken from previous version of the system which went through significant changes. Some of the elements that were optimal for the system as it was before the changes were not optimal any more, but were left due to time pressure. We highlighted what these decisions are, having in mind that the modeller of the future system will not only have our guidelines but also the old model at his disposal. Therefore, we highlighted the decisions that might have to be reconsidered.

4.9. Model stakeholders

The table in Figure 12 shows the stakeholders of the emulator we analyzed. The stakeholders represent roles that also reflect the company's organizational structure.

The main sources of information about the inserter, its components and processes are mechanical and software engineers. Requirements for the model come from both heads of departments as well as the project leader. Some practical limitations are also coming from software engineers.

5. Discussion and conclusion

Our framework distinguishes different classes of plant-modelling decisions and modelling knowledge suitable for reuse. The main classes are as follows: reusable modelling steps and knowledge, requirements for the model, re-usable modelling strategies and a record of the solution exploration (modelling experience). In addition, to ellicit the requirements for the model itself, it is necessary to determine who the model stakeholders are.

The research we presented in this paper is part of the research that explores what the non-formal elements of plant modelling are, and what establishes stakeholders' trust that the model is adequate. Researchers of design call the non-formal elements 'creative' elements and they explore in general what they are. Our goal is to identify these steps for the plant modelling. As a result, non-formal modelling elements become explicit. When these steps are made explicit they can be reviewed, evaluated and discussed. Our framework also can be seen as an explicit description of non-formal modelling aspects.

In software engineering, there is a lot of work in automating the modelling process. Only those steps and decisions that can be formalized can be automated. Our research focuses

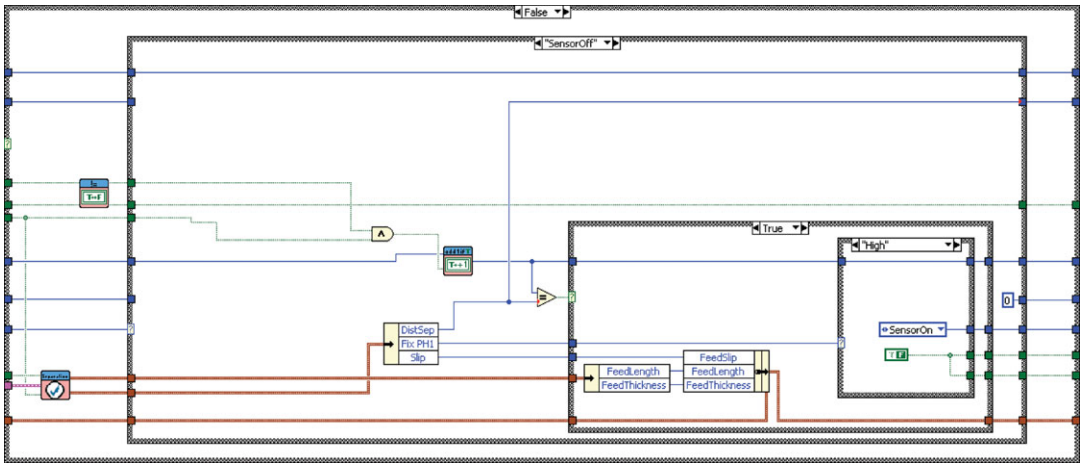


Figure 13: *Feeder state machine implemented in LabView.*

on those steps that cannot be formalized. Before pushing the button of a system that will design the model, there are many things that the modeller has to do on her own. We explore the generic elements of these decisions, and their relations to each other. Our work, therefore, complements the work on formalization and automatization.

Other methods address some elements similar to those of our framework. Our contribution is to collect and adjust the elements relevant to plant modelling, in contrast to software modelling, architecture design or software reuse. Moreover, where more general guidelines are little specific for the application domain and leave a lot of decisions to the modeller, we support the process of modelling such that it becomes more efficient and less dependent on the modeller's experience.

Of all the related work the POE framework is most similar to our approach. However, design is explored there on a more general level, it is not specified for plant modelling. Moreover, the design steps in the POE approach are formalized, which we did not do in our framework. This is due to the fact that many of the steps are inherently non-formalizable, and additionally, in the case study presented, there was no need for further formalization.

We represented our handbook in natural language, state machine diagrams, tables and sketches. The natural language descriptions may be ambiguous, and the plan for future work is to use a more standardized and less ambiguous notation such as SysML.

Appendix A

The feeders in the inserter machine are modelled as tracks with sensors, as shown on Figures 6, 7(a) and 7(b). Tracks in the rest of the document path pass the data about documents that move along them, while feeders state machines start with this data defined by the modeller in form of the model parameters.

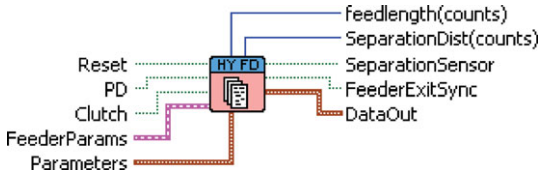


Figure 14: *Feeder module in LabView.*

Figure 13 shows the implementation of a feeder in LabView. LabView allows hierarchies, therefore on a higher abstraction level tracks are combined into the path. On this level, combination of state machines looks like data-flow diagram, with modules like the one on Figure 14, with data about documents passing from one track to the next.

References

Booch, G., R. Maksimchuk, M. Engle, B. Young, J. Conallen and K. Houston (2007) *Object-Oriented Analysis and Design with Applications*, 3rd edn, Boston, MA, USA: Addison-Wesley.

Coad, P., D. North and M. Mayfield (1995) *Object Models: Strategies, Patterns, Applications*, Upper Saddle River, NJ, USA: Yourdon Press.

Colombo, P., V. Del Bianco, L. Lavazza and A. Coen-Porisini (2007) A methodological framework for SysML: a problem frames-based approach, in *'Proceedings of the 14th Asia-Pacific Software Engineering Conference', APSEC '07*, Washington, DC, USA: IEEE Computer Society, 25–32.

Cross, N. (2001) Design cognition: results from protocol and other empirical studies of design activity, in C. Eastman, W. McCracken and W. Newstetter, eds, *Design Knowing and Learning: Cognition in Design Education*, Amsterdam, The Netherlands: Elsevier Science, 79–103.

Dutoit, A., R. McCall, I. Mistrk and B. Paech (eds) (2006) *Rationale Management in Software Engineering*, New York: Springer Verlag.

Gamma, E., R. Helm, R.E. Johnson and J.M. Vlissides (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA, USA: Addison-Wesley.

Hall, J.G., D. Mannering and L. Rapanotti (2007) Arguing safety with problem oriented software engineering, in *Proceedings of the Tenth IEEE International Symposium on High Assurance Systems Engineering, HASE 2007*, 14–16 November 2007, Dallas, Texas, USA: IEEE Computer Society, 23–32.

Hall, J.G. and L. Rapanotti (2008) The discipline of natural design, in *Proceedings of the Design Research Society Conference 2008*, 16–19 July 2008, Sheffield, UK: Sheffield Hallam University Research Archive.

Hall, J.G., L. Rapanotti and M. Jackson (2008) Problem oriented software engineering: solving the package router control problem, *IEEE Transactions on Software Engineering*, **34**, 226–241.

Harel, D. and A. Naamad (1996) The statemate semantics of statecharts, *ACM Transactions on Software Engineering and Methodology*, **5**, 293–333.

Huang, E., R. Ramamurthy and L.F. McGinnis (2007) System and simulation modeling using SysML, in *Proceedings of the 39th Conference on Winter Simulation: 40 years! The Best Is Yet to Come, WSC '07*, Piscataway, NJ, USA: IEEE Press, 796–803.

Jackson, M. (1995) *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*, New York, NY, USA: Addison-Wesley.
Jackson, M. (2000) *Problem Frames: Analysing and Structuring Software Development Problems*, Boston, MA, USA: Addison-Wesley.
Kruchten, P. (2004) An ontology of architectural design decisions in software-intensive systems, in *Proceedings of the 2nd Workshop on Software Variability Management*, 3–4 December 2004, Groningen, NL.
Kruchten, P., P. Lago and H. van Vliet (2006) Building up and reasoning about architectural knowledge, in QoSA, Heidelberg: Springer Berlin 43–58.
LabView (2011) Available at http://www.ni.com/labview/ (accessed 15 April 2011).
Mader, A.H., H. Wupper, M. Boon and J. Marincic (2008) A taxonomy of modelling decisions for embedded systems verification, Technical Report TR-CTIT-08-37, Enschede, The Netherlands: Centre for Telematics and Information Technology University of Twente.
Marincic, J., A.H. Mader, R.J. Wieringa and Y. Lucas (2010) Structuring problem analysis for embedded systems modelling, in *International Workshop on Applications and Advances of Problem-Orientation (IWAAPO)*, Cape Town, South Africa.
Mili, H., A. Mili, S. Yacoub and E. Addy (2002) *Reuse Based Software Engineering: Techniques, Organizations, and Measurement*, John Wiley & Sons, Inc.
NeopostTechnologies (2011) Available at http://www.neopost-technologies.nl (accessed 15 April 2011).
Nkwocha, A., J.G. Hall and L. Rapanotti (2010) Design rationale capture in the globalised enterprise: an industrial study, *Software Engineering Advances, International Conference on*, **0**, 284–289.
Paredis, C.J.J. and T. Johnson (2008) Using OMG's SysML to support simulation, in *Proceedings of the 40th Conference on Winter Simulation, WSC '08*, Winter Simulation Conference, 2350–2352.
Pohl, K., G. Böckle and F.J.V.D. Linden (2005) *Software Product Line Engineering: Foundations, Principles and Techniques*, Secaucus, NJ, USA: Springer-Verlag New York, Inc.
Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991) *Object-Oriented Modeling and Design*, Upper Saddle River, NJ, USA: Prentice-Hall, Inc.
SysML – Open Source Specification Project (2011) Available at http://www.sysml.org/ (accessed 15 April 2011).
Turski, W.M. (1986) And no philosopher's stone, either, in *Information Processing 86, Proceedings of the IFIP 10th World Computer Congress*, Dublin, Ireland: Elsevier Science Publishers B. V. (North-Holland), 1077–1080.
UML – Unified Modeling Language (2011) Available at http://www.uml.org/ (accessed 15 April 2011).
Vincenti, W.G. (1990) *What Engineers Know and How they Know It: Analytical Studies from Aeronautical History*, Baltimore: Johns Hopkins University Press.

The authors

Jelena Marincic

Jelena Marincic is a PhD candidate at the University of Twente at the Faculty of Electrical Engineering, Mathematics and Computer Science in the Netherlands. She is investigating the non-formal elements of the modelling process that accompany formal elements, structuring them to make modelling less dependent on a modeller's talent and experience. Previously, she has worked as an embedded software engineer on design and verification of software for telecommunication and mechatronic systems. She holds a Dipl. Ing. degree in Electrical Engineering from the University of Belgrade in Serbia.

Angelika Mader

Angelika Mader is an assistant professor in the Control Engineering Group of the faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente in the Netherlands. Her research interests include creative technology, and modelling and verification of embedded systems. On the latter she has also been working in Computer Science departments, at both the University of Twente and the University of Nijmegen in the Netherlands. Her PhD was in the area of logics and verification, from the Technical University of Munich.

Roel Wieringa

Roel Wieringa is Chair of Information Systems at the University of Twente, the Netherlands. His research interests include requirements engineering, conceptual modelling and research methodology for software engineering and the design sciences. He has written two books, Requirements Engineering: Frameworks for Understanding (Wiley, 1996) and Design Methods for Reactive Systems: Yourdon, Statemate and the UML (Morgan Kaufmann, 2003). He has been Associate Editor in Chief of IEEE Software for the area of requirements engineering from 2004 to 2007. He serves on the board of editors of the Requirements Engineering Journal and of the Journal of Software and Systems Modeling.

Yan Lucas

Yan Lucas is a manager of the Software and Electronics Development at Neopost Technologies in Drachten in the Netherlands. He has been involved in introducing and implementing the agile development method as well as several software product lines. His other responsibilities include defining and implementing technology roadmaps; offshoring activities to Hanoi, Vietnam and coordinating Neopost Technologies teams in multiple countries. He has been in software engineering for 17 years, having worked as a project manager, consultant and software engineer.