Saving the Software Specification by Transforming the old SA/RT Models into UML

Thomas Weyrath⋆, Herbert Schreyer⋆⋆, and Jürgen Sellen⋆
System Support Center NH90 / Tiger
⋆Elektroniksystem- und Logistik-GmbH
Thomas.Weyrath@esg.de
Juergen.Sellen@esg.de
⋆⋆Eurocopter Deutschland GmbH
Herbert.Schreyer@eurocopter.com

Abstract

The avionic software of the military helicopters Tiger and NH90 has been developed since the 1990s. Their specifications are based on Structured Analysis / Real Time (SA/RT), a methodology that was state-of-the-art in the 1980s. But nowadays the tools for modeling SA/RT are no longer supported and the knowledge to understand and maintain the models is going to diminish over time. Hence, ways must be found to keep the expertise on the specifications, models and software for future work. One possible solution for this problem is a transformation of the existing software models from SA/RT to UML. In this paper, we describe the results of an investigation concerning procedural method, practicability, and implications of such a transformation.

1. Introduction

The System Support Centre NH90/Tiger (SSC) is a cooperative organisation of German Federal Armed Forces, Eurocopter, ESG, Cassidian and MBDA. It comprises a team of 200 engineers. Its main task is the avionic software maintenance and modification of the military helicopters NH90 and UH Tiger. The maintenance comprises more than 12.000.000 lines of code in 16 avionic computers, including the ground support system. Additionally, the documentation of the development is immense. It contains over 500 documents of specifications and design descriptions. Most of them have hundreds of pages, some of them over a thousand. In view of the fact that the helicopters have a life cycle of more than 40 years, the software maintenance and modification is a challenge, but it has to be done.

1.1 Rationales behind our Work

Over the years the helicopter software will be changed due to modifications in the software environment, new customer requirements or simply by fixing errors. Even an extensive facelift towards modern technologies cannot be ruled out. Some points guide our actions:

- The software changes will have to be done efficiently and reliably. This includes that current standards, processes and methodologies of software maintenance and modification have to be applied.
- The human expertise of the past and present must be maintained as long as our helicopters exist. As their software has been developed since the 1990s, a lot of engineers of this time have already gone into retirement or will be retiring in the next ten years. But their valuable knowledge must not be lost. Hence, the older colleagues should share their knowledge with the younger ones and enthuse them for the technical implementation just as it is.
- The processes, methods and tools need to be improved and adapted to the technological progress, otherwise future obsolescence problems are unavoidable. Hence, the younger colleagues should bring a fresh breeze into the development and enthuse the older ones for their new technologies.
- When modifying the specifications and software we want to introduce well-accepted software concepts, like modularization, structuring and abstraction to improve the readability and understandability. This should enable us to create, modify, and maintain the software easier and more cost-effectively.

- The contractual conditions, e.g. concerning standards, have to be changed likewise. The customer and the management are needed here to clear the way ahead.

1.2 The Teamwork Obsolescence Problem

Teamwork is a CASE tool, which supports the development of SA/RT models based on the method of Hatley and Pirbhai [7]. The tool was applied in the context of the helicopter Tiger development to create the specification and design of a large part of the operational software. But Teamwork is obsolete and must be replaced. The tool maintenance has been discontinued in 2002 after a shift from Cadre Technologies to Cayenne Software, today Computer Associates. In addition, the operating system Solaris on which Teamwork is running is outdated, too. Hence, the tool support for the SA/RT methodology is vanishing. To solve the Teamwork obsolescence problem we have identified three ways :

- *Alternative 1: Preservation of the SA/RT method.* In this alternative the SA/RT method will be retained. However, it must be clarified what has to be done with the obsolete tool. Two options are possible: a) keep Teamwork running anyhow (e.g. running on an emulated Solaris plattform) or b) replace Teamwork by another SA/RT tool (e.g. by Statemate).

- *Alternative 2: Transfer of the Teamwork model artefacts as text and figures into a requirements engineering tool, like DOORS.* Because many specifications of NH90 and Tiger are already managed with DOORS, this alternative would have the following advantages: a) DOORS is widespread in requirements engineering and accepted within the company, b) the specifications are stored in a central place and are under configuration management, and c) the traceability between specifications can be done within the tool, etc. In a first step we have transformed the Teamwork model into HTML where the diagrams are converted into SVG figures. But transferring the Teamwork model into DOORS means also the loss of modeling and a switch to natural language based specifications.

- *Alternative 3: Transformation of the SA/RT model into UML.* In this alternative we can retain the model-based method, which has clear benefits compared with purely text-based requirements. UML is gaining popularity and constitutes the *de facto* standard notation used for modeling software systems. UML is therefore a future-proof technology. There is much literature confirming that SA/RT can be modeled with UML constructs (see [8, 9]).

In this paper we focus on the third alternative. That means, we want to a) introduce a new tool for the software specification instead of an older one (Teamwork) and b) replace the old methodology (SA/RT) by a newer one (UML).

2. Methods for Real-time System Specification

SSC uses the same processes as the development department of Eurocopter even for the tasks of software maintenance and modification. This is done for reasons of certification, which stipulates standardized processes for development and assurance (see [2]). Viewed from this perspective a lot of effects have to be considered by eliminating an obsolescence problem. In our case the following side effects occur:

- The new UML based specification methodology must be described in the Software Requirements Standards (respectively Software Model Standards). This document defines a) the modeling techniques regarding the methods and the tools used for developing the models, b) the modeling languages, styleguides and conventions, and c) the method to be used to identify and delimit requirements (and derived requirements) contained in the model [2, 3].

- The introducing of a new tool must be considered in the Software Development Plan regarding the development environment, configuration management, quality assurance, qualification, etc.

- After transferring the software models from SA/RT to UML, it must be verified that the results are compliant to the Software Requirements Standards. Furthermore, it must be shown that all requirements of the SA/RT model are represented adequately as requirements in the UML model and that no requirement is lost or has been added.

The next two subsections describe the formerly used SA/RT method and the new UML methodology that should replace it.

2.1 SA/RT Methodology

Structured Analysis (SA) is a method to develop the specification of a software system. It was created by Tom DeMarco in 1978 [6]. The aim is to describe the software system through an understandable model. The idea is to decompose the functionalities of the software system successively and to specify the data that flows in between. The decomposition is expressed graphically by a set of data flow diagrams (DFDs) that are hierarchically structured. The hierarchy starts on the highest level with the context diagram that shows which actors outside of the system interact with the system. An example context diagram is shown in figure 3.

Real-time systems have a great portion of control signals and timing constraints. But the classical SA is lacking in description elements that allow the modeling of real-time and control behavior of a system. Structured Analysis with real-time extensions (SA/RT) was developed to make up for this drawback by a) a clear seperation of data and control flow and b) techniques for modeling finite state machines to describe control and real-time behavior of a system. Important representatives of this approach are Hatley and Pirbhai [7]. Their method is also supported in the tool Teamwork. The SA/RT approach extends the SA method and provides three models: the process model, the control model and the data dictionary.

2.1.1 Process Model

The process model is described by data flow diagrams (DFDs) and process specifications (P-Specs). DFDs consist of:

- *Data flows*, which represent pipelines through which data of known composition flows. A data flow may consist of a single element or a group of elements. A data flow is a solid arc with a name.
- *Processes*, which indicate the transformation of incoming data flows into outgoing data flows. A process is a circle with a name and a number.
- *Data stores*, which are data flows frozen in time. A store is a pair of parallel lines containing a name.
- *Terminators* (or actors), which represent entities outside the context of the system that transmit and receive data. A terminator is a rectangle containing a name and occurs only in a context diagram.

A process can be refined again by a lower-level DFD. If the refining is sufficiently detailed, then the process is an elementary process described by a P-Spec with texts in prose or pseudo code.

2.1.2 Control Model

The control model consists of control flow diagrams (CFDs) and control specifications (C-Specs). The purpose is to determine which processes are enabled or disabeled under given external or internal conditons or operating modes. The decomposition of the system into CFDs parallels the decomposition into DFDs: each CFD is associated to an "adjacent" DFD, and contains the same processes as this DFD. In addition, CFDs consist of:

- *Control flows*, which are pipelines through which control information of known composition flows. A control flow is a dashed arc with a name.
- *C-Spec bars*, which indicate the interface between the CFD and an associated C-Spec. A C-Spec bar is a short unlabeled bar.
- *Control stores*, which are control flows frozen in time. The symbol is identical with the data store symbol.

Each CFD may have one associated C-Spec. C-Specs evaluate input control signals and produce output control signals or process controls. A C-Spec contains combinational and/or sequential machines, defined via decision tables (DTs), state transistion diagrams (STDs), and process activation tables (PATs). DTs and STDs process the control signals, which are then mapped to process activators via a PAT.

2.1.3 Data Dictionary

The data dictionary lists the data and control flow names, and data and control store names, and specifies by regular expressions how they are composed.

2.1.4 High-level Requirements in the SA/RT Model

The DFDs and P-Specs in the Process Model, the CFDs and C-Specs in the Control Model, and the data dictionary represent the high-level requirements. This scheme reflects the evolution of the two methods: the process model corresponds to the classical structured analysis model [6], the control model is the part added by Hatley and Pirbhai [7], derived from the finite state machine theory; and the requirements are represented by the complete integrated combination of the two.

2.2 UML Methodology

The Unified Modeling Language (UML) [5, 10] is a language with well-defined syntax and semantics (see [13]) but without methodology. That means that UML (in contrast to SA/RT) does not provide any procedures or methods how models should be created and which diagrams should be used to build a clear and understandable model of the software. To fill this gap, there are several well-known methodologies like IBM's Rational Unified Process (RUP), Telelogic's Harmony-SE or INCOSE's OOSE (for a survey see [14]). But all these methodologies do not fit our software development processes based on the "Vee" model. Hence, and with help of external consultants, the SSC has established a customized UML methodology for software requirements and design (see [12]). Furthermore, this methodology refines an EADS guideline for using UML (see [11]).

According to the software development process our UML methodology provides two models: a software requirements model and a software design model. In this paper we focus on the software requirements process, which uses the outputs of the system life cycle processes to develop the high-level software requirements. These high-level requirements include functional, performance, interface, and safety-related requirements (see [2]). The software requirements process is methodologically broken down into two smaller steps: *Software Use Case Modeling* and *Software Domain Analysis Modeling*.

2.2.1 Software Use Case Modeling

The Software Use Case Modeling establishes the system's functional requirements as black-box view. Use cases capture the intended behavior of the system that should be specified independent of their realization. They provide a way for engineers and developers to obtain a common understanding with customers and users. The Software Use Case Modeling defines the following steps:

1. Decompose the software system into high level functions to handle the complexity of the model. Avionic software systems are very complex. Hence, there is a need to devide their functionality into functional components (or functional partitions, see [4]) to handle the complexity of the model.
2. Perform a use case analysis for each high level function. Identify use cases and actors. Structure them graphically and specify the scope of the system (boundary box).
3. Refine the use cases with use case descriptions and further natural language requirements and model the use case behavior with an activity diagram. Each action of the basic scenario and the alternative scenarios of the use case are represented by a UML action. Swim lanes can be used to specify the subject (for details, see [12]).
4. Model the activity that triggers the use cases. That can be function calls, e.g. from an external interface, or a user interaction event (modeled by send and receive actions).

2.2.2 Software Domain Analysis Modeling

The Software Domain Analysis Modeling defines the domain specific vocabulary and refines the results of the use case analysis to high-level software requirements, that are correct, unambiguous, complete, consistent, verifiable, modifiable and traceable [1]. The software domain analysis is performed as an expanded use case analysis with focus on the functional aspects of the software under development (instead of an object-oriented analysis). This is closer to the used development standards, DOD- 2167A respectively DO178-B/C. The Software Domain Analysis Modeling defines the following steps:

1. Create a domain model, which defines the vocabulary and the domain specific knowledge. It consists of actors (human actors and external systems), data dictionary and data types.
2. Model the external interfaces to the actors.
3. Develop natural language requirements to specify non-functional requirements or requirements that are too involved to model with UML.

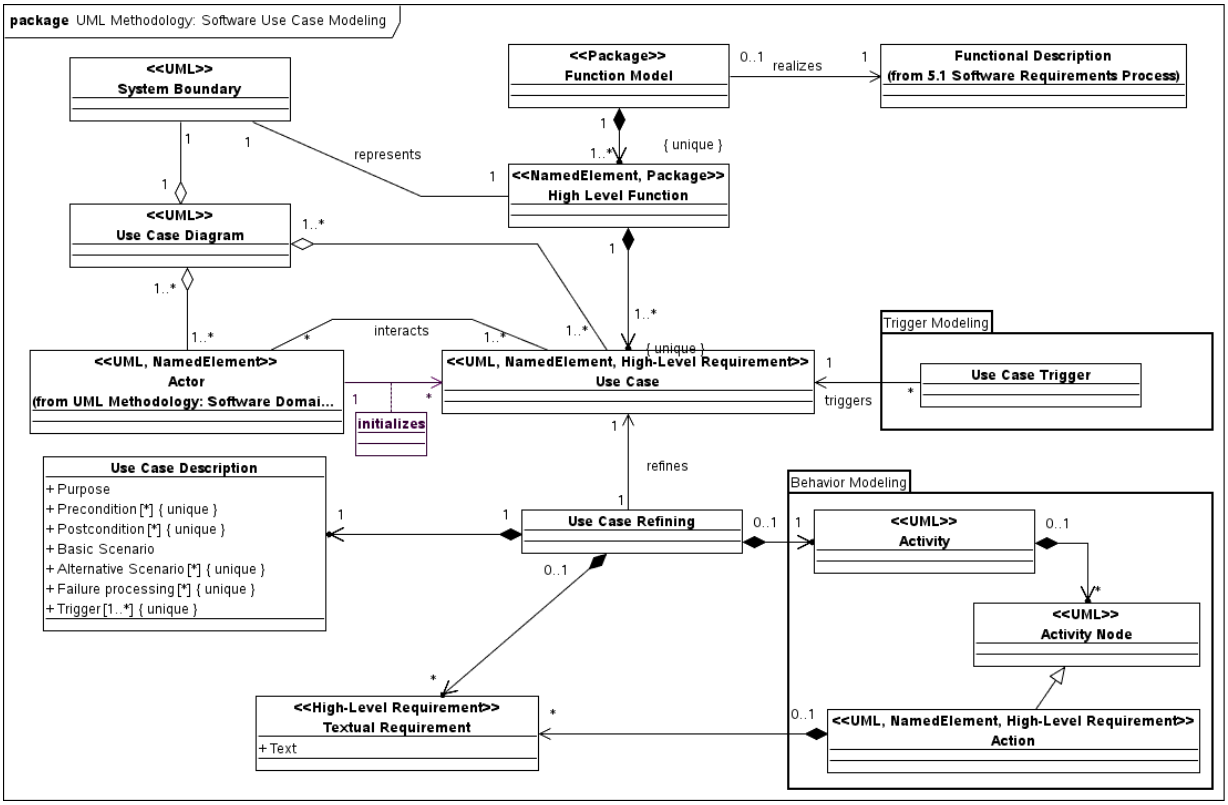

Figure 1: Meta Model of the Software Use Case Modeling.

2.2.3 High-level Requirements in the UML Model

Figure 1 depicts the meta model of the software use case modeling. It shows how the different concepts are related to each other. The concepts "Use Case" and "Action" (Behavior Modeling) represent high-level requirements contained in the model. In the domain analysis modeling the entries of the data dictionary and all data types are high-level requirements.

3. Modeling of Software in Military Helicopter Avionics

3.1 System Description

Military helicopters have a quite complex equipage of avionic devices. Multiple computers control the capabilities of flying, even at night and under worse weather conditions, and of acquisition and combat of possible threats. A typical system architecture is divided into a basic system and a mission system. Basic avionics is responsible for acceptance of crew inputs, navigation, communication, engine control, flight control and the generation of the symbology for Multi Function Displays. The mission system steers and evaluates the different sensors for piloting and target acquisition and operates the weapons.

Each of these systems is controlled by a main computer, which receives the entries of the crew and the signals coming from the avionic equipment (see figure 2). These computers evaluate all the inputs and send the necessary commands to the devices. A further important task is the generation of symbology for the different displays in the cockpit, as e.g. Multi Function Displays or Helmet Mounted Displays. The communication between the avionic equipments is realized via one or several data buses, e.g. a 1553B Bus (Milbus). The software running on these computers has the size of about half a million lines of code. Most of the processes are running cyclic, to ensure continuously up-to-date information on the displays and the avionic equipments. For nonrecurring processes acyclic events have been defined as e.g. for loading waypoints etc.

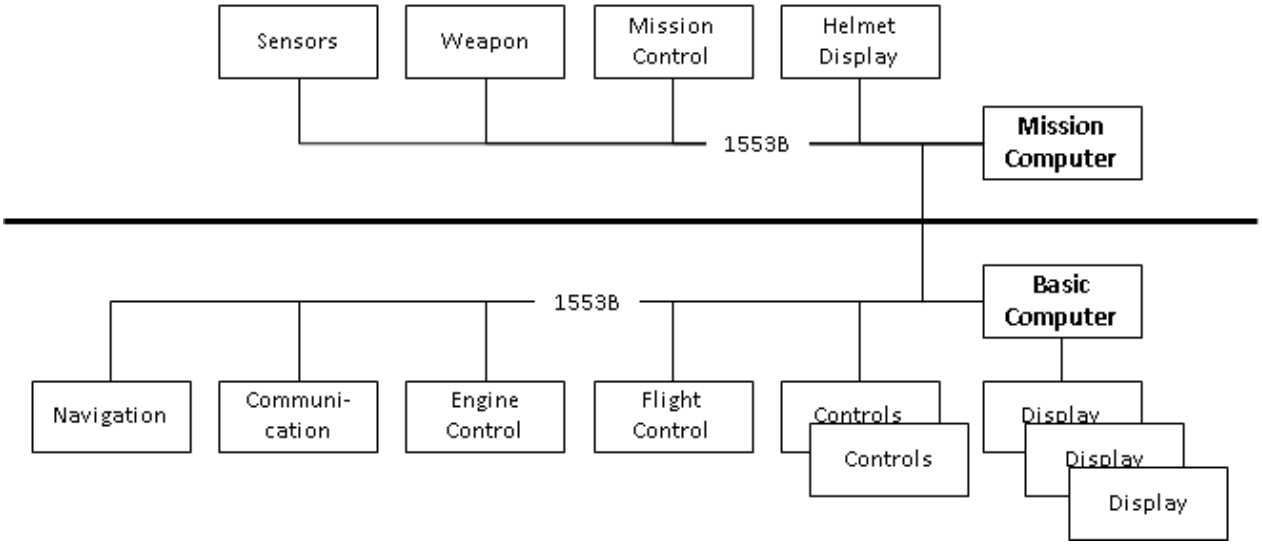

Figure 2: Typical System Architecture.

3.2 Structure of our Models

Military projects have usually a quite long development period. At the start time of our projects the Structured Analysis (SA) was state-of-the-art and has been used as method for the software specification. Teamwork, a well established tool for SA, at that time, has been chosen to develop the models. Several years later the market for tools supporting the SA methodology decreased, partly due to the spread of UML. Finally the tool manufacturers vanished, with the consequence that the tools are no longer supported.

The software requirements specification of each computer is realized in a separate model. A typical model has a size of approximately 800 processes, the data dictionary contains approximately 7000 entries. To achieve a good understandibilty of such a complex system the model is decomposed via a top-down approach. The decomposition is oriented according to the functionality, the hierarchy goes down up to 12 levels.

Models of such magnitude cannot be transformed manually into a new method of description, the huge effort in terms of time and money would never be accepted by program and project responsibles. Therefore, a way has to be found that extensively automates this process.

4. Transformation from SA/RT to UML

In this section, we propose a transformation method which we conceive as best fitting our requirements. In order to demonstrate the transformation method with an example, we use the SA/RT model delivered with the Teamwork tool, a traffic light pre-emption. This model is publicly available, has a known problem domain, and, though small, has all ingredients which we also find in our models.

The UML example diagrams have been prepared with the UML tool TOPCASED (see [15, 16] for further information).

4.1 SA/RT Example Model "Control Traffic Lights"

The traffic lights example has a hierarchical decomposition with 4 levels. The decomposition induces a tree structure that can be viewed in Teamwork as "process index". The highest level (or root) is given by the context diagram which describes the interfaces of the system. The context diagram from Teamwork is shown in figure 3.

The top-level DFD "control_traffic_lights" is shown in figure 4 and decomposes the system into three sub-processes. Each DFD contains control and data flows between the sub-processes, as well as flows between sub-processes and the upper level. Data flows may terminate at data stores.

In this paper, we will investigate the further decomposition of the sub-process "crossing_request_button". The DFD for this process is shown in figure 5.

All sub-processes of the process "crossing_request_button" are on the lowest level (P-Specs or Mini-Specs), i.e., they have no further decomposition. The activation sequence of these processes is described inside the DFD by a

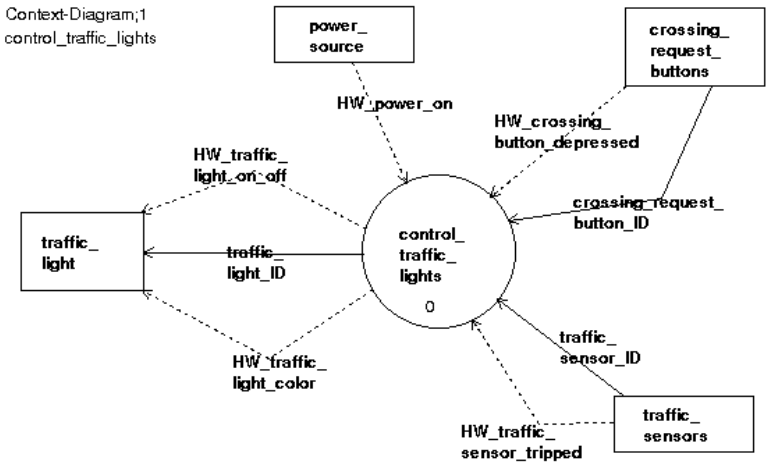

Figure 3: Context Diagram of the Traffic Lights Example.

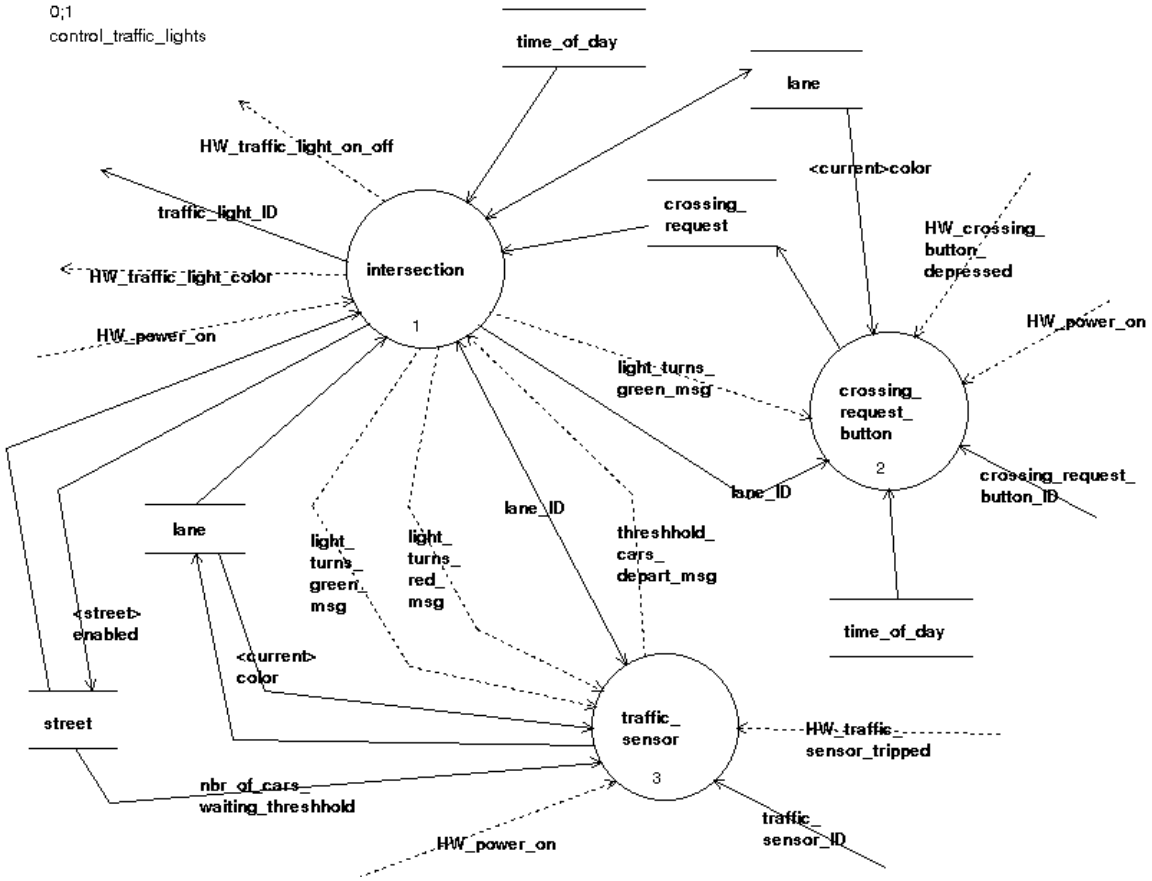

Figure 4: Top-Level DFD "control_traffic_lights".

sequential machine consisting of a state transition diagram (STD, see figure 6) and a process activation table (PAT, see figure 7) [1]. The STD produces an array of four booleans which are then input to the PAT. Note that, in Teamwork, the process model and the control model are not strictly divided but described within the same diagrams (DFDs).

The data flows, control flows, and data stores are recorded in the data dictionary. Note that even elementary control flows are associated with a data value (usually a boolean).

[1] The process "accept_crossing_request" referenced in the PAT does not exist and obviously corresponds to the process "identify_regulated_street" in the DFD. This seems to be a bug in the traffic lights example model delivered with Teamwork.

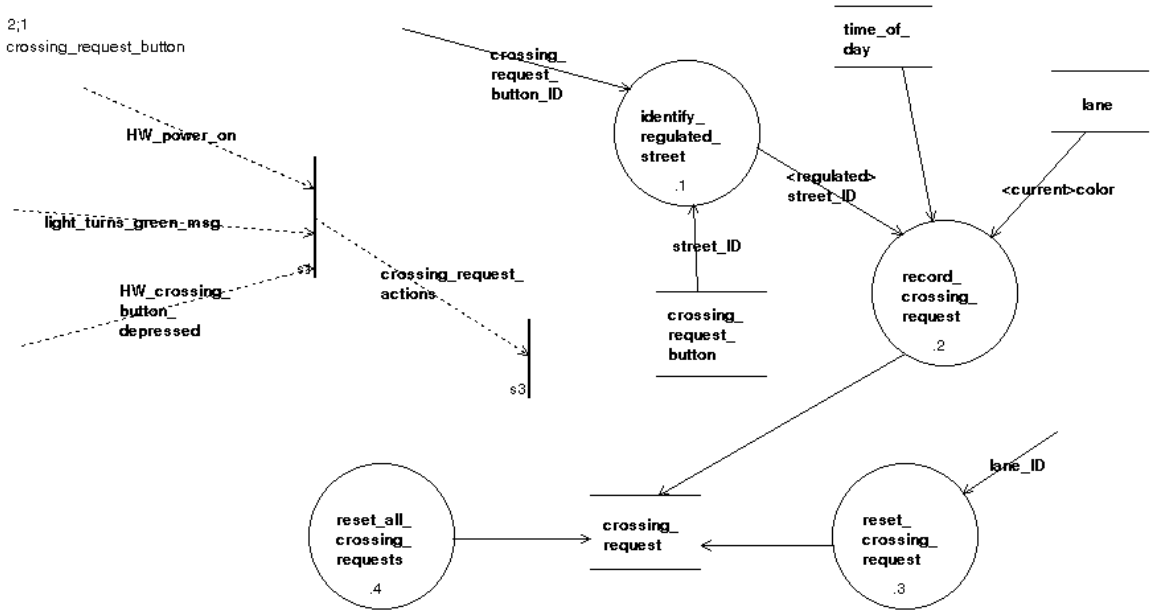

Figure 5: DFD "crossing_request_button" with Control Model.

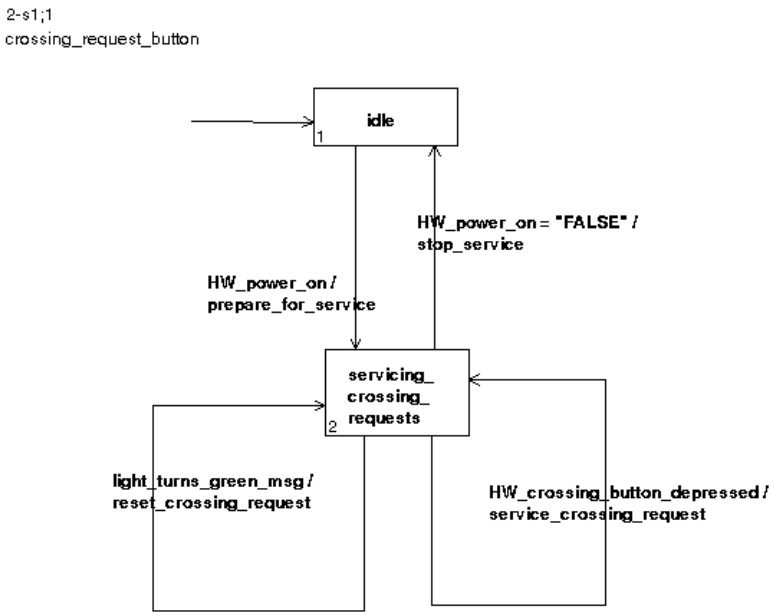

Figure 6: STD "crossing_request_button".

4.2 Transformation Process

4.2.1 Step 1: Transforming the Hierarchical Structure

The hierarchical process structure of SA/RT (as given by the "process index" in Teamwork) is transformed into a hierarchical UML package structure:

- The UML package "Functional Model" represents the top-level DFD. Each process which is refined by another DFD represents a UML package with the same name, marked with stereotype «process». The process packages may contain activity diagrams, state machines, etc.

- The UML package "Domain Model" contains the data dictionary.

prepare_ for_ service	service_ crossing_ request	reset_ crossing_ request	stop_ service	1	2	3	4
"TRUE"				0	0	0	1
	"TRUE"			1	2	0	0
		"TRUE"		0	0	1	0
			"TRUE"	0	0	0	0
				* accept_ crossing_ request *	* record_ crossing_ request *	* reset_ crossing_ request *	* reset_all_ crossing_ requests *

Figure 7: PAT "crossing_request_button".

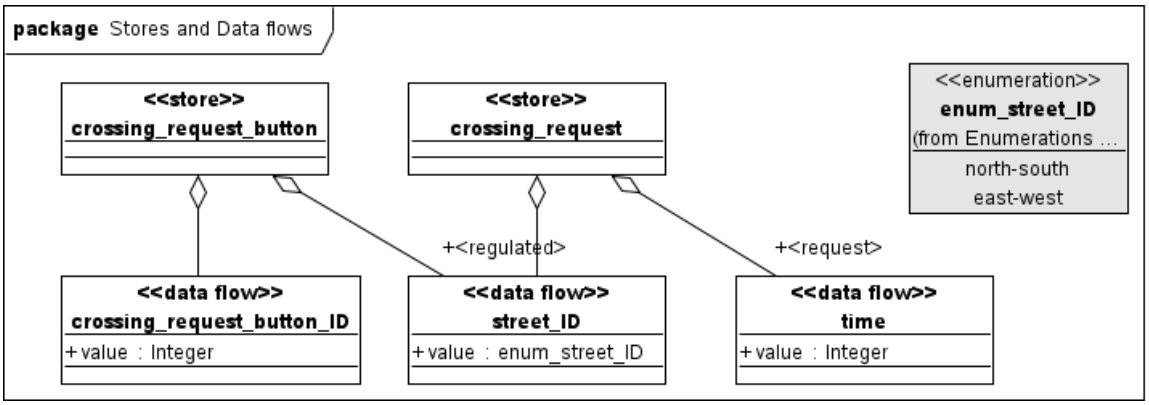

Figure 8: UML Domain Model of the Traffic Lights Example (Excerpt).

4.2.2 Step 2: Transforming DDEs to a UML Domain Model

Each Data Dictionary Entry (DDE) is modeled as UML class. The DDE type is specified by the stereotypes «store», «data flow» or «control flow». Note that, in the Teamwork tool, control flows are handled like data flows, i.e., they are associated to a data item and appear in the data dictionary. This artefact of the Teamwork model is preserved in our UML model.

DDEs can be primitive (e.g. integer or enumeration) or composed by other DDEs. The hierarchical structure of DDEs is modeled in UML via aggregation. All DDEs, including the primitive ones, are modeled as classes (opposed to class attributes). This underlines the uniqueness of each DDE according to its name, and allows to keep the characteristics of a global data dictionary. Enumerations are modeled as UML Enumerations.

Figure 8 shows an excerpt of the domain model for the traffic lights example.

4.2.3 Step 3: Transforming DFDs to UML Activity Diagrams

Data flow diagrams (DFDs) are the main building block of SA/RT, and are used for the structural decomposition of the system. We will distinguish between the following "levels" of DFDs in our transformation: the *top-level* diagram (context diagram), *intermediate-level* DFDs, and *low-level* DFDs.

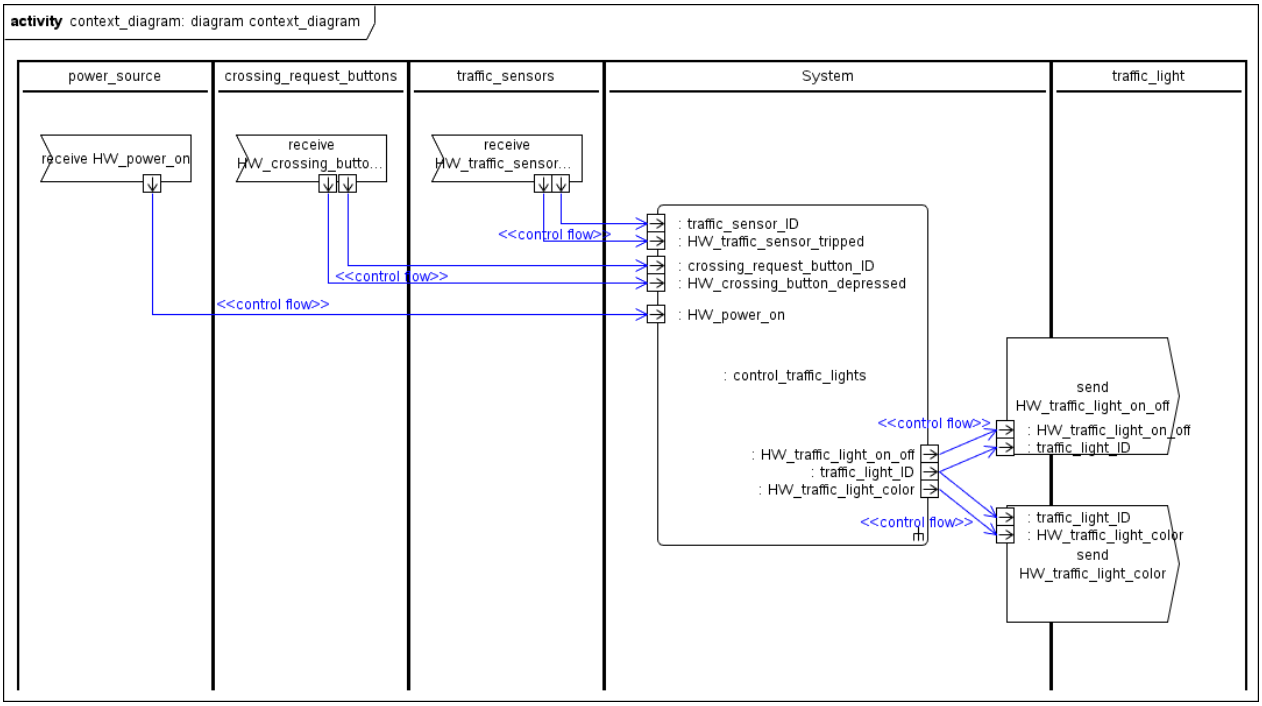

Figure 9: UML Activity representing the Context Diagram of the Traffic Lights Example.

Low-level DFDs are DFDs which contain C-Specs (i.e., DTs, STDs, PATs) or have no further decomposition. Usually, the two conditions for low-level DFDs coincide, i.e., process activation is modeled on the lowest level of the decomposition hierarchy. We note that intermediate-level DFDs are merely an artefact of the decomposition. In general, all DFDs are transformed to UML activity diagrams in accordance to the following rules:

- Elementary processes (i.e. processes which possess no further decomposition) are mapped to UML opaque actions. The textual descriptions (P-Specs) are copied directly to annotation attributes of the actions.
- Non-elementary processes (i.e. processes which correspond to lower-level DFDs) are mapped to UML call behavior actions.
- SA/RT stores are mapped to UML data stores.
- Data flows are modeled via UML object flows which terminate at input/output pins of UML actions. The input/output pins of the actions are tagged with classes from the domain model. External sources/sinks of data flows are modeled via activity parameters (flows from the upper level correspond to input parameters, flows back to the upper level correspond to output parameters).
- Control flows between processes and/or other DFDs are modeled in the same way as data flows, but with a special stereotype «control flow».

The transformed context diagram of the traffic lights example is shown in figure 9. For the context diagram, we used a transformation in which the interfaces of the system are modeled as *send signal actions* or *accept event actions*. Start or end symbols are not used inside the diagram. This underlines the event-driven character of the system, which continuously waits for sensor input.

The transformation for an intermediate-level DFD (DFD "control_traffic_lights" from figure 4) is depicted in figure 10. The call behavior actions are linked with UML activities that represent lower-level DFDs.

We note that SA/RT control flows in intermediate-level DFDs are represented in our UML model by UML data flows, not by UML control flows. In SA/RT, the control flows entering or leaving the processes do *not* activate or deactivate those processes [7], contrary to UML. The real control model in SA/RT is not contained in the intermediate-level DFDs, but in the low-level DFDs and their contained DTs, STDs, and PATs. The transformation of these elements will be the subject of step 4.

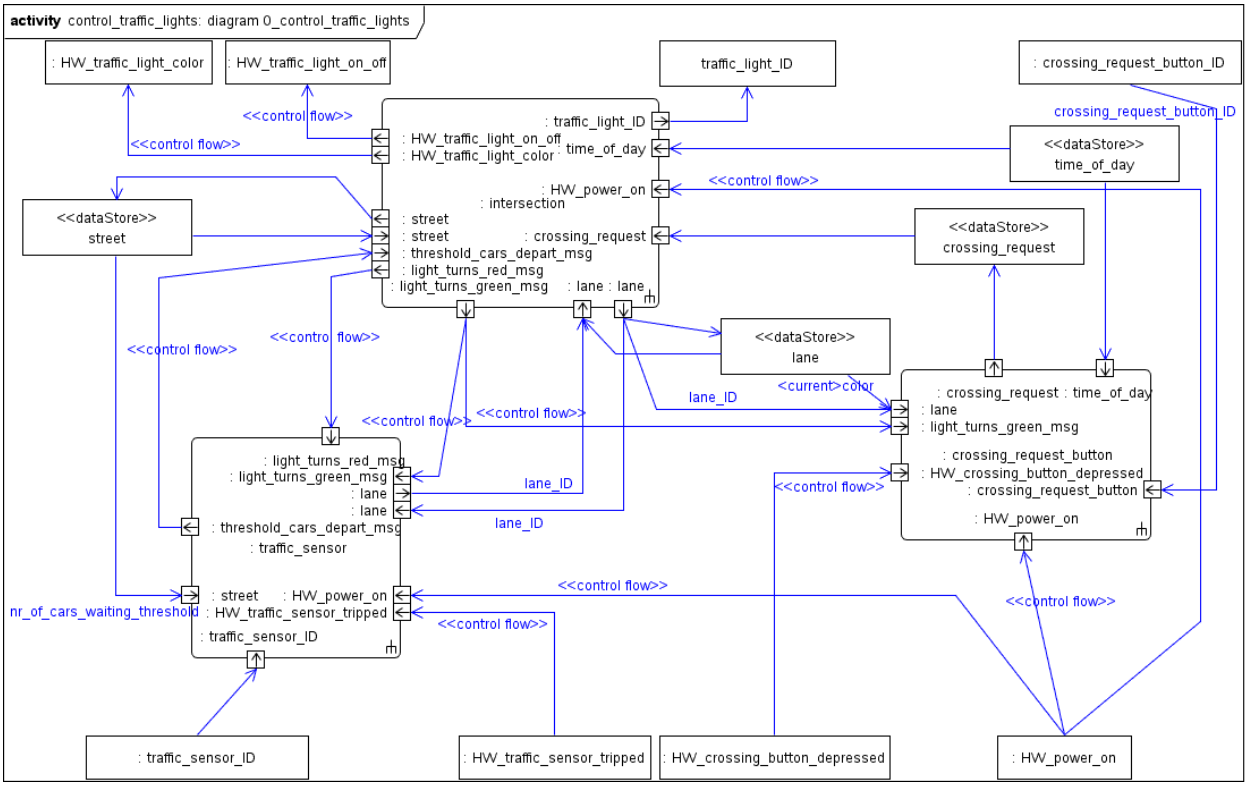

Figure 10: UML Activity representing DFD "control_traffic_lights".

4.2.4 Step 4: Transforming Control Specifications (DTs, STDs, PATs) to UML

The C-Specs which are part of the low-level DFDs in Teamwork describe the activation sequence of processes via combinations of DTs, STDs, and PATs. Each low-level DFD is first transformed to an activity diagram according to step 3. Afterwards, UML control flow elements representing the process activation are added.

STDs (e.g. STD "crossing_request_button" in figure 6) are transformed directly into UML state machines. The close coupling between the DFD and its contained STD is represented by our UML package structure: in the UML package structure, the state machine is located directly beside the activity diagram that "uses" input from the state machine.

PATs (e.g. PAT "crossing_request_button" in figure 7) are transformed into UML control flows inside the corresponding activity diagram. In our example, we use a decision node which receives the input from the STD as activity parameter. However, this is not supposed to be a general rule, as further control nodes may be required in more complex cases to find a readable diagram structure. The resulting activity diagram for our example is shown in figure 11.

4.2.5 Step 5: Identifying the Use Cases

In the used UML methodology, high-level software requirements are to be described by use cases. Thus, the diagrams and P-Specs must be analyzed to derive corresponding use-cases.

The major use cases of the traffic lights example are depicted in figure 12 and can be refined according to the structural decomposition starting at the context diagram.

4.3 Automation Aspects

The migration from an existing large-scale SA/RT model into UML is only practicable if the following conditions are satisfied:

- The user of the model (requirements engineer) is provided with a complete model that allows day-to-day modifications without the necessity of major re-design beforehand.
- The structure of the model (which is well-known to the current users and to the customer) is kept as much as possible.

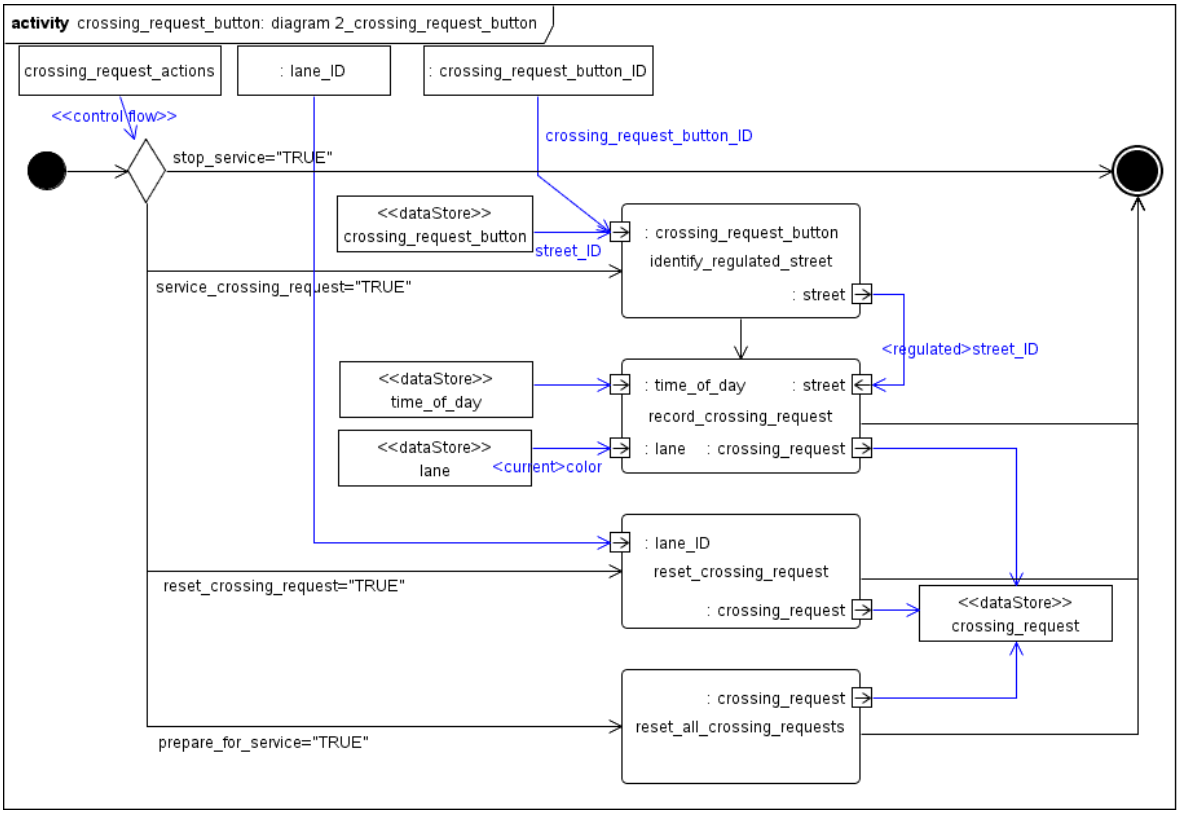

Figure 11: UML Activity representing DFD "crossing_request_button".

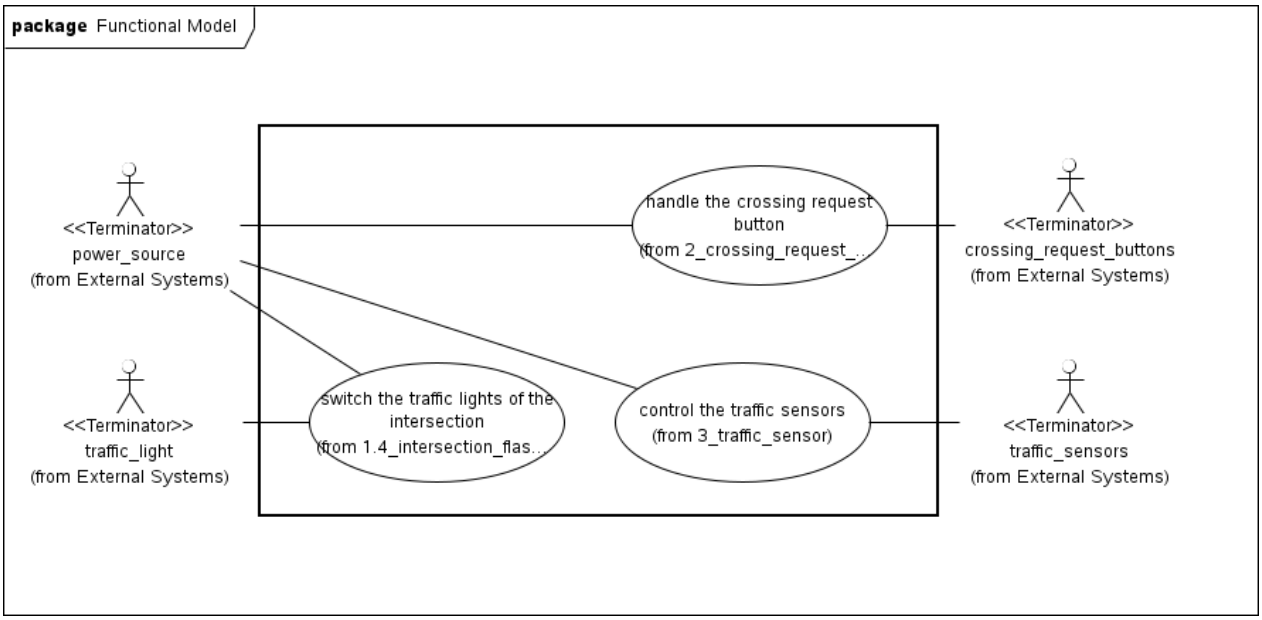

Figure 12: UML Use Cases for the Traffic Lights Example.

- The model elements are cross-linked and allow easy navigation through the model by adequate UML tools.
- The transformation can be automated to such an extent that only few parts need to be modified manually.
- The UML model can be shown to be equivalent to the original SA/RT model. Note here that this evaluation can be substituted by an evaluation of the used transformation method, plus a check of the manually modified model elements.

The proposed transformation fulfills these criteria, as most steps can be automated. The possibility to dump the Teamwork model in a proprietary but readable ASCII file format, and to use the standardized XMI format to interface with UML tools, allows to implement a generic transformation software for these steps. We also note that the graphical layout of the DFDs can largely be preserved in the activity diagrams (e.g., the location of processes, respectively, actions). This is an important issue concerning the readability and thus the usability of the transformed diagrams.

The only steps which require manual interaction are the modeling of the context diagram, the inclusion of control flow in activity diagrams for low-level DFDs, and the derivation of use cases. These modifications concern parts of the SA/RT model where completeness and soundness of the software requirements can be gained by migrating to UML.

4.4 Pros and Cons of the Proposed Transformation

In the literature, many possible transformations from SA/RT to UML are discussed, and it seems that there exists no silver bullet to solve this problem. Our transformation strategy converts SA/RT data flow diagrams into UML activity diagrams. Though this transformation fits well into our customized UML methodology, it also reveals some major differences between SA/RT and UML.

UML activity diagrams describe an *invocation hierarchy*, whereas SA/RT data flow diagrams are basically a means for *structural decomposition*. Moreover, UML activity diagrams and in particular the token concept inherited from petri nets, provide a stricter interpretation of control flow than SA/RT.

Thus, the proposed representation of intermediate-level DFDs as activity diagrams should not be read in a strict sense. Intermediate-level activity diagrams are mostly an artefact of the decomposition and provide no additional information that would be relevant for specification. However, they are included to preserve the structure of the model, including e.g. possibilities to navigate between decomposition levels and between data flows and the domain model.

The control flow concept in UML is not only stricter but also richer than in SA/RT. The higher expressiveness manifests itself e.g. in the existence of timers, events, and concurrency. This means that a completely automatic transformation of control flow will likely be inadequate. On the other hand, the higher expressiveness allows a more detailed modeling of cyclic and/or event-driven behaviors of embedded systems, and gives the chance to improve upon the existing models.

5. Conclusion

In this paper, we described the Teamwork obsolescence problem, and we introduced a transformation from SA/RT to UML as one alternative to solve this problem. The described transformation fulfills basic needs concerning practicability. In particular, we identified the possibility to automate most of the transformation as one key factor for a successful migration.

However, migration from SA/RT to UML means much more than a transformation of the existing models. Besides an implementation of the automatable transformation steps, and the application of the transformation to our models, the following points need to be addressed:

- The UML methodology has to be implemented as integral part of the software development process, including the necessary documentation according to the standards.
- The transformed model has to be shown to be equivalent to the original model. This can be done by an evaluation and acceptance test of the transformation software, plus a check of manually adapted model elements.
- A suitable UML tool has to be selected to maintain the model, and the requirements engineers have to be trained both concerning tool usage and UML methodology.
- As side-product of daily work, the existing model has to be enhanced to increase the benefit of the migration to UML. This may e.g. include improved modeling of control flows, concurrency and timing specifications.
- Finally, further steps in the software development process (e.g. architectural design, detailed design, and qualification tests) should be based on the same UML model in order to improve consistency among the process steps, and in order to enable further automation.

The efforts and consequences of such a migration project are huge, and thus a fair comparison of all alternatives to solve the Teamwork obsolescence is a necessity. The consideration must include all interests and constraints of the affected teams, such as specification, software development, testing, and service. A decision how to proceed does not only include the selection of the method, but also the selection of appropriate tools which shall then be used by all parties. In our case, this decision process is ongoing and the outcome is yet open.

References

[1] IEEE Recommended Practice for Software Requirements Specifications. 1998. IEEE Standard 830-1998.

[2] Software Considerations in Airborne Systems and Equipment Certification. 2011. RTCA DO-178C, December 13, 2011.

[3] Model-Based Development and Verification Supplement to DO-178C and DO-278A. 2011. RTCA DO-331, December 13, 2011.

[4] CMMI for Development. 2006. Version 1.2 – Improving processes for better products. Carnegie Mellon University.

[5] OMG Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1). OMG Document Number: formal/2011-08-06. (6 August 2011) by OMG.

[6] DeMarco, T. 1978. Structured Analysis and System Specification. Yourdon Press computing series.

[7] Hatley, D. J., and I. A. Pirbhai. 1987. Strategies for Real-Time System Specification. Dorset House, New York.

[8] Jilani, A. A., M. Usman, A. Nadeem, Z. I. Malik, and Z. Halim. 2011. Comparative Study on DFD to UML Diagrams Transformations. In: *World of Computer Science and Information Technology Journal (WCSIT)*.

[9] Fernandes, J. M., and J. Lilius. 2004. Functional and Object-oriented Views in Embedded Software Modeling. In: *Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS '04)*. IEEE Press, New York.

[10] Rupp, C., S. Queins, and B. Zengler. 2007. UML 2 glasklar. Hanser Verlag. 3. Edition. ISBN 978-3-446-41118-0.

[11] Gast, P., and O. Bender. 2009. EADS GUIDELINE - Using UML for Software Analysis and Design. EADS Internal Paper.

[12] Weyrath T., B. Schinnerl, F. Schöttl, and H. Schreyer. 2012. A new UML tool-based Methodology for the Software Requirements Analysis. In: *European Congress: Embedded Real Time System and Software (ERTS^2)*. Toulouse. France.

[13] Pilone, D. 2005. UML 2.0 in a Nutshell. O'Reilly. 2. edition.

[14] Estefan, J. A. 2007. Survey of Model-Based Systems Engineering (MBSE) Methodologies. INCOSE MBSE Focus Group.

[15] Farail, P., P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Cregut, and M. Pantel. 2006. The TOPCASED project: a Toolkit in Open Source for Critical Aeronautic Systems Design. In: *ERTS 2006*, 25-27 January 2006, Toulouse, France.

[16] TOPCASED The Open-Source Toolkit for Critical Systems. www.topcased.org.

Author's Biography

Thomas Weyrath is project manager in the department for Integrated Systems in the Business Area Aviation at ESG. Since 2009, he has been working in the SSC and manages the process improvement project UML. Previously, he worked in the Automotive Area at ESG as software engineer and project manager for 9 years.

Herbert Schreyer works on the aviation system development of the Tiger and NH90 helicopters since 1995. 2005 he joined the SSC where he took over the role of a system architect. Since several years he is in charge of evaluation and introduction of new methods of avionic definition.

Jürgen Sellen is senior system analyst and software architect in the department for Integrated Systems in the Business Area Aviation at ESG. He has been working in the SSC since 2011. Previously, he worked in the Telecommunications Business Unit at ESG as software architect and project manager.