III. PROVEtech:R2A – A Tool for Dedicated Requirements Traceability

A fool with a tool is still a fool
- unknown

This part describes the *traceability* tool solution PROVEtech:R2A (R2A) especially dedicated to cross the substantial gap between the requirements (i.e. *problem space*) and the design solutions (i.e. *solution space*).

The author is convinced that a successful development of SW based systems is not alone guaranteed by strict compliance to SE processes someway developed in theory, but it is at least in the same way (or maybe even higher) influenced by both the real unique constellations of projects (so called practice) and by soft factors as humans and their communications (e.g., cf. [Mu06a]).

Correspondingly, the solution proposed here tries to account for all three factors. The next chapter (ch. III.11) outlines this in more detail deriving the goals of the tool approach described here. To better illustrate the mechanisms and findings of research about R2A, the author has tried to use an accompanying case study whose basic characteristics are described in ch. III.12.

As derived in ch. III.11, two fundamental gaps must be addressed by the R2A approach. Concerning the first merely tool related gap, ch. III.13 shows how R2A intends to address this gap. In the author's opinion, the transition between requirements and design generally is difficult, because designers perform a significant mental transfer process from the requirements to the resulting solution design leading to a substantial gap between both. This second gap is the core problem. Correspondingly, ch. III.14 describes R2A's principal ideas to ameliorate the problem.

From ch. III.15 to ch. III.21 different mechanisms of R2A are introduced helping to better overcome the second gap. To achieve this goal two major strategies are employed.

The first strategy is to better support designers on documenting design information and providing means for capturing *traceability* information as a mere *by-product* of normal design activities. Ch. III.15 to ch. III.17 describe mechanisms to generally improve the design processes without yet considering the *requiremental* dimension. Basing on these mechanisms, ch. III.18 shows then how a more requirement-centered design process (see *requirement dribble process* ch. III.18.2.4) can be employed where traceability information is rather established and information on basic design decisions is rather captured as a *by-*

B. Turban, *Tool-Based Requirement Traceability between Requirement and Design Artifacts,*
DOI 10.1007/978-3-8348-2474-5_4, © Springer Fachmedien Wiesbaden 2013

product of the design activities. This first strategy part from ch. III.15 to ch. III.18 can be considered as a whole complete in itself topic where the design theory of Simon (cf. ch. I.6.2.1) dominating in the design theory about SysEng and SE is set into context with traceability needs usually expressed by *process standards* for *safety-critical* processes. These two aspects are then also combined with findings of Schön's design theory about design as situated action (cf. I.6.2.3). So far, however, the design theories considered yet rather represent a view on design assuming that the development of a design is rather a linear process of step to step actions transforming information into a design at the end. The design theories about wicked problems (see ch. I.6.2.2) and patterns (see. ch. I.6.2.4) rather suggest that the design process is not such a linear process but rather a complex nonlinear process driven by complex design decisions. In the author's opinion, design is both – in some situation design is rather a linear process of step by step transformation of information into a design, however in other situations complex decisions must be taken where the design rather emerges out in a nonlinear fashion. To cover these nonlinear aspects of design, *decision models* have been developed allowing the documentation of *rationale* behind complex decisions. These *decision models* are tightly integrated into the traceability information and the design process building a tightly woven network supporting all four design theories described in ch. I.6.2 by a unique integrated way. This second major strategy to address the second gap is treated in ch. III.19, ch. III.20 and ch. III.21.

After the ch. III.15 to ch. III.21 describe the core innovational ideas how to address the two-fold gap between requirements and design domain, ch. III.22 then shows how *traceability* information once gathered can be used in R2A for *impact analyses* and *requirement change propagation* in order to ensure consistency.

Ch. III.23 then discusses issues about embedding R2A and the R2A design processes into a higher level process environment. This starts with a description, how R2A can be used to improve supplier management. The sub chapters following then describe how this mechanism can also be used to reduce redundancies when different artifact models are crossed in a development project and how this may help to have a decoupled development of different requirement and design artifacts.

The core of R2A's innovations can be considered in the orientation on its mechanisms. Correspondingly, R2A has been designed in a way to provide an optimal support for the mechanisms. Last but not least, ch. III.24 provides an overview of the *architecture* and *meta-model* of R2A that realize the mechanisms.

III.11 Research Goals

The biggest problem of system development has always been the confusion of requirements and design.
Hatley et al. [HHP03; p.27 (*)]

Concerning *traceability*, the transition from requirements to design has been identified as one of the most critical issues as it includes a twofold structural gap:

- At first, requirement activities and design activities are usually performed in different tool environments. Correspondingly, the transition usually implies to cross a tool gap.
- More important, requirement activities deal with exploring the *problem space* and design activities deal with exploring the *solution space*. Thus a substantial conceptual gap exists between requirements and design.

A useful solution must try to bridge both gaps. The first gap seems more to be a technical issue of how to couple two tools into an integrated environment. However, as mentioned in ch. I.6 projects often use a combination of multiple *design tools* for design. Thus, an adequate solution for automotive purposes must also consider a way to couple several *design tools* in an integrated way. The next chapter will discuss the issue from the merely technical coupling perspective, but questions remain whether this gap also involves incompatible methods due to different task performed in *REM* or design.

This leads to the second mentioned gap about requirements and design discussed in ch. II.10.2. Today's *traceability* models, as seen by theory or process standards as SPICE, assume that requirements and its realizing design are connected by simple linear relationships mappable by a simple *traceability linking* schema. In reality, however, a considerable gap between requirements and design arises from the design process as it represents a creative and complex mental transfer process of a unique problem constellation into a sustainable solution that is per se difficult to reproduce. During design, designers make decisions. This gap is mentally bridged by designers by taking design decisions. Each decision involves consequences and *constrains* the *solution space* until the *solution space* (hopefully) converges to a solution fulfilling the requirements.

From the author’s perspective, the second point is the rather neuralgic issue. The author even considers that point one actually is just a symptom for the deeper

underlying problem described in point two, because tools were at first developed around the two core topics requirements and design having a higher cohesion[202].

Now, the question arises what exactly may be the cause for the second point. Considering the problems that *RatMan* solutions have with succeeding in practice, Dutoit et al. [DMM+06a; p.7] emphasize that *rationale* documentation *schemes* usually differ from the way a *rationale bearer* would structure *rationale* intuitively, thus creating "a *cognitive dissonance* that adds to the cognitive overhead that designers must cope with" [DMM+06a; p.7].

In the author's opinion, this also is exactly the issue for a *traceability* solution to address in order to help to bridge the gap. When a *traceability* solution helps designers to easily[203] capture *traceability* information as a *by-product* without imposing significant *cognitive dissonance* and bringing early benefit to designers, it is more likely to achieve better *traceability* information actually useful for projects. In this way, the promises of the *traceability* concept may be achievable.

Ch. I.6 has described four different theoretical views on design. All these views describe different – in the author's view essential – characteristics of design and its processes. However, current systems and *SW design* theories rather concentrate on design structural aspects as provided by the theories of Simon (ch. I.6.2.1) and the *pattern* theory of Alexander (ch. I.6.2.4), neglecting other – admittedly more ambiguous – theories about designers' thinking and decision making (cf. ch. I.6.2.2 and ch. I.6.2.3). The author considers improving support on designers' thinking in order to avoid *cognitive dissonance* as the *neuralgic point*. In R2A, this shall be achieved by a requirement centered modeling: Supported by a suitable methodology and a newly developed tool, the necessary work for establishing *traceability* to design shall be intuitive for designers and support their normal design work in a way that *traceability* occurs as a *by-product* of the usual design process. To achieve this, also the design theories about designers' thinking and decision making are significantly considered in the concepts of R2A.

One dedicated goal for the research was to find a tool solution whose usage in practice really brings early benefit (ch. II.10.5). As Moro [Mo04; p.26 (*)] points out in reference to modeling: "The primary decision criterion about what modeling technique or level of detail is used always is the benefit for the architect". In the author's eyes, this is correspondingly true for design *traceability*. The

---

[202] In terms of software theory, it may be said that the topics requirements and design have within each other a significantly higher cohesion within each other leading to the development of tools within their specific topics. Later, it was then discovered that coupling both may be a good idea.

[203] In this context, 'easily' means 'does not infer further *complication*' or even 'helps to reduce *complication*' (see footnote 80 (p. 77)).

benefit for the development team members must be in the center of *traceability* approaches. Otherwise, *traceability* usage will fail due to *Grudin's principle*. A symptom connected to this problem is the problem that *traceability* establishment is often performed later after design has reached a relatively stable state (see ch. I.7.2.3, comment on BP.2 and ch. II.10.5). In this way, the development team especially avoids effort for *traceability* establishment when design must be changed; however, paying the price that a lot of relevant *traceability* information is lost. Correspondingly, a major goal is to lower the burdens for *traceability* establishment and raise benefit for designers to an extent that designers rather establish *traceability* as a *by-product*.

As *traceability* is mainly established by hand [EGH+07], it is often very cost intensive and bureaucratic with little use for the development team [RJ01], [EGH+07]. The author disagrees with the idea to lower *traceability* effort by using coarser *traceability* to abstract high-level design elements (see, e.g., [EGH+07]), because feedback from practice [Pe04], [Al03] indicates the need for detailed *traceability* even at lower-level design elements, but the author agrees that *traceability* efforts must be lowered and benefits for the bearers of *traceability* information must be significantly raised. Otherwise, *traceability* will always face the *benefit problems* as all collaborative systems do in danger of failing due to *Grudin's principle* [Gr96b] (cf. also ch. II.9.4.2).

R2A offers several characteristics contributing to lowering the effort of establishing *traceability* and raising benefits for the *traceability bearers*:

- *Traceability* can be easily and fast established via *drag-and-drop* and other simple operations, by which multiple requirements can be selected in parallel to perform the operations.
- The operations adapt to how designers think and perform their design steps so that the designers can establish *traceability* information as a side-effect[204]. The same principles guide the operations that are possible to document decisions.
- All important information for a designer's situation is adequately presented in-time to support the designer's cognitive flow. Especially in-time information that is easily comprehensible supports designers in their phases of intuitive *knowing-in-action* (Schneider) by preventing that important aspects are missed. In the same way, the in-time information supports designers in their *thinking-in-action* phases of rational thinking, because the facts that are considered are directly presented. One of the most important information to

[204] As already stated in ch. II.10.5, Dömges and Pohl [DP98] emphasize that *traceability* should evolve as a side-effect of the daily development activities and not cause extra bureaucracy.

mention here are requirement information and recorded *traceability* information accompanied with information about important decisions.

- Connected to the points above, the author is convinced that a tool solution for practice should be as easy to use as possible. Theoretic research often bears theoretically sound (often in connection with strict *formality*), but complex and formal solutions (e.g., cf. Knethen's solution via *meta-models* [Kn01b]). However, in practice, developers often do not have the time to work into such complex solutions but rather prefer solutions with low entry barriers and a possibility for 'learning by doing'. This point is closely connected with the discussion about *formality* in development methods (see ch. II.9.4.2). The author tried to address these problems by providing an easy to understand, basic *skeleton* of *formal* concepts in R2A. R2A then allows enriching this *formal skeleton* with further *informal* information[205] at nearly any location.
- R2A provides a collaborative environment where all created information is automatically shared with other designers, who can immediately use and extend the information to evolve their further design.
- Operations for recording *traceability* information provide possibilities for designers to delegate requirements to other designers, who can immediately analyze and further process the requirements. In case of problems, possibilities to reissue the requirements back to the delegating designer accompanied by a note about the problem support the designers to communicate with each other.
- Short communication paths between developers and designers responsible for the model are often the decisive factor to ensure flexibility in identifying and handling necessary and reasonable model changes [Mo04; p.25]. Since all the steps of design work above are recorded, the communication actions between the designers can also happen asynchronously. This improves situations in which important designers are absent, because the other designers can delegate information (e.g., requirements or notes) to the absent designers through R2A. The absent designers are then able to consider this information and take actions after they have returned back.

[205] At minimum *informal* notes can be added on any item present in R2A (see ch. III.17.2), but also other mechanisms exist at specific locations to add *informal* descriptions, etc..

III.12 Accompanying Case Study

Every module ... is characterized by its knowledge of a design decision which it hides from all others. Its interface is chosen to reveal as little as possible about its inner workings.
Parnas [Pa72; p.1056]

In the following chapters, R2A and its features are described. To explain these features, a practice-oriented case study shows how the features interact with each other to support a good design process. Here, the basic characteristics of the case study are introduced. Later, extra chapters show the case study outcome with the features described.

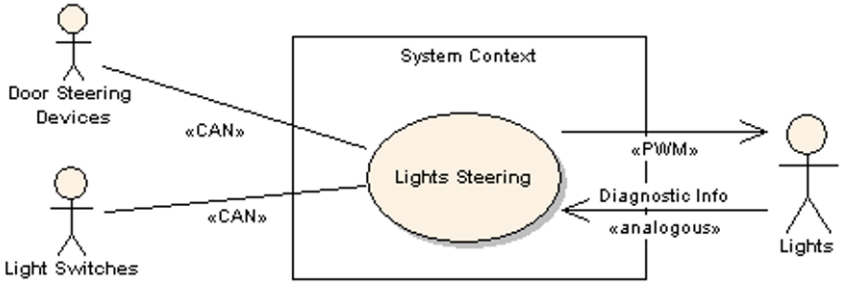

Figure 12-1 Example *use case* of the case study

The case study starts with an example *use case* (fig. 12-1) for a lights steering device in an automotive context: At first, the system retrieves different signals from the *controller area network* (CAN) bus. Then, the lights steering task determines whether some lights must be activated or deactivated. Finally, the lights are steered via *pulse-width modulation* (PWM) and diagnostic information is retrieved via analog feedback, which must be analyzed.

Fig. 12-2 shows an example *requirements specification* for the case study in IBM Rational DOORS. The requirements ReqSpec_2 to ReqSpec_6 are functional requirements describing the *use case* of fig. 12-1. It is here important to mention that requirement ReqSpec_2 is a special case as it also describes the context of the system. In this way, according to the view of Hruschka and Rupp [HR02; p.86ff] (see fig. 5-1 in ch. I.5.1), it can also be seen as a *system constraint* and thus as *nonfunctional requirement*. In practice, often requirements exist not clearly identifiable as being of one specific type.

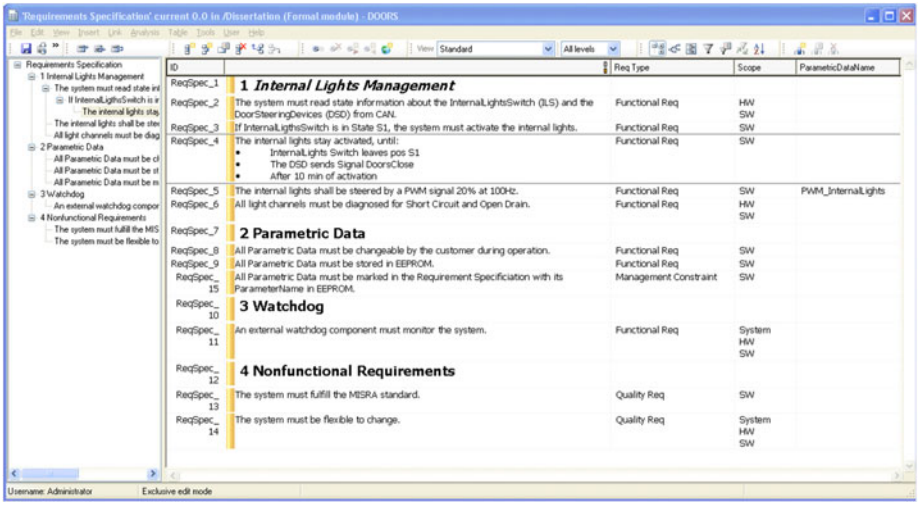

Figure 12-2 Requirements specification for the case study in IBM Rational DOORS

The items ReqSpec_1, ReqSpec_7, ReqSpec_10 and ReqSpec_12 are no requirements but just headings structuring the *requirements specification* text, whereas ReqSpec_13 and ReqSpec_14 are clearly *nonfunctional quality requirements*, and ReqSpec_15 is a *nonfunctional management constraint*.

The corresponding ECU's *SW design* outcome is shown in fig. 12-3. A high level SW architect[206] has partitioned the SW into three subsystems (the three packages *LightsManagement*, *Communications*, and *Drivers*). For each subsystem a subsystem designer determines their sub components[207].

---

206 The term high-level does not impose any specific role such as *system designer*. High-level and lower lever are rather seen in relativity to the current design task. Design activities take place in different levels of abstraction. A high-level architect is involved in designing at a high level of abstraction, e.g., determining the overall structure of an architecture, whereas for other parts of the design – e.g., for a component – a designer at a lower level of abstraction will work.

207 This example illustrates aspects of collaboration. In a real project of this size, only one designer could most probably cope with it. But in larger projects with complex application domains, a separation into several layers of design liability is common. In the automotive industry, a current trend exists to merge several previously independent devices into one powerful multifunctional device (cf. [Br06]).

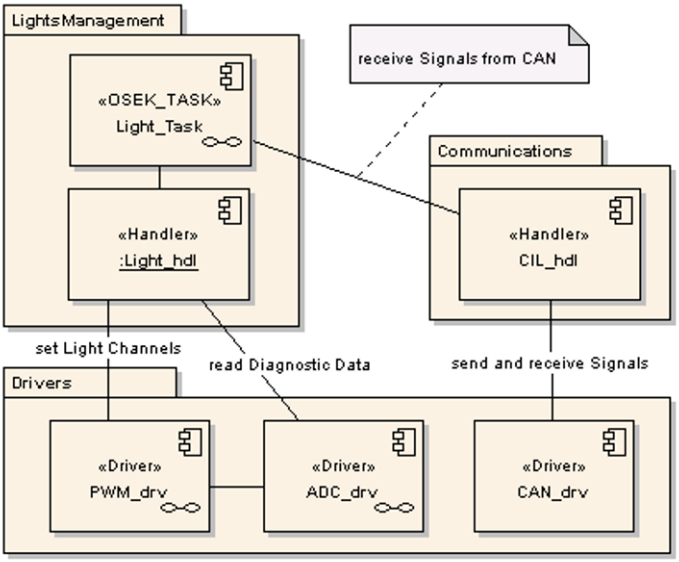

Figure 12-3 Example *SW design* for the *requirements specification* in fig. 12-2

The following project decisions have been made:

- The lights management contains an active process *Light_Task* with a complex state machine. An underlying light handler *Light_hdl* knows how to manage the underlying drivers according to the light signals to set. Both components are being developed in-house.
- The drivers (*PWM_drv*, *ADC_drv* and *CAN_drv*) are supplied by different subcontractors. Code size, performance and other parameters are highly dependent on their individual configuration. Therefore, a subcontractor manager shall monitor each driver for these parameters.
- The CIL_hdl (CIL=*CAN Interaction Layer)* depends on the types of signals relevant for the device. These settings are defined by the customer (OEM) because it affects communication.

This example case study has been chosen being as easy and clear as possible to illustrate the concepts of R2A. However, its easiness turned out to be a disadvantage for illustrating complex decision situations in ch. III.20.4. Correspondingly, in ch. III.20.4, the author deviates from the case study by referring to some further requirements and components not mentioned here in the case study. This

decision situation is then again referred to in ch. III.22.1.1 describing *impact analysis*. The author thinks this deviation from the case study is no problem, because the reader can imagine (as it is in reality) that the ECU-project involves more *use cases* and *components* than just one *use case* for internal lights steering.

III.13 Closing the Tool Gap

How does a project get to be a year late? ... One day at a time.
[Br95; p.153]

To close the gap about proper tool coupling mentioned first, R2A is designed to work as an enhancement for a *design tool*. Fig. 13-1 shows R2A in combination with the *design tool* Enterprise Architect. R2A docks its main GUI[208] window (right side) onto the main GUI window of the corresponding *design tool* generating an user experience in which both tools seem to be one tool.

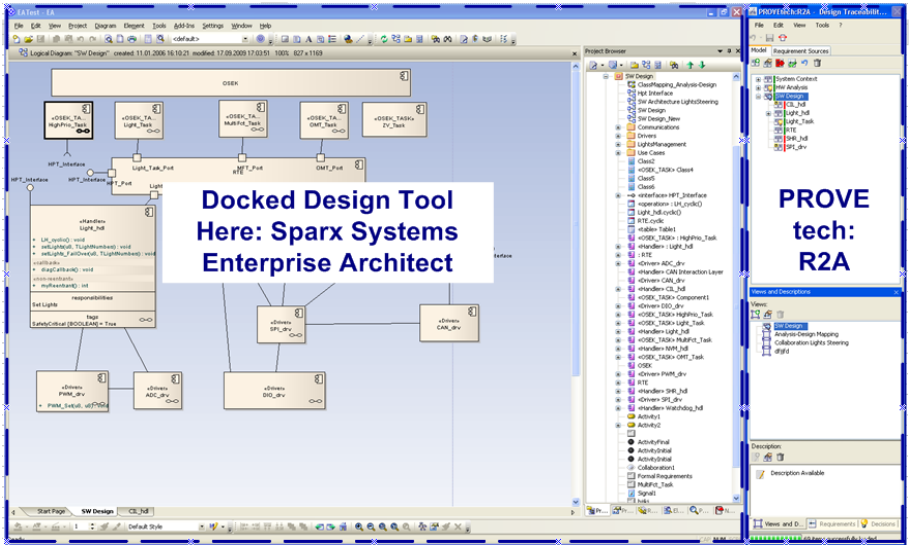

Figure 13-1 R2A in combination with a *design tool* (Sparx Systems Enterprise Architect)

[208] Graphical User Interface

From the logical architecture viewpoint (see fig. 13-2), R2A can be seen as an interlayer between an *REM-tool* providing the requirements and a supported *design tool*.
First of all, the requirements are imported from the *REM-tool* as direct representations (so called *'surrogate requirements'*) into R2A. Later, these representations can be synchronized with the requirement changes in the *REM-tool* by a regular controlled synchronization process. This is described in detail in ch. III.18.1.

All relationships relevant for *traceability* and *IA*s are consistently modeled and stored in R2A. Currently, the following relationships are considered:

- *Satisfy* relationships between requirements and *design model elements* (‘*req model dependency'*).
- *Hierarchic* relationships between *design model elements* (‘*refinement dependency'*).
- Other relationships between *design model elements* (‘*between model dependency'*).

All other not *traceability*-relevant relationships occurring in design activities are not considered in R2A but must be covered by the features of the used *design tools*.
This structure provides the following advantages in comparison to other methods:

- The *traceability* relationships between requirements and design are managed directly, whereas only some distinct model relationships (the *refinement* and *between model dependency* mentioned above) are taken into account for *IA*s. This prevents the *requirements fan-out effect* (cf. [Al03] and ch. II.10.6.2) during *IA*s.
- The synchronization between the requirements in the *REM-tool* and the surrogate representations can be performed at specific points in time and thus requirement changes between the old requirement version, present as surrogate representation and the new version in the *REM-tool* can be tracked in R2A to support a consistent change of the design to fit to the requirement changes (cf. ch. III.22).
- Besides, the surrogate representations concept allows that change works on the requirements baseline for the next release is decoupled from the requirements baseline for the current release. In this way, requirements engineers can already work on the requirements specification for the next release, whereas designers can design the system according to the requirements specification baseline of the current release in parallel. Details on this are provided in ch. III.23.3.

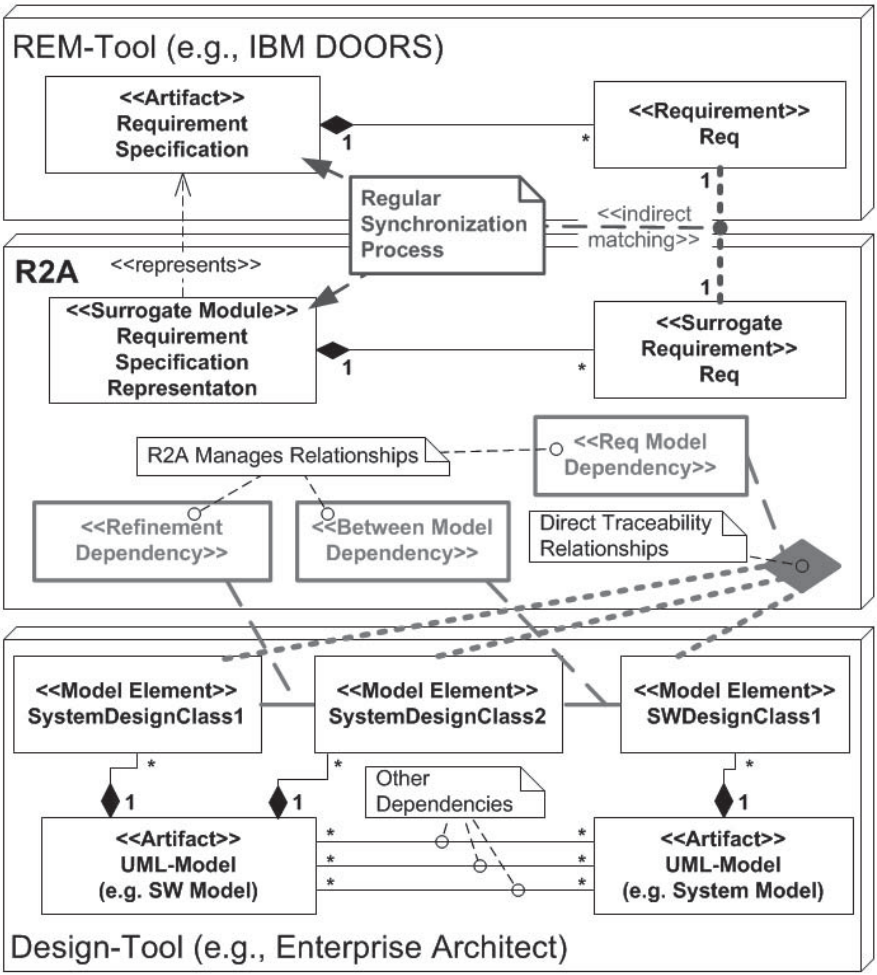

Figure 13-2 Logical structure of the R2A tool approach

III.14 Closing the Gap between Requirements and Design

Technology evolves from the primitive over the complicated to the simple.
Antoine de Saint-Exupéry

Besides the structural advantages mentioned above helping to close gap one, the R2A approach shall go beyond closing the first gap. It shall also change the way how designers treat requirements and design by establishing an intuitive process that allows to establishing *traceability* information between requirements and design as a *by-product* of the usual design activities.

According to the experiences of Moro [Mo04; p.351], it makes no sense to consider a *design model* without also considering the corresponding *requirements specification* or *software architecture documentation*. Following this finding, the author considers these items as a threefold unity. Correspondingly, R2A tries to find a solution in which all three aspects can be considered in an integrated way during design activities. The following chapters deal with the different features that try to provide a solution to better address this structural gap.

In a lot of cases, design is the result of a collaborative work between several designers working together to find a solution for fulfilling the requirements. Correspondingly, several designers must work in parallel on the same model and they must be able to easily share information. Thus, ch. III.18.2.4 shows how establishing *traceability* as part of a design process can be used as an essential means to organize collaboration and sharing contemporary requirement information between designers, working together to find a solution for all requirements. Finding good solutions essentially involves making design decisions in a collaborative manner and information about decisions must be propagated as soon as possible to all stakeholders affected by the decision. As a consequence, a design solution should also support a collaborative decision process as *rationale management systems* do.

III.15 Abstraction Layers and Abstraction Nodes

There are a lot of advantages of hierarchically organized systems and sub systems. ... If we work on a certain level of abstraction, we will be able to concentrate on this level without having to go into detail too fastly.
[HHP03; p.52 (*)]

Following the design theory of Simon (see ch. I.6.2.1), design deals with managing complexity. Central concepts for managing complexity are *abstraction hierarchies* (also called *hierarchic decomposition*) and posing different *views* on *design aspects*.

To simplify the understanding and the structure of the design, R2A emphasizes the *hierarchical abstraction structure view* (*hierarchical decomposition*) as nodes in an *abstraction tree*. Fig. 15-1 shows an example of such a hierarchical decomposition. In the further of this document, such a node is called *abstraction node* (*AN*), whereas the tree is called *abstraction nodes hierarchy* (*ANH*). An *AN* is formed out of two aspects. On the one side, it represents a *design element* usable as a symbol in diagrams. On the other side, an *AN* contains a diagram showing its internal structure composed of new *design elements* and thus a new *AN* in a more detailed abstraction level. In this way, detailing relationships (*refinement dependencies*) arise between an *AN* and its sub *AN*, in which the diagram of an *AN* contains the *design elements* (symbols) of the *AN* it is built of (composed) of.

Concerning this issue, it must be mentioned that all *ANs* at one level in the hierarchy represent one level of abstraction (or detail) in the design. This is called an *abstraction layer* (*AL*) in the further. In other words, an *AL* builds a comprehensive view on a system at a certain level of abstraction[209]. With increasing depth of an *AL*, the design gets more specific.

An *AN* is more than a node in an abstraction tree. *ANs* build the central starting point to connect to further design related information. Below, fig. 15-2 shows the conceptual characteristics of an *AN* in R2A on the basis of an example *AN* "SubSystem1" enriched with further information.

[209] $AL = \sum AN_{at\_one\_hierarchial\_level}$ – this is similar to refinements of data flows in *structured analysis* (*SA*) [De78].

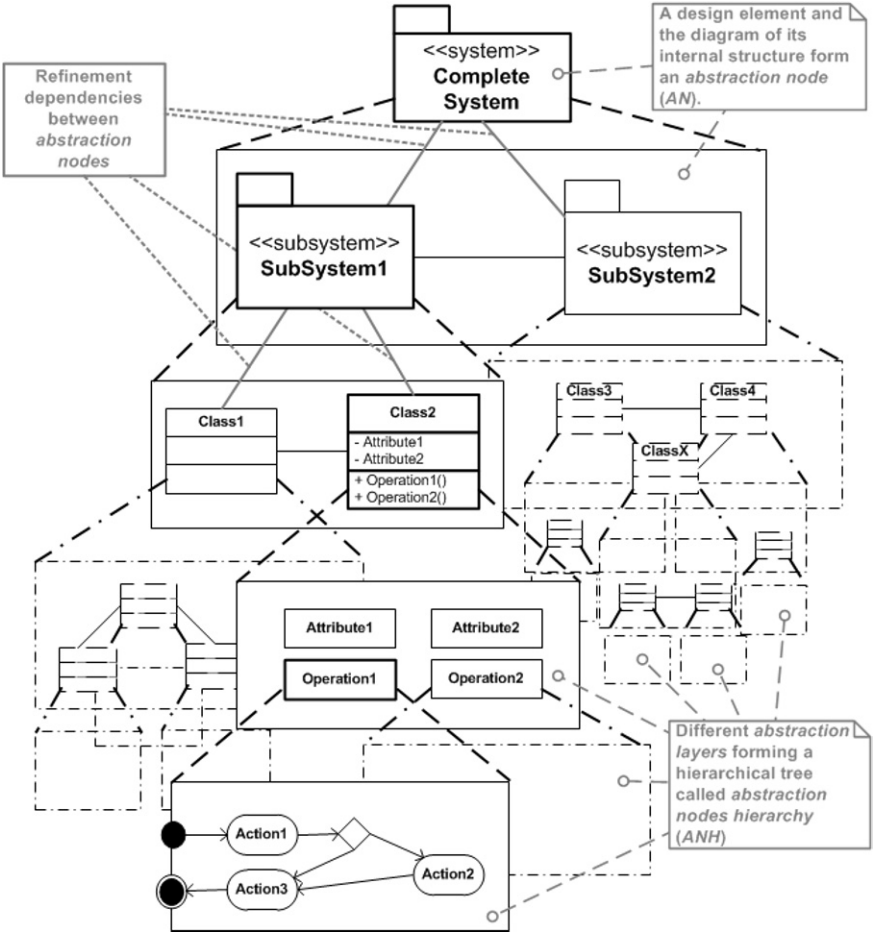

Figure 15-1 Hierarchical decomposition of a system shown as abstraction tree

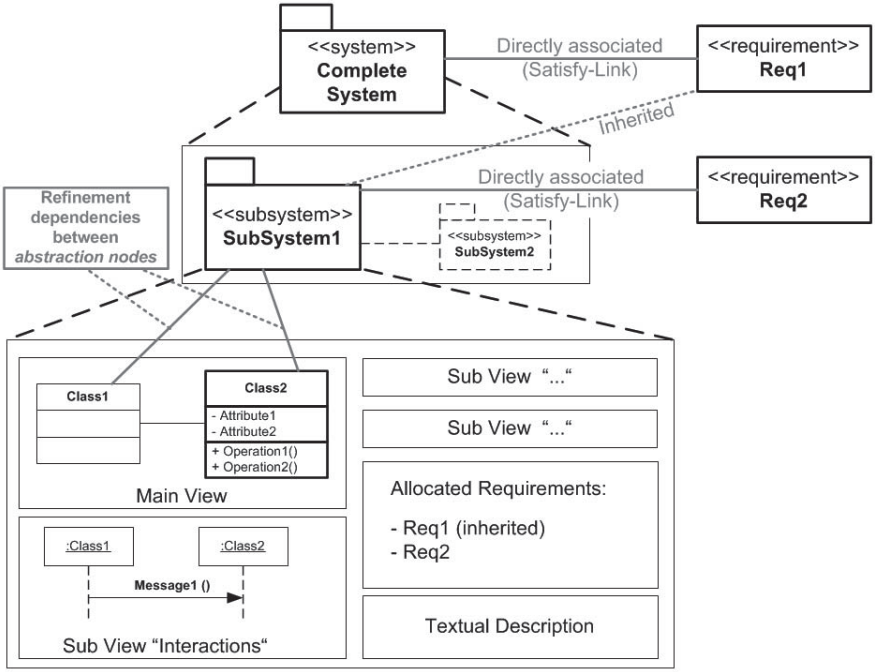

Figure 15-2 Detailed content and structure of an *abstraction node* (SubSystem1)

The goal is to present as much relevant information as possible for an *AN* and its realization. Consequently, one idea of R2A is using the *AN* concept to represent the following information to designers:

- Each *AN* consists of a representation element (symbol) that represents the abstraction node in other diagrams.
- Each *AN* has one central diagram ('Main View') as main entry point. The diagram represents a decomposition view showing how the *AN* is decomposed by sub *AN*s, in which the *design elements* of the sub *AN*s are shown in the diagram.
- Other views or diagrams can be attached to an *AN* as further views ('Sub view') to allow detailed modeling of other important aspects (e.g., dynamic behavior, concurring processes, complex behavior).
- Diagrams without further explanation can be misinterpreted. Consequently, a design must be accompanied by textual descriptions. R2A supports adding a

textual description as a rich text document for each *AN* ('Textual Description'). This allows the designers to document each *AN* separately.

- Requirements can be linked to *AN*s to indicated that the *AN* satisfies the requirements. Requirements associated to an *AN* of a higher *abstraction level* get inherited by *AN*s of lower *abstraction levels*. All these connections of an *AN* with requirements can be shown to the designers ('Allocated requirements'). The details about requirements and *AN*s are described in the following ch. III.18.
- As described in ch. II.9, important aspects about taken design decisions should be documented. The *AN* concept makes all decisions connected to structure building of the design leading to the *ANs* automatically visible to the designers. Additionally, through the history function described in ch. III.17.5, the decision history is collected. This is close to ideas of Gruber and Russel [GR96a] (see ch. II.9.4.2) to automatically capture side information on processes providing *rationale* in a way that allows to inferring *rationale* later when it is needed.

The first two points have a strong analogy to the concept of different abstractions in *structured analysis* and *design* (*SA/SD*) [De78]. Currently, the concepts of *SA/SD* have mostly been ousted by the concepts of UML.

Concerning the design language UML, a central concept is the usage of different *views* on a system under development. UML as well as UML-tools usually do not impose any demands on the definition or usage of views and their relationships. Instead all views are treated with the same priority. In UML-tools like Enterprise Architect, all elements present in a design are stored in one project repository browser. Fig. 15-3 shows an example of a project repository browser as it is provided by the UML-tool Enterprise Architect. A project repository containing all elements of a design is important for a project to have an overview of the available elements of a design. Besides the rich tool set, the relative freedom of not imposing demands for a structured approach has probably contributed to the vast success of UML in the development community. This egalitarian treatment of all design concepts, however, also makes it difficult to understand the design and the relationships between the different *views*[210] (resp. *diagrams*).

[210] Broy and Rumpe [BR07b] speak of incondite consistency between the different model views in UML (see also ch. I.6.6.1).

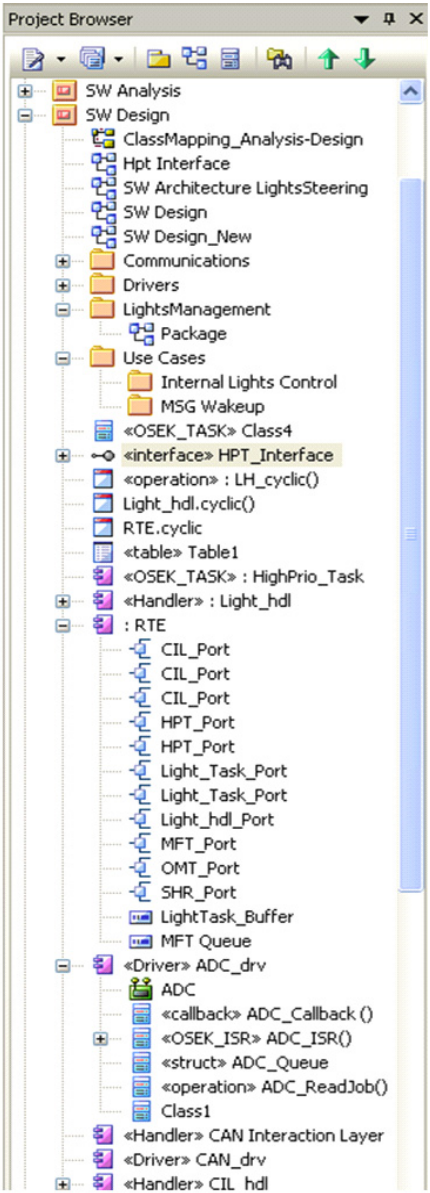

Figure 15-3 Example of a UML project repository in Enterprise Architect

This is where R2A with its *AN* concept can help designers to master complexity as it extracts and visualizes the most important structural information of a design repository. At first, R2A breaks down the information contained in a project repository into the abstraction hierarchy described in the points one and two resulting in the main view connecting the strength of the *SA/SD* concept with the strength of UML. In the next step, described in point three, each of the *AN*s in the *ANH* can contain further diagrams as further views fulfilling the concepts of view partitioning as inspired by Simon's design theory (ch. I.6.2.1). To master design complexity for designers, this structure provides an easy way to mentally structure a model with specific navigation support in two ways:

1. As main view, the *ANH* allows the designers to order the design into a structure easy to overview for a designer. This can be seen as navigation into the vertical of the *design model*.
2. To each node in the *ANH*, further associated *views* can be seen as a parallel *view* on other aspects of an *AN*. This can be seen as navigation into the horizontal of the *design model*.

Resembling accordance express the remarks from Hatley et al. [HHP03; p.47] that, if several models for a system shall be created, these models must be organized in a way orienting themselves on the relationships between the models and the system. They use the metaphor "scaffold" [HHP03; p.47]. From this perspective, R2A imposes a kind of scaffold to structure a design. Other modeling approaches as Matlab or ETAS ASCET do not provide different *views* but only have one view showing the *abstraction hierarchy* (corresponds to the *ANH*) of the design. As R2A's only required assumption about design is that an abstraction hierarchy is present, these design methods are fully compatible to R2A except for the only difference that these modeling approaches do not provide modeling of further *views*.

Nevertheless, the *AN*s concept has one major drawback: The *ANH* is a redundancy to the *design elements* hierarchy modeled in the modeling tools. This means that this information must be modeled twice and later changes must also be maintained twice – once in the modeling tool and once in R2A. Mechanisms to manage this redundancy should offer relief for these situations and explicitly prevent information drift between the redundant information. R2A offers three mechanisms:

1. As basic mechanism, a wizard helps the designers with combining *design elements* in a modeling tool to *ANs* in the *ANH*.
2. For better convenience, it is also possible to perform *drag-and-drop* operations dragging *design elements* from a modeling tool to R2A. If R2A can recover enough information about the *design elements* to fit them directly into the *ANH*, the elements will be directly added (as mentioned above, UML-

tools do not provide as clear hierarchy dependencies as tools such as Matlab or ETAS ASCET do). Otherwise, the wizard mentioned in point one opens, containing all automatically retrievable information, to which the designer only has to add the missing information which could not be automatically retrieved.

3. An automatic synchronization mechanism explicitly helps to resynchronize the *design elements* and their hierarchy in the modeling tool with the *ANH* in R2A. Before really synchronizing, the mechanism analyzes both structures and displays a synchronization wizard, where the differences and proposals for potential changes to overcome the differences are shown. Using the wizard, the designer can analyze the proposed changes for correctness or adapt the proposed changes in order to perform the changes according to the designer's intention. After the designer has approved the changes highlighted by the wizard, the synchronization mechanism applies them. The mechanism is explicitly helpful, when changes in a modeling tool shall be adapted to an already existing *ANH*, but the mechanism can also be applied to create an *ANH* from scratch using the *design elements*' abstraction hierarchy in the modeling tool. However, experience has shown that this mechanism only works frictionless for tools with a definite hierarchy (such as Matlab or ETAS ASCET), whereas for modeling tools in which the hierarchy cannot be determined definitely (e.g., UML-tools), the synchronization wizard often identifies unintended changes due to false-positive or misleading interpretations of the automatic synchronization mechanism. It is possible in the wizard to correct all these unintended changes and turn them into intended changes, but this can become cumbersome for designers. In these cases, using the two mechanisms mentioned first to create the *ANH* and then using the synchronization mechanism to synchronize later adaptations on the hierarchy may be the better alternative.

A design scaffold also is a central concern for design documentation purposes (cf. [IEEE1471], [GP04], [CBB+03] and [Ha06]). Design documentation aims at documenting design to communicate it to persons not directly involved with the design or even non project members. Besides the documentation of *design elements* and their relations documented in diagrams arising out of design, also the relations between the diagrams must be documented. This is implicitly fulfilled by R2A's scaffold (i.e. skeleton) structuring the relations between diagrams. Beyond these points, design documentation also demands a textual description of the design. Textual documentation is supported in R2A by the possibility to add a textual description to each *AN*, as described in point four of the listing about information possible to add to an *AN* (see p.274 in this ch. III.15). To ensure a certain quality of the textual design description, documentation tem-

plates can be defined and used for the documents. Last but not least, design documentation literature also demands for documenting other important information as assigned requirements and important decisions. As these points are also part of R2A, R2A is a valuable support for design documentation.

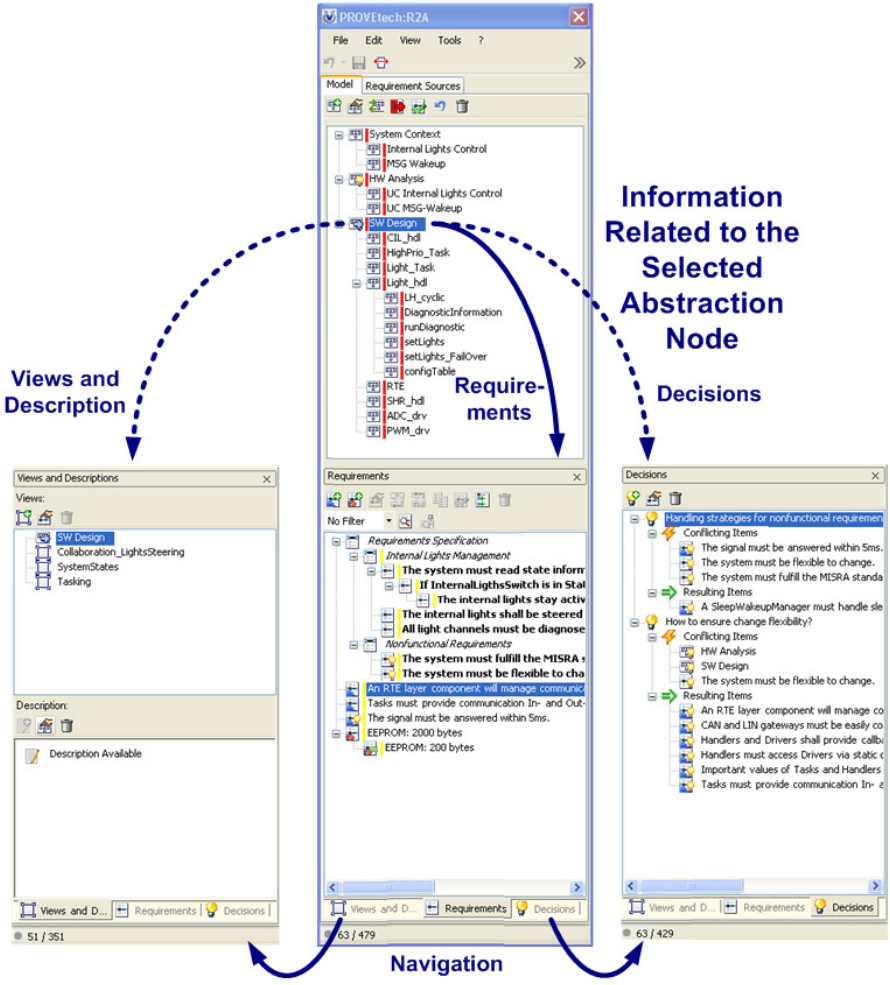

Figure 15-4 With the *AN* tree view and the tab "Views and Description"

A further point to mention here is the fact that for modeling an ECU, often several different models are used, where even several different modeling tools may be used in parallel. To reduce design complexity within such a heterogeneous model environment, R2A also provides mechanisms to manage different models and their relations in an integrated manner using the *AN* concept. Ch. III.16.2 describes this in detail.

Now, after elucidating the theory, the concrete realization of the *AN* concept in R2A is described. Fig. 15-4 shows a *design model* in R2A. In the upper part, a tree view contains all *AN*s building the hierarchical composition structure as the main view. When the user selects an *AN* in the tree view, the *AN*'s main view diagram is selected in the *design tool* and all other information related to the *AN* is shown in the lower part. This part is segmented into three tab pages (see fig. 15-4):

- The tab "Views and Description" contains a control to add diagrams as *further related views* to an *AN* and a control to add a description to an *AN* in *rich text format (RTF)*.
- The tab "Requirements" contains a control that helps to maintain *requirements traceability* information with *ANs*. This is further described in ch. III.18.
- The tab "Decisions" deals with relating important design decisions to *AN*s. This is further described in ch. III.20.

III.16 Models Crossing Tool-Barriers

Couplings between textual specification and modeling tools are immature and seldom used.
[WW02; p.22]

III.16.1 Insertion: Coupling Different *REM*- and *Modeling Tools*

In engineering practice, different *REM*- and *modeling tools* are used. The tool-based methodology proposed by the R2A project is very general and could be used by all kinds of systems or *SW design* projects. Thus, R2A is designed to be open for different kinds of *REM*- and *modeling tools* to provide flexibility in the usage of *REM*- and *modeling tools* in order to allow the usage of the best-suited tool support for a project.

To ensure this flexibility with minimal effort at maximal benefit, R2A is designed according to concepts of *software product line design* [PBG04; p.259-298]. A *software product line* is "a set of software-based systems sharing a conjoint, controlled set of product characteristics, orienting itself on the specific needs of a specific domain and being developed on the basis of a collective pool of software artifacts" [PBG04; p.262 (*)].

Here, the focus is to adapt R2A and its processes as a common development approach to fit with different *REM*- and *modeling tool* environments. In this way, R2A is not a classical *product line*, but is merely a tool framework allowing different *REM*- and *modeling tools* to be coupled. However, *product line* design differentiates a system into the invariable *product line core* and its *variation points*. The invariable core contains the constant characteristics of the systems, whereas the *variation points* define the differing characteristics of the systems [PBG04; p.276]. R2A could be differentiated in the invariant *core* of concepts described in this thesis and the *variation points* of different tool couplings to embed R2A into an integrated tool chain. Correspondingly, the *REM*- and *modeling tool couplings* have been identified as *variation points*. For each identified *variation point*, adequate strategies and design concepts to handle the variation must be found. A common problem at *product line development* is that the *product line core* is in constant danger of creeping erosion. This means that the variations along the boundaries between the *core* and a *variation point* always demand variations at parts of the *core* leading to a growing extent of the *variation point*, whereas the invariable *core's* extent shrinks (erodes) with passing time in a *product line* project.

To address creeping erosion in R2A, the main strategy for both *variation points* was to ensure strong encapsulation between *R2A's core* and its *variation points*. This is accomplished by the usage of concepts and *patterns* such as the *interface* concept, *proxy*, *observer* and *abstract factory pattern* (cf. ch. I.6.2.4).

III.16.2 Integrating Several *Modeling Tools* in a Single Model

As described in ch. I.6.6.1, often several *design tools* are used simultaneously in an automotive embedded project, due to different strengths of the different tools. Correspondingly, R2A supports to handle several *design tools* in one integrated model[211].

[211] See also Medvidovic et al. [MGE+03; p.199]: "While individual models help to clarify certain system aspects, the large number and heterogeneity of models may ultimately

Fig. 16-1 shows an example of such a model basing on the accompanying case study about internal lights control. The model starts with the *AN* "SW Design" that refers to the high-level design diagram of the software. This diagram is modeled in a UML-tool (in the example Sparx Systems Enterprise Architect). In the diagram, several *design elements* are shown, among them the elements "CIL_hdl", "Light_hdl" and "Light_Task". These elements become further *AN*s in R2A.

Due to the different roles and characteristics of the *AN*s, different modeling tools are used to model the diagrams showing the internal design structure of the individual *AN*s:

- The "Light_Task" contains a complex state machine. In order to tame the complexity, the state machine can be modeled, early simulated and then be converted to code via Matlab Stateflow. Thus, the diagram of the "Light_Task" *AN* refers to a Matlab model diagram.
- The "Light_hdl" maps abstract signal definitions used in the "Light_Task" to concrete signals according to the used HW and manages HW diagnosis functions. This involves complex algorithms that are sketched best via UML activity diagrams and then manually implemented in C. Therefore, the "Light_hdl" *AN* is also modeled best in a UML-Tool.
- The "CIL_hdl" (CAN Interaction Layer Handler) cares about managing different signals sent or retrieved via CAN. The signals are usually described in a so-called CAN matrix. A CAN matrix is often described in Microsoft Excel or a dedicated CAN configuration tool. Correspondingly, R2A could[212] refer to this application and the corresponding CAN matrix file.

Once an R2A-model is setup, where the *ANs* with their diagrams are realized in the different modeling tools, the designers can use R2A to navigate in the inte-

---

hamper the ability of stakeholders to communicate about a system. A major reason for this is the discontinuity of information across different models". As a solution, Medvidovic et al. [MGE+03] propose using a model connector concept, where relationships between models can be modeled. This model connector concept rather seems to be an extended link concept (link with different assignable properties) and seems not to be in significant practical application. Nevertheless, the model connector concept may be significantly more flexible than the functionality of R2A. On the other side, the model connector concept leaves open how these connections may be adequately visualized to provide an overview for designers. In this aspect, R2A's concept provides a clear structure, well-known to designers.

212 Currently, R2A does not support to include Excel or any other application for managing *CAN matrices*, but it will be possible similar to the support for a UML-tool or Matlab, if a coupling of the tool with R2A is implemented. In this way, this indicates a possibly promising extension of R2A's current state of development.

grated model built up from the parts modeled in the different modeling tools. For example, when a designer selects “SW Design” or the “Light_hdl” *AN,* R2A will dock to the UML-tool and show the corresponding diagram. In the case of the “Light_Task”, R2A will dock to Matlab and shows the corresponding Matlab diagram, and so on.

If a modeling tool is not available (e.g., the designer does not have a license for the corresponding tool), R2A provides a model viewer mode, where R2A shows a snapshot of the model as bitmap taken by R2A the last time a designer worked with the corresponding modeling tool. In this way, R2A provides one integrated *design model* to the designers even though different tools are used. The *AN* concept once again proves its value as the integrative scaffold.

In most cases, design is a collaborative task, where several designers must work together. Following the example above, it is very likely that the “SW Design” *AN* and its connected information is designed by a SW architect, whereas the details of the individual sub *AN*s (“Light_Task”, “Light_hdl” and “CIL_hdl”) are designed by developers being specifically responsible for their component (so-called component designers or module designers). Thus, immediate information sharing between the designers is essential. Such cases are especially important in the context of sharing information about requirements, *requirements traceability* and decisions.

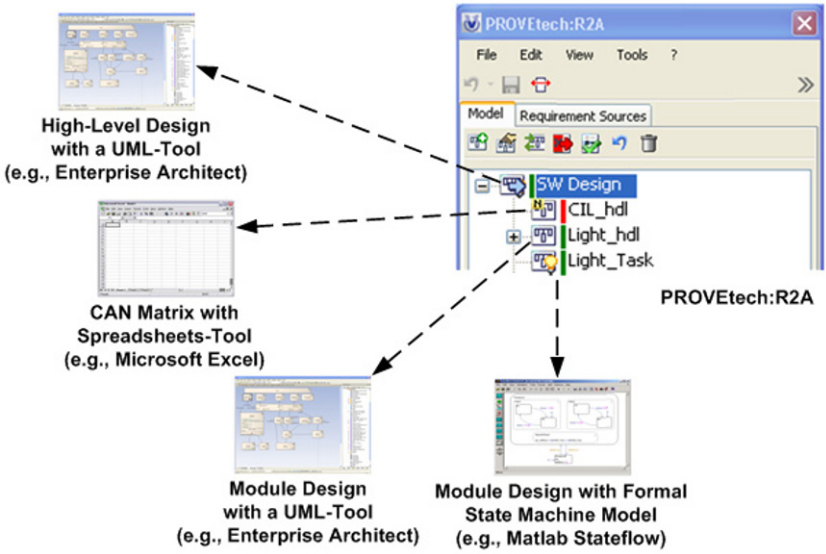

Figure 16-1 Different modeling tools integrated into one *design model* via R2A

As the following chapters describe, the *AN* concept plays the key role in connecting those information with the modeling information in a collaborative way. A possible scenario can be that the software architect makes the decision that a certain requirement must be handled by the "Light_Task". The software architect can document this decision by assigning the requirement to the "Light_Task" *AN*. R2A then immediately notifies the component designer of the "Light_Task" about the newly assigned requirement, and the component designer can immediately use the information to adapt his component design.

Details to these options are presented in the following chapter. In this context, the reader should note that all statements about information propagation between *AN*s also imply that it is possible to cross the information beyond modeling tool boundaries by the integrated model concept described here.

III.17 Basic Support Features of R2A

Design is the most demanding activity within the development cycle.
[ER03; p.34]

R2A also contains some features that are well-known in other tool environments, but the combination of these features with the innovative concepts of R2A brings interesting bonus values. In the following these features are sketched.

III.17.1 Support for Collaborative Design Tasks

As already stated above, design is usually a collaborative task. Consequentially, R2A is also construed to support collaborative aspects of design. When a user performs and saves a change in a R2A model, the change is automatically distributed and updated in all other R2A instances connected to the model.

For improving communication between users, a notes mechanism has been realized in R2A. Details to the notes mechanism are described in the next chapter. One big advantage is that it allows asynchronous communication between the users.

Later in ch. III.18.2.4, the process heuristic *requirement dribble process (RDP)* is introduced that extends the collaborative mechanisms described here to a heuristic to collaboratively find the best design solution for requirements and simultaneously documenting the *traceability* information with a history of the decision-making process leading to the solution.

III.17.2 The Notes Mechanism

Design is a collaborative task, where information sharing is essential for project success. Thus, a notes mechanism[213] provides decisive means to improve communication, i.e., reconciliation between the project members. Concerning communication, three factors must be considered:

- At first, good design lives from good (i.e. creative) ideas. Unfortunately, often creative ideas emerge from a designer's mind for particular aspects of the design, for which no specific structure around the idea has shaped yet. This means a good idea may not be immediately integrated into the current *stable intermediate form* of the design. This point appears to be closely connected to what is discussed in the course of Schön's theory (ch. I.6.2.3) about sketching as an essential activity in design. According to Goel ([Go99], [Go95]), sketching occurs at the beginning of design. Sketches often shape ideas in a kind of ill-structured nature. A notes mechanism provides a flexible, easy to use and fast way for sketching and documenting such ideas.
- R2A allows attaching these notes to any item present in R2A. This enables designers to notify other designers about their ideas. As an example, it often occurs that a designer has a good idea about the solution for a specific requirement, but it is not clear yet what part of the system will handle the requirement. In this case, the designer can attach a note to the requirement and easily sketch the idea in the note text. At a later time, the requirement gets assigned to an *AN* that shall provide the solution for the requirement. Often, a different designer will be responsible for finding the solution to this specific *AN*. In this case, this designer now can open the note attached to the requirement and retrieve a hint about the idea of the other designer how to solve the problem imposed by the requirement at best. Obviously, the example shows that the notes mechanism[214] is a means for communication between the designers inferring the advantage of enabling indirect, asynchronous communication[215] between the designers at their collaborative work.
- Additionally to sketching ideas, designers sometimes also identify interconnections between parts of their design and requirements that are difficult to express in normal design documentation. For these cases, R2A's notes-

---

213 See fig. 17-1 (p.289) for a description of the user interface implementation in R2A

214 Here, in combination with the *requirements traceability* mechanisms described in ch. III.18.

215 For detailed information on implementation, advantages, and disadvantages of synchronous and asynchronous team communication mechanisms in collaborative environments refer to [GK07; p.103-114].

mechanism also allows attaching several items to one note helping designers to document these interconnections (and perhaps also sketching an idea how these interconnections may influence the further design).

- Another source of communication problems between designers are often interdependencies between the designers' work. For example, it is possible that a designer cannot design a solution for a requirement because another designer has not yet designed a solution for another part of the design (e.g., another *AN*), on which the solution of this requirement bases. In this case, the notes mechanism allows the designer of the requirement to apply a note on the requirement and the *AN* not yet fulfilling the necessary design. In this note, the designer can sketch what the other *AN* misses so that he cannot find a design solution for his design problem. Through this, the designer of the other *AN* retrieves then the information that he must find a design solution for the specific problem the other designer's work depends on.

As a side-effect, such notes also provide valuable information when later changes on the design must be maintained at later phases. In this way, notes also provide weak support for *traceability*. However, it must be mentioned here that a few chapters later a significantly stronger support for *traceability* with slightly overlapping possibilities is introduced. This mechanism deals with describing design decisions for problems and their consequences in a traceable way. It is highly possible that some notes sketching ideas about a problem, later become a documented decision.

III.17.3 Extensibility: XML-Reporting and User Tagging

No ever so big tool development effort can anticipate all user needs. This is especially true for all usages of once gathered information. To provide additional flexibility all gathered model information of R2A can be exported to XML and developers can add individual user tags in free text form. This allows organizations to reuse the R2A information in other tools or to develop own special purpose tools using the information for their specific needs.

Experiences with pilot users of R2A revealed that this is especially important for extended information analysis and specific reporting to management. Through the user tags[216] it is possible to add additional *meta-information* on R2A items which is often important to steer information analysis and reporting.

---

[216] See fig. 17-1 (p. 289) for information on how *user tagging* is integrated in R2A's user interface

For the future, the currently discovered reporting needs can be further integrated into R2A as a standard reporting concept, however the mechanisms described here further allow users to quickly check out and adapt new promising uses[217] of the gathered R2A information.

III.17.4 Unique Identifier Support for any Item in R2A

Any item created in R2A automatically receives a unique identifier. As described in ch. II.10.4.2.1, the unique identifier concept is essential to allow textual references as linking is not always possible. In this way, items can also be textually referenced in other development tools, where no direct connection exists. Thus, e.g., in the case of R2A any item in R2A can be referenced in a textual change proposal issued in a *change management* tool by simply writing the unique identifier of the R2A-item in the change proposal's text. To ensure that R2A's identifiers are unique R2A uses the GUID[218]-mechanism provided by the Microsoft Windows operating system.

III.17.5 Evolutionary *Traceability* – Recording History and Baselines

As ch. II.10 has exposed, *traceability* also involves recording the evolution history in project development. This means that all operations performed in R2A must be comprehensible in retrospect. Correspondingly, R2A provides a *history mechanism* to record the history of every operation performed in R2A accompanied by information about the performing user and a time-stamp of the time when the operation has been performed. This history information can be regathered any time by the users if needed (see fig. 17-1 (p. 289)).

---

[217] One issue regularly showing up at discussions with potential users is the idea to integrate the information with project planning information to measure accuracy of project planning and getting a deeper insight about the real status of a project.

[218] GUID stands for General Unique IDentifier and is a well-tested mechanism in Windows ensuring that each generated GUID is world-wide unique (e.g., Microsoft Windows heavily relies on the mechanism to ensure that system internal interfaces or services have a unique identifier).

R2A's history mechanism also provides a possibility for users to save a certain status of the model as a fixed version baseline[219]. Any baseline can be any time reopened in a *baseline viewer* to analyze the status of the design at a certain point in time. Additionally to all information gathered in R2A at that time, such a baseline also records snapshots of all diagrams modeled in the connected *design tools*. Thus, when a baseline is opened in the baseline viewer, also the state of all modeled diagrams at that time can be viewed and analyzed. This is especially helpful to provide an overview over a certain baseline state when more than one modeling tool is used in a model.

III.17.6 The Properties Dialog

For any item present in R2A, a properties dialog shows its properties, evolution history and attached notes. Fig. 17-1 shows the properties dialog of the *AN* "SW Design". On the left, the properties of the item are shown. This dialog varies corresponding to the item type because each item type has different properties. E.g., a requirement mainly has the requirement text as properties, whereas an *AN* type has the properties shown in fig. 17-1. Only the last property "User Tags" is an exception because this property is shown for any R2A item as it enables the user tagging mechanism described in ch. III.17.3.

Through the tab button "History", the user can navigate to the history tab shown in the middle of fig. 17-1. The history tab is segmented in an upper part showing different version entries of the item (here two). In the part below, the differences of versions selected in the upper part are highlighted (cf. ch. III.17.5).

Via the "Notes" tab shown at the right side, the user can add notes to the item according to the notes mechanism described in ch. III.17.2. This tab is divided into three sections. The lower right section contains an overview of all notes attached to the item. In the upper section, the selected note's text can be viewed or edited. All items to which the note is attached are displayed in the lower right section. To attach the note to other items, the designer can *drag-and-drop* the items in the lower right control.

---

[219] "A baseline is a configuration assembled and verified that it is considered as stable and works as referring point for further development. A release is a baseline defined for delivery to the customer" [LL07; p.521].

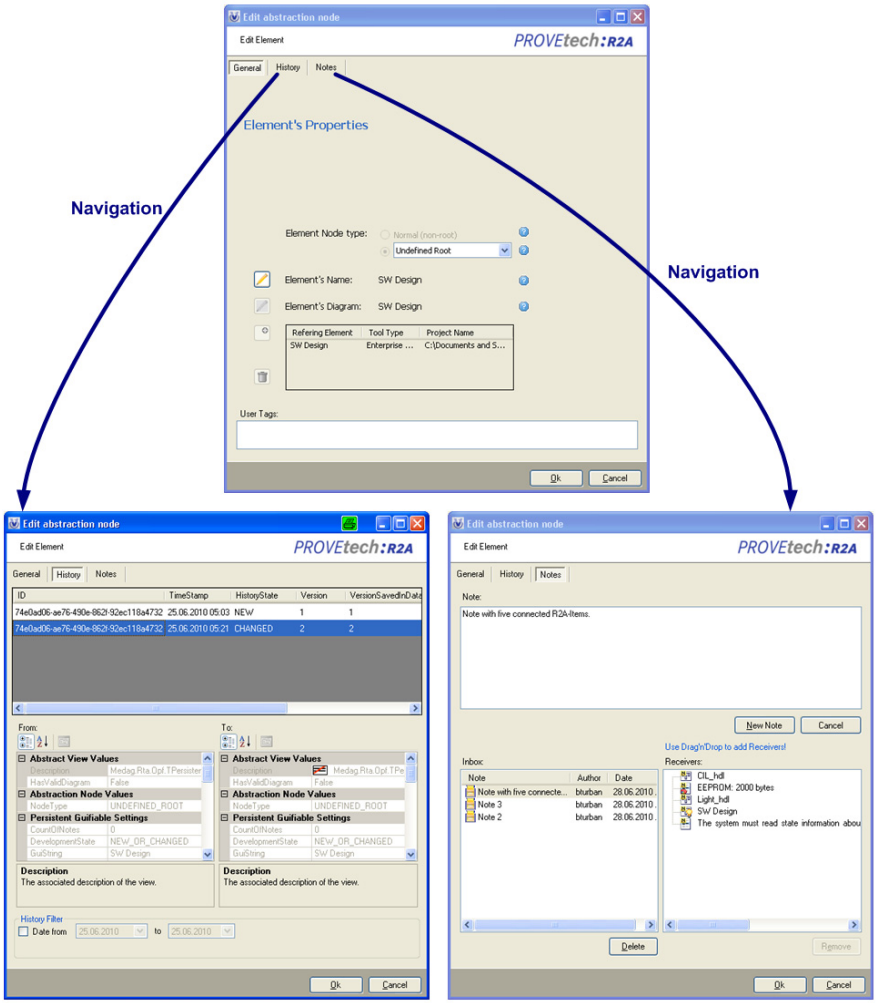

Figure 17-1 The properties dialog in R2A

III.18 Requirements and Requirements *Traceability*

If the language is not right, the spoken is not the meant.
Confucius (*)

In the following, R2A's handling of requirements and how *requirements traceability* is established is described. Both points have a slightly different meaning. Correspondingly, the first sub chapter discusses managing requirement sources and how basic *requirements traceability* can be established with R2A. The later ch. III.19 and ch. III.20 then discuss how basic *requirements traceability* can be extended to improve quality of *traceability* information and to improve problems of SPICE in connection with *traceability* (see ch. I.7.3.2).

Afterward, ch. III.22 discusses how all the collected information can be used to predict effects of requirement changes and how changes can be consistently inferred into a R2A model in order to avoid degradation of *traceability* information. Finally, ch. III.23 discusses how R2A can be integrated in a more general process context to manage suppliers or to manage decoupled development for different versions.

III.18.1 Managing Requirement Sources

At first, it should be mentioned that R2A is not intended for the usage as a complete *REM-tool* like IBM Rational DOORS. Thus, R2A does not concentrate on features for requirement elicitation, documentation or management. Instead R2A is assumed to be a broker, who can retrieve requirement documents from different sources. In this way, different requirement documents and their sources can be managed in the “Requirement Sources” part of R2A (see fig. 18-1).

Here the different documents containing requirements from a source can be managed. These documents are called in the further *requirements source document (RSD)*. An open *RSD* can be seen in fig. 18-2.

Currently two[220] different types of *RSD* exist:

- Documents originating from an *REM-tool* (*requirements specification* items),
- Sources that can be manually managed to allow documenting information otherwise neglected;

[220] Actually, the figure also contains the items “Decisions”, “Design Constraints” and “Resource Constraints”. These items are not *RSDs* in the sense discussed here. These documents are rather containers for all items discussed in ch. III.19, ch. III.20, and ch. III.21.

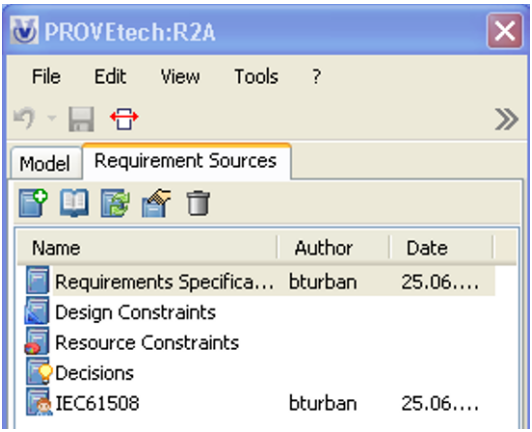

Figure 18-1 Managing different requirement sources in R2A

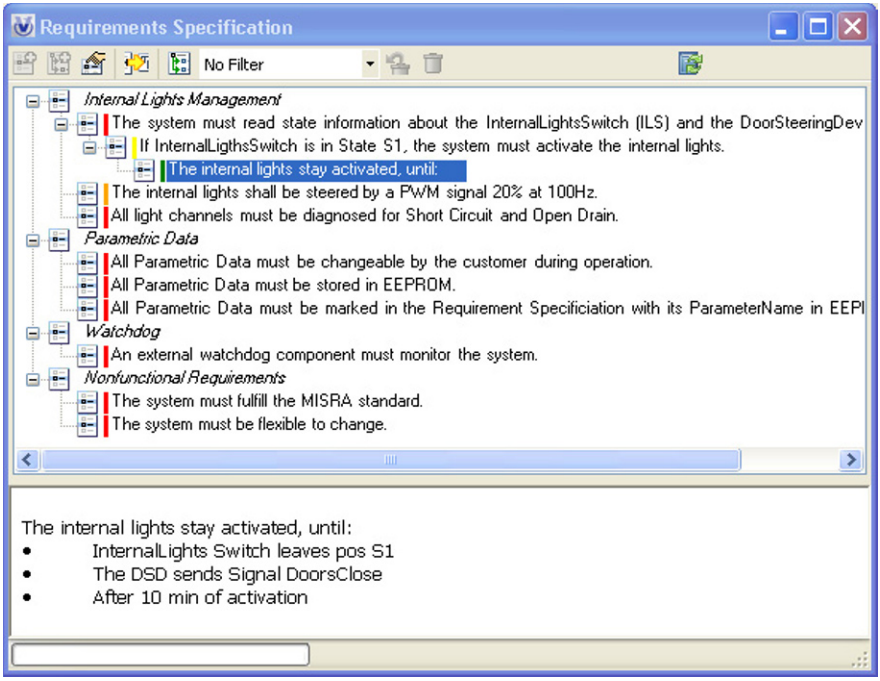

Figure 18-2 *Requirements source document* synchronized with IBM Rational DOORS

Point one refers to requirement documents that are edited and managed in an *REM-tool*. In this case, the *REM-tool* functions as data source from where the available requirements can be continuously synchronized[221]. Fig. 18-2 shows a *RSD* being synchronized with the case study's requirement document managed in the *REM-tool* IBM Rational DOORS shown in fig. 12-2 (ch. III.12). A filtering mechanism allows importing only the requirements from the *REM-tool* that are important for the *design model* managed in the R2A project. For better orientation of the designers, *REM-tool* items not included by the filter criterion but containing items as sub items that are included by the filter criterion are imported into R2A as headings. Fig. 18-2 shows items "ReqSpec_1", "ReqSpec_7", "ReqSpec_10", and "Req–Spec_12" as headings in italic.

Once an *RSD* has been synchronized with the *REM-tool*, the present requirements can be related to *design elements* via the *traceability* operations described in the following chapter. Headings are only there for structuring the document and have no further meaning. This means, none of the *traceability* operations described in the following chapters can be performed for headings.

If requirements are changed in the *REM-tool* then, continuous synchronization procedures allow the requirement changes to be introduced into the design in a consistent way. This is described later in ch. III.22.

Point two offers additional freedom for the users as easy and fast way to document information that would otherwise be omitted. As Hörmann et al. [HDH+06; p.93] emphasize, many requirements have other sources (e.g., company-internal requirements deriving from product politics or the product *architecture*). As outlined by ch. I.7.3.2 and ch. III.19, the author demands to consider negotiability as a criterion for *requirements specifications*. In the author's opinion, the requirements specification should only contain the requirements that must be negotiated with the customer. Company-internal requirements[222] (not

---

[221] The coupling of *REM-tools* is much looser than the coupling of modeling tools because R2A docks its user interface directly to a modeling tool, whereas *REM-tools* only function as data source. Thus, the interface for *REM-tools* is not as complex as the interface for the modeling tools.

[222] The probably most often occurring company-internal requirements are what the author calls internal management requirements. In most cases, internal management requirements might probably deal with ensuring cost efficiency and ensuring monetary benefit. Parts of these requirements have *impact* on design. For example, management can require using COTS (components off the shelf) components or components originally developed in other projects to avoid development effort. A significant problem for design often having significant influence on the design outcome is then how to integrate these components with the other parts of the design. Including such requirements in an extra requirement document helps to separate real requirements from the customer

originating from the customer) could thus be stored in a second requirements specification, or in more pragmatic processes, just be documented in a manually managed *RSD*, or derived from former design decisions (discussed later in ch. III.19 and ch. III.20).

Another scenario to consider here is that *requirements specifications* often refer to industry standards to be fulfilled. In this case, often the requirements imposed by the standard are not directly referenced in the *requirements specification* because these requirements are fixed. Now, the feature to manually write down requirements would allow defining a requirement source referring to the standard (e.g., IEC 61508 in fig. 18-1). In this document, the designers can now note down requirements for the design derived from the IEC 61508 standard.

A manually managed *RSD* looks like and is treated in the same way as a synchronized *RSD* shown in fig. 18-2, except that its containing requirements can be edited in R2A. The handling of the requirements described in the following is also the same as for synchronized requirements.

As described in ch. II.10.4.2.2, requirements can be managed via decomposition hierarchies and decomposition hierarchies are the state-of-the-art management technique offered by *REM-tool*. Correspondingly, *RSDs* originating from *REM-tools* take over the decomposition hierarchy in the *REM-tool*. Fig. 18-2, e.g., shows the requirements in a hierarchic tree directly taken over from the hierarchic decomposition in the IBM Rational DOORS document shown in the left column of fig. 12-2 (ch. III.12). In manually managed *RSDs*, the users can manually arrange the requirements' hierarchic decomposition in R2A.

III.18.2 Establishing Requirements Traceability

Before going into R2A's support for *traceability* establishment, some preliminary considerations shall lead to a better understanding of the ideas.

First of all to mention, different *traceability* models have identified different relationship types between requirements and design. As discussed in ch. II.10.4.2.3, e.g., SysML differentiates between *<<DeriveReqt>>*, *<<Satisfy>>*, *<<Verify>>*, *<<Refine>>*, *<<Trace>>*, and *<<Copy>>* relationship types [SV08], Ramesh and Jarke [RJ01] identify four different relations '*allocated to*', '*satisfy*', '*drive*' and '*addressed by*' in their *high-end traceability model*, other re-

---

from requirements originating somewhere in the developing organization. This already reflects an idea further discussed in ch. III.19 that requirements must be separated according to their negotiability. Surely, requirements originating within the developing organization are easier negotiable within the developing organization than requirements originating from the customer building the contractual basis of the development.

search as, e.g., [Wi98] even surfaced more relationship types. The probably most usual link type is the *'satisfy'* type, indicating that a requirement related to a *design element* is satisfied by the *design element*. In fact, the author believes that, e.g., the three types of SysML are only a little more special variation of the *'satisfy'* link type, as it is the same case for the *'allocated to'* [223] and *'addressed by'* [224] *link types* in the *high-end traceability model* of Ramesh and Jarke[225] [RJ01].

In the context of this research, the question of the relationship type has been left open as research concentrated on an efficient way to establish significant *requirements traceability* providing support for helpful *IA*s. In the author's practical experience, the question whether a relationship has been recorded and thus an *IA* identifies a possible *impact* has higher priority than the correct kind of a relationship, because relationships identified by an *IA* will still be interpreted by the developers leading to the exclusion of false-positive relationships, whereas relationships not found may just never come to the minds of the interpreting developers. In this way, R2A leaves the question about a particular kind of relationship open by using the term a requirement is *assigned to* a *design element* which equally corresponds to a *satisfy-link* type. Later, if usage of R2A in practice proves the necessity to further differentiate different kinds of recorded relationships, the R2A approach can be easily enhanced by a feature to provide more specific relationship type information.

Following Simon's design theory (ch. I.6.2.1), the design process is a continuous decision process, where a lot of the decisions are performed on the basis of the requirements. R2A directly supports this decision-making, because R2A directly shows these requirements to the designers that are important in the design situational context.

Another issue to consider is that continuous *refactoring* of the design structure is necessary due to bounded rationality, arbitrary complexity and Berry's findings about the need to restructure modularization [Be04; p.56], (see ch.

---

[223] Definition of '*allocated to*': “REQUIREMENTS are ALLOCATED to COMPONENTS that are supposed to satisfy them” [RJ01; p.73].

[224] Definition of addressed by': “Several focus groups mentioned that it was important to identify the FUNCTIONS PERFORMED BY COMPONENTS. These FUNCTIONS are typically traced to the functional REQUIREMENTS explicitly identified in requirements documents.” [RJ01; p.74].

[225] The '*drive*' relationship only expresses that requirements drive the design (“REQUIREMENTS DRIVE DESIGN, that are often BASED ON MANDATES such as STANDARDS or POLICIES or METHODS that govern the system development activity” [RJ01; p.73]). Correspondingly, the author is not even sure whether this is really intended as a link type by Ramesh and Jarke. Instead, the author considers the '*drive*' relationship as a conceptual metaphor for the design process.

I.6.2.1.2). Accordingly, it must also be possible to easily *refactor traceability structures*. Today's current state-of-the-art methods of relating requirements are not very flexible for changing requirements assignments. As an effect, designers often perform their design process first to such an extent that the design has shaped to a relatively fixed state and then establish *traceability* information.

This has the effect that the requirements are the basis for a lot of performed decisions, but on the other side the connections between requirements and design are documented afterwards. In this way, a lot of information on certain decisions is lost[226]. As described in ch. II.10.4.3.1, capturing and description of traces should orient themselves on the way the traces occur in the real world. Otherwise, a mismatch between reality and the actually captured information occurs significantly diminishing the quality of captured information [Pi04; p.104]. In R2A, all these issues are achieved by the *requirement dribble process* heuristic described in ch. III.18.2.4.

Taking into account Schön's *Theory of Reflective Practice* (ch. I.6.2.3), most design decisions are taken in an intuitive, non-reflective state of *knowing-in-action*. Former experiences and *tacit knowledge* (see ch. II.9.4.2) are important factors in this state. In this phase, tools must not interrupt the cognitive flow of the designers (see Schön; ch. I.6.2.3). Since R2A's *traceability* concept bases on the *ANH* concept, the R2A's *traceability* operations do not produce a *cognitive dissonance* for designers, thus establishing *traceability* as a *by-product* should not impose significant barriers for designers even in their *knowing-in-action* phases.

In summary, the real value of gathered *traceability* information mainly depends on the following criteria (see ch. II.10):

- Most *traceability* information must be recorded manually. Thus, the efficiency of how *traceability* can be established is crucial. This means that the effort for *traceability* must be outweighed by the reduced efforts and the higher quality, reached through improved *IA* and change processes.
- Accurateness of the *traceability* information is decisive. Approaches that establish *traceability* after the design process involve the danger that certain *traceability* information is not recorded. Thus, *traceability* should be established as a *by-product*.
- Besides efficiency itself, it is a central issue that the process does not interfere with the designers' way of thinking.

---

[226] For details see for details ch. I.7.2.3 description to ENG.3 BP.2, where it is described that allocations of requirements to design are often not possible at first because important design decisions are missing.

- On the other side, designers must perceive enough benefit for themselves because otherwise they will only record insufficient *traceability* information. One benefit can be the improved communication and collaboration between designers as, e.g., R2A offers with the *requirement dribble process* heuristic (cf. ch. I.18.2.4).
- As, e.g., ch. II.10.6.2 outlines, *traceability* information should be detailed (go deep into a *design model*) to achieve good results. It should rather be recorded directly than derived from other information such as relationships within a *design model* with other purpose because the manifold meanings of these non-*traceability*-specific relationships rather lead to a *requirements fan out effect* during IAs (ch. II.10.6.2).

To ensure these criteria and thus to ensure that the recorded *traceability* information brings a real practical benefit to projects, R2A is designed to be embedded into a process specifically addressing these issues. The following sub chapters illustrate the core concepts employed to achieve this. However, the real implementation of such a process in practice requires substantially flexible processes due to the complex connections involved in design processes. Thus, a dedicated goal of this documented research also was to find the optimal, necessary process set for these criteria, where additionally maximal flexibility to adopt processes to project specific needs is possible. In other words, the process sketched here is proposed as a possible way to use R2A, but the offered operations used in a process can also be used to perform different design processes.

Last but not least to mention, this chapter only shows mechanisms for general improvements for rudimentary *traceability* as demanded in today's *traceability* theory and process standards (e.g., SPICE). Then, in the next ch. III.20 and ch. III.21, this rudimentary *traceability* information is extended by decision models allowing much richer *traceability* information taking more complex design decisions into account to be recorded.

III.18.2.1 *Traceability* Operations in R2A

In order to prevent disturbing designers during their *knowing-in-action* cognitive phase, but nonetheless to help to document *traceability* information, R2A aims to lower the burden for documenting the traces as soon as they occur. In this way *traceability* more or less emerges as a *by-product* of the design process.

To address this point, the R2A's *traceability* approach has five key characteristics:

1. The approach takes advantage of the *AN* concept basing on the abstraction hierarchies principle strongly resembling the designers' way of thinking (cf.

ch. III.15). An approach basing on this principle, thus easily fits into the cognitive processes of the designers. If an approach does not really match with the designers' way of thinking, the designers will have to bridge the cognitive gap between their thinking and the thinking required by the approach. This would significantly disturb the designers in their *knowing-in-action* phase and therefore would increase the usage barriers for the approach.

2. Design involves processing of an extended amount of information leading to the extended complexity to be managed during design. Following Simon's theory (ch. I.6.2.1), the *abstraction hierarchies* principle addresses taming the complexity of the information produced during design. Another complexity source to be tamed in the design process is the multitude of requirements influencing the design. R2A here provides a simple answer: Only show what is relevant in the design situational context. Again referring to point one, the *AN* concept is used to set up the situational context. Fig. 18-3 shows a design situation in R2A, where the designer has selected the *AN* "SW Design". Beneath the *AN* tree view, now the tab "Requirements" is opened showing the requirements assigned to the *AN* "SW Design". In ch. III.18, the used mechanisms, and *GUI* controls with its representation features are discussed.
3. Recording *traceability* information when the traces occur but not disturbing the designers, involves that *traceability* information must be maintained in an easy and fast manner. R2A achieves this by offering an establishment of *traceability* information via *drag-and-drop* operations. As illustrated by the arrows in fig. 18-3, principally three different *traceability*-relevant *drag-and-drop* operations are possible. Via possible *multi-selection* of items in R2A, all *drag-and-drop* operations can be performed for several requirements at the same time, making the *traceability* establishment process more effective. Again, the *AN* concept appears as useful for providing central orientation to all three *drag-and-drop* operations. Operation "1.)" allows assigning requirements from the requirement source document (described in the chapter above) to any *AN* in the *AN* tree view, whereas operation "2.)" allows assigning the requirements to the currently selected *AN*. As also described above, design must also allow easy *refactoring*. In this course of action, other components than previously intended may become responsible for a requirement. Thus, requirement assignment must be changed from the formerly responsible component to the now responsible component. To easily make this possible, operation "3.)" allows reassigning requirements from the currently selected *AN* to any other *AN*. In the course of *refactoring*, it can also be evident that a requirement may just also have influence on another *design element*, but the element shall still be handled by the currently selected *AN*. In this

case, the operation “3.)” accompanied by pressing the 'CTRL'-key just allows copying the assignment information to the other *AN*, but the assignment information of the currently selected *AN* stays untouched.

4. Requirements can significantly differ in its influence on design. *RE* theory refers to this notion by distinguishing *FRs* from *NFRs*. R2A provides a concept to characterize the influence scope of requirements in a more fine-grained manner. Again, the *AN* concept builds the basis for this concept further described in ch. III.18.2.2.
5. Last but not least, Simon described the phenomenon that design usually evolves from one s*table intermediate form* to another (ch. I.6.2.1). This means design usually not emerges in a kind of big-bang process but more in an evolutionary process, where design reaches stable states forming the basis of evolution to the next stable state. The R2A approach takes this into account by proposing a process heuristic called the *requirement dribble process* described in ch. III.18.2.4.

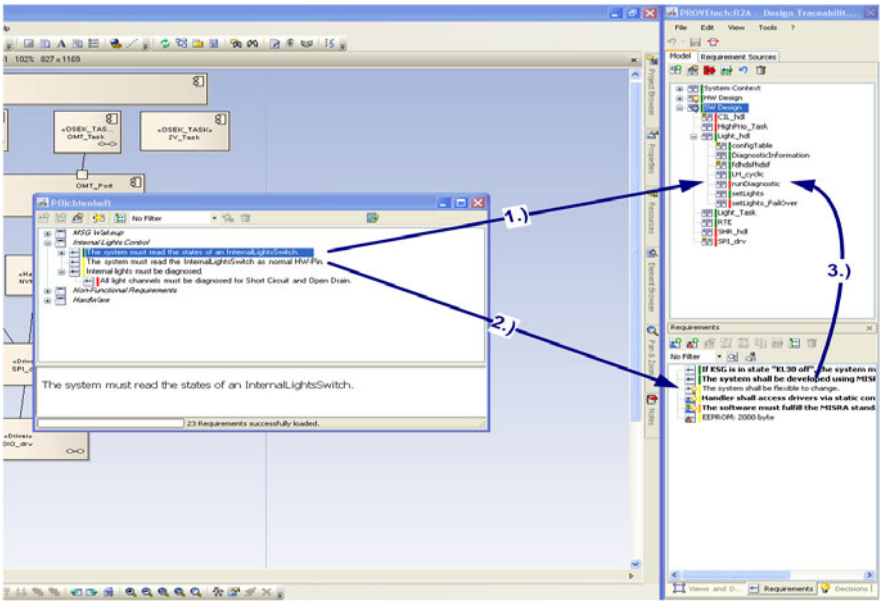

Figure 18-3 Ways of establishing *requirements traceability* via *drag-and-drop* in R2A.

In opposition to the *knowing-in-action* cognitive state, Schön has discovered that designers also switch to a cognitive state he termed *reflection-in-action*. Designers usually switch to this state when they step into a problem they cannot handle by their usual tool-set of internalized everyday problem solving experiences and knowledge. In this state, concrete rationally gauged decisions on a usually very difficult problem. In the author's view, such problems can be seen as what Rittel's design theory terms as *wicked problems* and the decisions taken to solve these problems often have drastic impact on the further outcome of the design. Correspondingly, here is the point where decision documentation and *RatMan* concepts can provide significant support to record this information. As ch. III.20 will further outline, this collected information also has strong importance for *traceability*.

III.18.2.2 The Requirement Influence Scope (RIS)

As shortly discussed in ch. I.6.2.1, strictly modularization-oriented compositional structures are again softened by design theories about architectural aspects, *cross cutting concerns* [CRF+06] or *nonfunctional requirements*. What this actually expresses is the phenomenon that not all requirements can be tamed by confining them in one module. Instead some requirements are fulfilled as a consequence of collaboration between several modules, by *architectural aspects*, *architectural styles*, *patterns* or other techniques acting on a wider scope than a single module. In order to provide meaningful *traceability*, these situations must be taken into consideration. For these situations the author will use the term *requirement influence scope (RIS)*.

Due to the *knowing-in-action* cognitive phase, an easy way to define and manage a requirement's *influence scope to design* should be possible.

Again, the *AN*s concept provides a valuable aid: If a requirement is assigned to an *AN*, all sub *AN*s beneath inherit the responsibility for the requirement. The idea behind this can be described that all *AN*s at the lower level must work together or at least share some common concern together to fulfill the requirement.

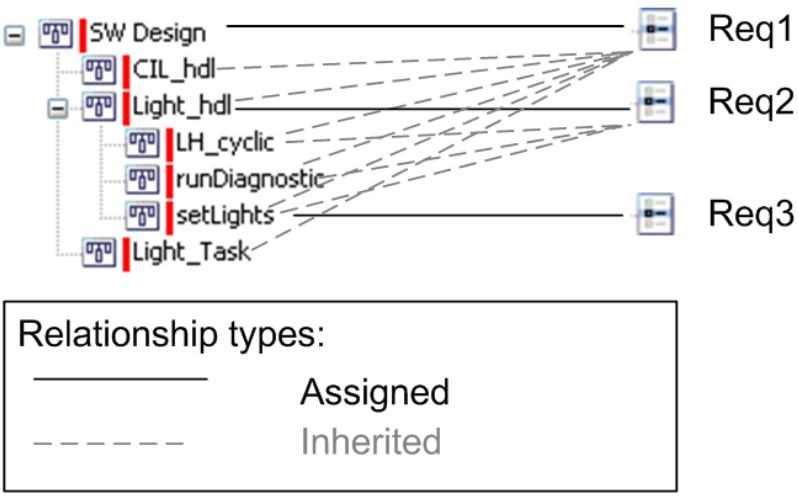

Figure 18-4 Requirements and the *requirement influence scope*

Fig. 18-4 shows an example[227]. Requirement "Req1" is assigned to the *AN* "SW Design". Its concern is then inherited by all sub *AN*s of the model, whereas requirement "Req2" is assigned to the "Light_hdl" module as a whole. This means all methods and contained data in the "Light_hdl" module must work together to fulfill "Req2". A very local requirement is then again seen by "Req3", whose influence scope only reaches to the method "setLights" within the "Light_hdl" module.

In this way, a requirement's *RIS* contains the *AN*s it is directly assigned to and the child *AN*s inheriting the responsibility. Inherited requirements of an *AN* are shown in the "Requirements" tab (cf. fig. 15-4 in ch. III.15) like all other requirements but with a gray colored requirement text.

The *RIS* has strong connection to the differentiation of *functional and non-functional requirements* in *REM* theory as *NFRs* per se have a higher influence

[227] Another striking analogy to this concept can be found considering a hierarchy of a company organization. If a requirement (or here rather to say issue) concerns the Chief Executing Officer of the company (corresponds to the "SW Design" *AN* on top of the design hierarchy), the issue will most likely become a concern of all other employees, whereas an issue concerning an employee at the lowest hierarchy level will be just a concern of this employee.

scope than *functional requirements*. However, the concepts are not the same. *NFRs* defining quality characteristics will most likely have the same influence scope as "Req1" meaning the whole software is responsible for fulfilling the issue. For other *NFRs* as, e.g., the demand for a user access rights management, the designers may find a realization that does not have such a high influence scope. As an example, it could be possible to define a *three layers architecture* ([BMR+00; p.31ff], ch. I.6.2.4), where the user access management – except for the graphical user interface dialog to assign rights – is handled in the data storage layer.

This example also points to three other aspects that must be considered:

- The lower the *RIS* of a requirement is in a design, the lower will be the *impact* of a requirement change to the design. Thus, designers should try to minimize the *RIS* of requirements in order to minimize the *impact* of the requirement. This topic will be a central goal in the next chapter discussing the *requirement dribble process* heuristic.
- On the other side, the *RIS* highlights requirements with high influence on a design, as they will stay at a very high level of abstraction being inherited by a lot of requirements. This is what Obbink et al. [OKK+02] term a*rchitecturally significant requirements*[228] *(ASR)* and what most probably imposes close connection to requirements imposing *neuralgic points* in the view of Moro [Mo04; p.326] (also cf. ch. II.9.4.1). In most cases, *NFRs* will be most of the *ASRs* (but also *FRs* could be *ASRs*) staying at the very high-level *ANs*.
- The *RIS* of a requirement can be influenced by the designers' decisions. As ch. I.5.1 and ch. II.10.4.2.2 indicate, a promising strategy to tame *NFRs* is to refine them into several *FRs* (cf. [PKD+03; p.145], [Pi04; p.99], [Mo04; p.339]). Often, these *FRs* then might have a lower *RIS* than the *NFR* would have had. In this way, a *NFR's* higher *RIS* is reexpressed through several *FRs* with a lower *RIS*. Such a step is a decision process. Due to the importance of *NFRs* concerning the general outcome of design (cf. ch. II.9.5), a dedicated support for documenting such decisions can prove very helpful. Ch. III.20 will discuss the decision problem and how R2A provides support to tame nonfunctional aspects with high *RIS* to a lower influence scope in a traceable way.

[228] "A requirement upon a software system which influences its architecture" [OKK+02; p.53].

III.18.2.3 Representing Requirement Contextual Data

As mentioned in the chapters above, R2A helps designers to cope with the complexity imposed by the high numbers of requirements by providing only requirement information relevant in the design situational context.

When the user selects an *AN*, the control shown in fig. 18-5 will show all requirements relevant for the selected *AN*. Directly assigned requirements are displayed in normal black text color. Inherited requirements are displayed in gray text color.

Fig. 18-5 also highlights two buttons for the operations "*dribble-up*" and "*dribble-down*" essential for the *requirement dribble process* described in the following chapter. Both buttons allow changing the requirement assignment in orientation to the *AN*-hierarchy. A requirement assigned to an *AN* can be moved up to the *AN*'s parent *AN* via the *dribble-up* operation. This means to change the realization of a requirement to a higher abstraction level implying that the *RIS* of the requirement is widened. Vice versa, a *dribble-down* operation allows delegating the realization of a requirement down from the currently assigned *AN* to one or more of its child *AN*s (the user can choose any combination of the child *AN*s). This corresponds to a narrowing of the influence scope of the requirement. In this way, a requirement becomes more local instead of global. Accordingly, this can also be termed as the localization of a requirement. Often, design is performed by several designers working together. In such constellations, it is often the case that one designer works on a higher *AL* and the other designer works on the lower *AL*. *Dribble-down* and *dribble-up* operations thus also traverse working boundaries. In this way also a collaborative information exchange between the designers takes place.

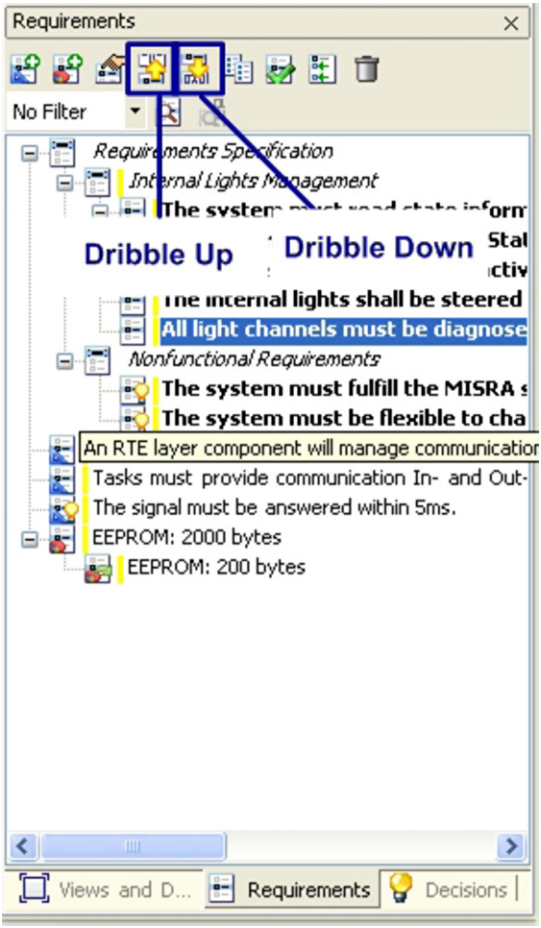

Figure 18-5 Showing requirements in the design situational context of an *AN*

III.18.2.4 The Requirement Dribble Process (RDP)

In the following, the *requirement dribble process (RDP)* heuristic[229] is introduced. As primary goal, the *RDP*'s intention is to provide a process for designers allowing them to establish *traceability* information as a *by-product* of their daily design activities and providing immediate benefits for the designers when taking the next actions of their daily design activities. In that way, the author hopes to solve the *traceability benefit problem* meaning that designers experience enough benefits for themselves to encourage them to record detailed, correct and thus valuable *traceability* information as a *by-product* of their daily design activities.

One major leverage to reduce the *traceability benefit problem* is to avoid what Dutoit et al. [DMM+06a; p.7] call "*cognitive dissonance*", meaning in Schön's view (ch. I.6.2.3) that establishing *traceability* might interrupt designers in their thinking, especially if they are in their *knowing-in-action* phase. Therefore, the *RDP* principles closely orient themselves on the *ANH* concept (ch. III.15) and try to be performable as fast and easily as possible in order to ensure that they can be realized without significant extra strains on developers.

Closely related to this issue is the problem that *traceability* information, once established, must be quickly and easily changeable in order to ensure that design is also adapted if assumptions, facts, or other factors spark the need for changing the design with its requirement allocation. Otherwise, either important design refactorings are just not performed due to more extensive effort, or *requirements traceability* information fastly degrades. A symptom often observed is that if *traceability* information is not easily changeable, design will be performed beforehand and *traceability* is established afterward when design has reached a relative stable state (see ch. I.7.2.3; comment on ENG.3 BP.2, ch. II.10.5, and ch. III.11). In these cases, however, much of the important *traceability* information may already be forgotten and thus gets lost. A special concern in this context especially is that important information on important decisions is easily lost.

Additionally, design usually is a collaborative task. Correspondingly, the heuristic provides dedicated support for collaborative information sharing between designers at different levels of abstraction.
Several ideas form the central pillars of *RDP*:

---

[229] The term heuristic emphasizes that it is more a guiding principle, where deviations are possible. However, the author is convinced that in principle most of the SW-based design processes – even in those design processes, where design is only present implicitly in code – follow this principle to the one or other extent. The so called *bottom-up* processes can be seen as the only big exception, but later it is shown that *bottom-up* processes are also merely compatible.

- The *abstraction nodes* concept,
- The concept of s*table intermediate forms* as developed by Simon (cf. ch. I.6.2.1 and ch. III.18.2),
- The *requirement influence scope* (*RIS*) concept;

III.18.2.4.1 Description of the RDP

The name '*RDP*' derives from a metaphorical analogy to rain water dribbling onto a mountain. In a similar way, the *RDP* heuristics allows design with its corresponding *AN*s to emerge in a requirement-driven way by letting requirements 'dribble' through the *ANH* tree. The basic idea is that requirements are not necessarily directly assigned to the *AN* that will finally be responsible in the future. Instead, a process is possible, where the optimal solution for a requirement is found in the course of the process heuristic. At first, this means that a requirement can be added to an *AN* at a very abstract *abstraction level* (*AL*), e.g., the highest *AN* of a model. According to the *requirement influence scope* (*RIS*) concept, this first of all implies that a large extent of the design would be responsible for fulfilling a requirement. In this constellation, later changes of the requirement would have far reaching consequences (*impact*). Thus, to avoid requirement changes having enormous consequences, all further design decisions shall act upon a maxim to reduce the *RIS* of any requirement to a level as local as possible. Keeping this in mind during design, the designer of an *AN* analyzes the assigned requirements and tries to find solutions which allow delegating the requirements to an *AN* at a lower level of abstraction via *dribble-down* operations. In this lower abstraction level with the lower *RIS*, the designer responsible for the corresponding *AN* again tries to find a solution allowing him to delegate the requirement to an *AN* to lower *AL*s, thus again lowering the *RIS*. This happens as long as a requirement cannot be realized by *AN*s of a lower *AL* in an expedient way. In this case, the requirement now either comes to rest at this *AN* and its sub *AN*s inherit the requirement as obligation to work together to fulfill the requirement's needs, or the requirement can be split[230] up to be fulfilled by several sub *AN*s of the lower abstraction.

[230] A split operation, however, should be omitted if possible. The general goal should be to perform "*dribble-down*"-operations of requirements into disjoint paths, so that most of the requirements will only take one way to *dribble down* into the design; but sometimes a split up may be not avoidable. If not avoidable, such a split up should occur at an *AL* as low as possible in order to avoid a *requirement-fan-out* as described in ch. II.10.6.2 leading to a high *RIS*.

In some cases, the designer of an *AN* could discover that an assigned requirement cannot be adequately fulfilled by the *AN*. For example, this can happen because the designer having delegated the requirement from a higher level *AN* to the current *AN* has not been aware of some facts (resp. problems). In this case, the designer of the current *AN* can again redelegate the requirement to its parent *AN* at the higher abstraction level by a *dribble-up* operation. Such a situation occurs when the requirement cannot really be fulfilled by the selected *AN*. Thus, the *dribble-up* operation will correct the mistake. Often, however, it could also be a communication problem when several designers work together at different *AL*s. Such a case can happen when the designer of the higher *AL* assigns a requirement to the *AN* of the lower *AL* but forgets to regard some other aspect influencing the potency of the *AN* to fulfill the requirement. For example, it can be the case that the AN is missing access to an information of another component necessary to fulfill the requirement. Here, R2A allows the designer of the lower *AN* to redelegate the requirement to the designer of the higher *AL* via a *dribble-up* operation accompanied by a note describing why the requirement cannot be fulfilled by the lower *AN* in the current setting.

This note information additionally helps the designer of the higher *AL* to regard the forgotten aspect and – if possible – to solve the problem. For example, by designing a solution that allows the *AL* to access the needed information. Afterward, the designer of the higher-level *AN* can again assign the requirement to the lower-level *AN* via a new *dribble-down* operation.

During the *RDP*, *dribble-down* and *dribble-up* operations can be performed by all designers involved in the design forming a collaborative form of information sharing. At the end, the *RDP* design process heuristic should converge to a design where all requirements are considered in pursuing the goal that each requirement has a *RIS* as low as possible, which leads to a design where changes on a requirement – hopefully – has minimal *impact*.

A significant advantage of the *RDP* is that the heuristic always preserves the exact current state of a design. Often, requirements important for an *AN* are scattered over several locations in a requirement document. Therefore in current practice, the designer of an *AN* often must analyze the complete requirements specification to identify all requirements important to the *AN*. In this way, every designer must nearly analyze the complete requirements specification to identify the requirements important for him. With the *RDP* approach, a list of the requirements concerning an *AN* is directly provided by R2A and thus, designers do not need to analyze the complete requirements specification but can directly benefit from works other designers have performed. Additionally, the *RDP* heuristic also promotes that a current snapshot of the current design status is available supporting the designers to take their next design steps and decisions, thus also

promoting that *requirements traceability* is performed as a *by-product* of the design effort and not afterward.

III.18.2.4.2 A RDP Case Study

To explain the heuristic the reader must consider the accompanying case study introduced in ch. III.12. At first, it is assumed that only the *requirements specification* as shown in fig. 12-2 (ch. III.12) is present and no design has taken place. Thus, a high level software architect (in the further called architect) starts the design from scratch.

At the beginning of the project, the architect starts the process by creating an empty diagram intended as the *high-level architecture* overview and adds this diagram to R2A as the first *AN* (in the further called high-level *AN*) in the *abstraction hierarchy tree*. When analyzing the requirements, the architect decides to care for the "Internal Lights Management" use case. He assigns the requirements of the *use case* to the high-level *AN*. This means the high-level *architecture* is now responsible for the requirements of the *use case*. From this first *stable intermediate form*, the designer can now analyze the *use case* requirements and take further actions. Requirement ReqSpec_2 implies that the system has a CAN connection. Correspondingly, the design needs a CAN_drv driver to control the CAN-HW in the ECU and a CIL_hdl mapping signals from CAN to signals within the software. Thus, the designer creates both *design elements* in the modeling tool, adds both elements to the high-level *architecture* diagram (see fig 12-2 (ch. III.12)), adds the *design elements* to R2A as new *AN*s located beneath the high-level *AN* and then performs a *dribble-down* operation relating ReqSpec_2 to the CIL_hdl, thus localizing ReqSpec_2 to the CIL_hdl.

In a similar way, the architect analyzes requirement ReqSpec_3 and ReqSpec_4 and determines that he needs a Light_Task component. Correspondingly, the designer creates the *Light_Task* component in the *design tool* and adds it to R2A's *ANH*. Now, the designer can delegate ReqSpec_3 and ReqSpec_4 to the Light_Task component via *dribble-down* operation. In this way, the architect roughly analyzes the diversity of the requirements and decides the modularizations, attributes, etc. important from the architectural viewpoint.

Following the current example, the architect identifies the following modules and their important roles:

- Light_Task: is responsible for the evaluation and propagation of the light requests received from outside (e.g., via CAN). The Light_Task can involve a complex state machine.

- Light_hdl: is responsible for translating logical light function requests into the different control of light channels provided by the HW. Further, the Light_hdl is responsible for error diagnosis functionality on the controlled HW light channels and the further processing of measured diagnostic information. To achieve this, the Light_hdl also is responsible for timing the diagnosis functionality as diagnostic measurements must be exactly timed to retrieve valid values.
- PWM_drv: is responsible for realizing demanded pulse widening modulation (PWM) to control the light intensity of the controlled lights.
- ADC: controls the analog-digital converter component within the microcontroller needed to convert analogous feedback currents of the steered lights into digital measurement values for diagnosis on the controlled HW light channels.

As the architect anyhow roughly analyzes the requirements and makes his design decisions on the bases of these, the architect can already assign the requirements to the identified and modeled *design elements*. In this way, he also implicitly documents the basic information on the decision leading to the *design element* as well as to its responsibilities and thus creates *traceability* information as a mere *by-product*.

In the next step, the module designers of the modules (usually, for each module an individual module designer exists) care for realizing the assigned requirements in the specific modules. Thus, at the abstraction level of the module, every module designer starts to analyze the present requirements in detail to identify and model the necessary sub-components, data, and operations. For each identified item the designer adds an *AN* in the abstraction nodes tree and assigns the requirements for the *AN* via a *dribble-down* operation. In this way, the designer automatically documents the basis of his design decision for the corresponding *AN*.

At the level of these newly created *ANs*, the requirements are very likely analyzed in more detail than it happened at the higher-level *ANs*. Correspondingly, the module designers will also encounter contradictions and incompletenesses in the entire design. As an example, the module designer of the Light_hdl module might recognize that, in order to be able to perform the analysis of diagnostic data according to the requirements (indicated by ReqSpec_6), he needs further – not yet considered – information currently only available to the Light_Task. As the solution of the problem is outside of his decision-making authority, he must submit the issue to the designer responsible for the design of the interaction between Light_Task and Light_hdl. In this case here, this is the SW-architect. For this, in a non-R2A project, the module designer of the Light_hdl would now need

to have a talk with the architect about the problem, in which both must use a synchronous communication mechanism.

However, in several cases the architect may be busy, or distributed to another location, or just absent. In all these cases, constant dangers exist that the issue gets somewhere stuck or forgotten. Using R2A, the module designer is able to redelegate the requirements back to the higher *AL* by performing a *dribble-up* operation. To provide further information on the issue, the module designer can add a note on the requirements describing the problem. The architect then is notified about these requirements again at his *AL*, can read the attached note to understand the problem, and then take decisive action whether the requirements should be fulfilled by a functionality to exchange the needed information between Light-Task and Light_hdl or an alternative strategy such as remodularization (the needed information is relocated into the Light_hdl) is used.

Through this way, asynchronous communication between the designers is possible, where no problems are forgotten, and decisions are implicitly documented in addition.

In the further project progress, the module designer can then refine the design of the module. In case the code is generated automatically, the software developer can then directly implement the realization of the module according to the design and the assigned requirements. Also, in this case, the implementer directly has all necessary requirements for the module at hand and is able to use the *dribble-up* mechanism in any case he discovers problems he cannot solve at his level of authorization.

III.18.2.4.3 Bottom-Up Design Processes within RDP

The *RDP* seems to be a method particularly fitting to *top-down* design processes. However, as discussed in ch. I.6.2.1.3, pure *top-down* design processes are rather an exception. In many cases, design evolves in rather *non-linear* decision processes. The other extreme to *top-down* design is pure *bottom-up* design. Most design processes will be a mixture somewhere between both (see, e.g., [HR02; ch.10]).

As mentioned before, the *RDP* is just a process heuristic. R2A's features provide flexibility to implement different processes. To support *bottom-up* design processes, the following process setting is conceivable:

- The designers created *design elements* in the used modeling tool, add the *elements* to R2A as *AN*s (via the wizard or *drag-and-drop*; see ch. III.15) and assign the requirements the *design element* is intended to fulfill.

- If the hierarchy later changes (e.g., a parent is added to the *design elements*), the *ANH* synchronization mechanism can easily reconstruct the new *ANH*. The requirements assignment stays untouched.
- When the *ANH* grows, the *dribble-down* and *dribble-up* operations also provide valuable support for changing requirement assignment and thus implicitly the *RDP* principles are again at work.

III.18.2.4.4 RDP Summary

The *RDP* approach offers significant advantages to other known *traceability* methods addressing *traceability* between requirements and design:

- The linking between requirements and model elements emerges indirectly as a *by-product* since the assignment of the requirements always resembles the current state of decision about a requirement (*stable intermediate form*). Later in ch. III.20 and ch. III.21, the author describes other kinds of decisions also addressed by R2A through dedicated decision models. Also products related to these *decision models* (*design constraints* and *budgeted resource constraints*; see ch. III.19 and ch. III.21) can again be treated by the *RDP*.
- In parallel, through the detailed recording of all steps taken to achieve a design, detailed documentation of the decision-making of a design is enabled allowing easier reconstruction of the original ideas behind individual design decisions in the case changes are needed.
- Also, the designer has an immediate overview of the remaining, not yet treated requirements at an *AN*, because the already treated requirements have been delegated – and thus disappeared – to one or several sub *AN*s. Later, in ch. III.22, the principle mentioned here is even extended by a mechanism for ensuring consistency.
- Normally, several developers work on a *design model*. Via R2A, the delegation of responsibilities between the developers can be achieved by interplay of the *AN*s with the *RDP* concept, building a *scaffold* (i.e. *skeleton*) for collaborative information interchange.
- Through the support of a dedicated process for assignment and care of a requirement, it is ensured that each requirement is adequately considered in the design process: If new requirements are assigned to an *AN* from a higher-level parent *AN,* these requirements get highlighted in the *AN* by a different color. Now, the designer of the *AN* must try, to find an adequate solution for the newly assigned requirements. If the designer of this *AN* is again able to delegate these requirements to a sub *AN* of the design, then these requirements '*dribble down*' one level deeper to a sub *AN* and the problem is solved for the

corresponding *AN*. However, if the designer is not able to clearly delegate these requirements to any sub *AN*, then the *requirements* sticks to this *AN* and are inherited to all lower level sub *AN*s (marked 'gray') indicating that all *AN*s together must deal with fulfilling these requirements. But if the designer responsible for the *AN* realizes that these newly assigned requirements cannot be fulfilled in the current state of design, the designer is able to repel these requirements back to the higher-level *AN* (its origin) accompanied with a corresponding note. In this case, the designer of the higher-level *AN* must care for a solution under consideration of the created notes.

- Effective communication between the designers is alleviated since the approach relies on mechanisms supporting asynchronous communication via the assigned requirements and notes. Thus, less synchronous consultation between the designers is needed.
- The documentation of views in design with their textual descriptions and all important decision information is essential for *architecture documentation* (*AD*), (cf. [Ha06], [CBB+03]). Thus, R2A also supports generating reports from all recorded information to fulfill *AD* needs. In this way, also information gathered through the *RDP* heuristic completes information needs for *AD*.
- When the design process is thought beyond the scope of mind discussed now, a similar mechanism for other information to dribble through the designed system in a similar fashion could be helpful. Thus, e.g., a design decision (see fig. 20-2 (see ch. III.20)) in a high *AL* often restricts the *solution space* in the lower *AL*s. If these so-called *design constraints* are formulated once, they can dribble through the system in the same fashion. In order to allow high adaptability to project specific needs, other item categories may be individually definable by additional information for each project.

III.18.2.5 Overview over Navigation and Handling of Requirements Aspects in R2A

Fig. 18-6 shows an overview how features described in the chapter above are integrated into R2A concerning navigation and handling. At the left part, the model with the *ANH* tree as described in fig. 15-4 (see ch. III.15) is shown. Via selecting the “Requirement Sources” tab (1.), the control for managing all *requirement source documents (RSD)* is displayed (see fig. 18-1 (see ch. III.18.1)).

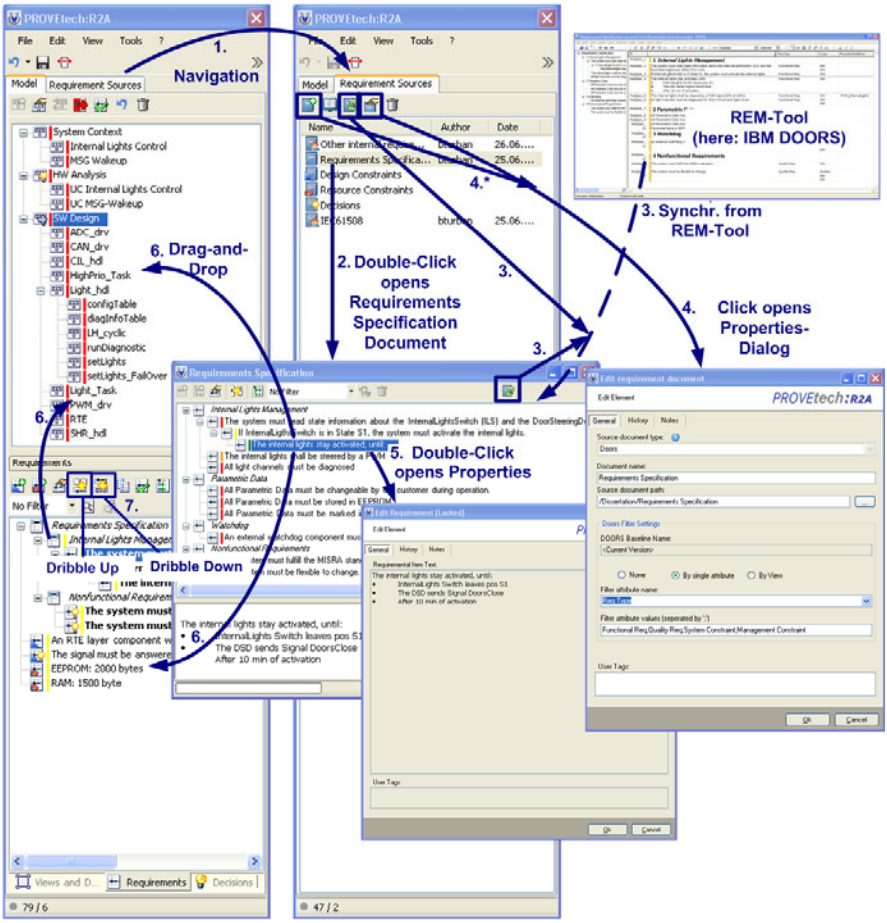

Figure 18-6 Overview of how the requirements-related features are integrated into R2A concerning navigation and handling

A double-click (2.) on a document opens the *RSD's* content window displaying the requirements of the *RSD* (see fig. 18-2 (see ch. III.18.1)). In fig. 18-6 the content of the *RSD* "Requirements Specification" is shown.

A left-click on the properties-button (4.) opens the properties dialog for the *RSD*. When the new-button (4.*) is clicked, a new, empty properties dialog is opened leading to the creation of a new *RSD* if the 'ok'-button of the properties

dialog is clicked. Fig. 18-6 shows the properties dialog of the *RSD* "Requirements Specification". As the properties show, this *RSD* is configured to refer to a requirements document managed in the *REM-tool* IBM Rational DOORS.

Through the synchronization buttons (3.), a synchronization mechanism can be invoked to synchronize the requirements contained in the *REM-tool* (symbolized by the upper right window in fig. 18-6) to the *RSD*. The synchronization mechanism can be continuously invoked to synchronize changes performed in the *REM-tool* to R2A's *RSD*, keeping it up to date. Ch. III.22.2 shows how this mechanism can be used to consistently infer requirement changes into a R2A design. Requirements being synchronized from an *REM-tool* cannot be edited in R2A.

In the properties dialog, a *RSD* can also be set to status 'Free Edit'. In this case, freely editable new requirements can be created in the *RSD's* content window.

In an *RSD's* content window, the requirements are displayed in the hierarchical decomposition structure. A double-click (5.) on a requirement opens the properties dialog of the requirement.

Via *drag-and-drop* operations (6.), *traceability* can be established to the *ANs* (also cf. fig. 18-3 (in ch. III.18.2.1)). These in combination with the *dribble-up* and *dribble-down* operations (7.) form the basis for the requirement dribble process heuristic (ch. III.18.2.4).

III.19 Taxonomy of Requiremental Items[231]

Each definition of a system layer yields some of the requirements for the subjacent layer.
Hatley et al. [HHP03; p.52 (*)]

The SPICE *process model* (described in ch. I.7.2) is a layered *process model*, in which *problem space descriptions* (requirement view: ENG.2, ENG.4) alternate with *solution space* descriptions (designs: ENG.3, ENG.5) at different levels of abstraction (cf. ch. I.7.3.2 for detailed exemplification).

Ch. I.7.3.2 has outlined the problems of this layered *process model* concerning *traceability*. Two major problems were discovered:

- High redundancies between the requirement artifacts lead to higher efforts for *traceability* and consistency management (see fig. 7-2 (see ch. I.7.3.2)).

[231] Significant parts of this chapter are taken from [TKT+07] and [TTW07].

Additionally, despite all consistency management efforts, drifts between the different requirement artifacts' redundancies are often not avoidable.

- Between the different artifacts (especially, when also the HW dimension is considered) other correlations are not adequately manageable (see fig. 7-2 (see ch. I.7.3.2) and fig. 7-3 (see ch. I.7.3.2)).

The first problem with redundancy could already be solved to a great extent by a process artifact model described in fig. 7-3 (see ch. I.7.3.2). One prerequisite, however, is to acknowledge that *process models* such as SPICE are to a certain degree rather a metaphor providing space for interpretation than a law to be obeyed word for word. In the author's opinion, this degree of freedom is present in SPICE, because SPICE itself emphasizes that the *process model* is only an example *process model* and other *process models* are possible to be defined as long as they conform to the original metaphoric ideas of the SPICE standard[232].

Now, the solution shown in fig. 7-3 (see ch. I.7.3.2) still neglects one central metaphoric idea of the layered *process model* that is covered by R2A via the concepts described in this chapter, ch. III.20 and ch. III.23.2: *System design* has high *impact* on its *SW design* by raising new "requirements" in addition to the pristine requirements of the stakeholders. For example, in the automotive sector, *SW design* must be subordinated under constraints of extremely cost-optimized HW components. At the moment, SPICE neglects these critical connections between HW and SW but at least acknowledges this connection concerning *system design* (see ch. I.7.2.4).

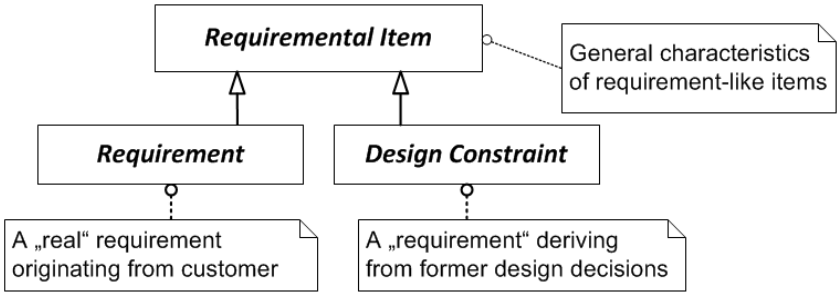

Figure 19-1 *Requiremental items*, *requirements* and *design constraints* taxonomy

[232] It is, however, more difficult for an organization to prove conformance to these metaphoric ideas for a different *process model* than for a *process model* just taking over the ISO/IEC 12207 *process model* used in the SPICE standard. Thus, most SPICE implementations in practice just use this *process model*.

However, one issue in SW requirements which might benefit from more intensive discussion is their negotiability. “Real requirements” are forming the contractual basis between the stakeholders – particularly with the customer. Occurring changes must be harmonized with the customer via a *change control board* (*CCB*) [PR09; p.144f], [VSH01; p.184f, p.216]. Whereas, for “requirements” to be changed with the origin of the definitions of the design, it is possible to search for a project-internal solution first, before escalating the issue to a *CCB* is considered.

Thus, both kinds of requirements should be strictly separated in their notion[233]. The author uses the following taxonomy (fig.19-1):

- *Requirements* are directly allocated to the *SYS_RS* since they concern the legal agreement between customer and contractor.
- 'Requirements' derived from requirements or designs are called *design constraints* (*DC*).
- *Requirements* and *design constraints* have similar qualities and structure. Thus, we use the term *requiremental*[234] *item* (*RI*) for both items.

Generally, requirements have to refer to their origin (cf. description to IEEE 830-1984 in ch. I.5.7). This relation should apply to all *RIs*. The origin of *DCs* lies in previously made design decisions solving the conflicts/forces between *RIs* and/or architectural items, constraining the broader, more abstract *solution space* to a more concrete one. The *decision model* connected with the *DCs* is discussed in the following ch. III.20.

Observations leading to the *DC* concept are not new. Leffingwell and Widrig define constraint as “a restriction on the degree of freedom” the developer has “in providing a solution” [LW99; p.55]. *DCs* also resemble to what the IEEE 610 defines as *design requirements* (“A requirement that specifies or constrains the design of a system or system component” [IEEE610; p.26]) or *implementation requirements* (“A requirement that specifies or constrains the design of a system or system component” [IEEE610; p.39]).

The *DC* concept directly corresponds to observations of Hatley et al. that design decisions[235] generate new requirements for sub system components [HHP03; p.18]. These new requirements are a result of former design and should be con-

---

[233] This directly corresponds to the view of Pieper in [RS02; p.33-35] demanding a clear separation between requirements from the customer and internal requirements in the project.

[234] The artificial word '*requiremental*' has been introduced by the author as a term for describing superordinate characteristics of *'real' requirements*, *design constraints* and *budgeted resource constraints* (see ch. III.21).

[235] See also Ebert's remarks that decisions constrain the *solution space* [Eb05; p.14].

sidered in a development process [HHP03; p.31]. In general, these 'requirements' are more numerous than the original requirements [HHP03; p.32]. This matches with Glass's note on complexity that "explicit requirements explode by a factor of 50 or more into implicit (design) requirements as a software solution proceeds" [Gl02; p.19], (also cf. ch. I.6.2.1.1).

Lehman's *principle of SW uncertainty* describes that assumptions on which design decisions depend can be implicit or explicit to developers, but both kinds can get invalid due to changes [Le89]. Requirements can be seen as a kind of assumptions (however, also other kinds of assumptions may exist). In this case and in the face of high volatility rates, changes on explicit assumptions are much easier to handle than implicit assumptions. Via the *DC* concept it is possible to make these implicit assumptions more explicit, thus potentially improving *IA* and consistent implementation of changes.

III.20 Support for Capturing Decisions[236]

A further complication is that the requirements of a software system often change during its development, largely because the very existence of a software development project alters the rules of the problem.
[Bo94; p.4]

Most current state-of-the-art *traceability* models assume that *traceability* between requirements and design can be expressed by a simple bidirectional linking concept, where each requirement is related to the *design elements*. The link concept can surely be helpful to cover relatively easy situations. However, *traceability* literature ([Kn01a], [Kn01b], [PDK+02], [Pe04], [RJ01], [Al03]) provides strong indications that the influence of requirements on design processes – and vice versa – is only insufficiently modeled by bidirectional linkages.

Paech et al. [PDK+02] indicate that these relationships can be of a far more complex nature (cf. fig. 20-1). By restraining the *solution space*, *non-functional requirements (NFR)* restrain *functional requirements (FR)* and *architectural decisions (AD)*. On the other hand, *NFRs* are realized by *FRs* and *ADs*, whereas *FRs* are realized and restrained by *ADs*.

[236] Significant parts of this chapter are taken from [TKT+07] and [TTW07].

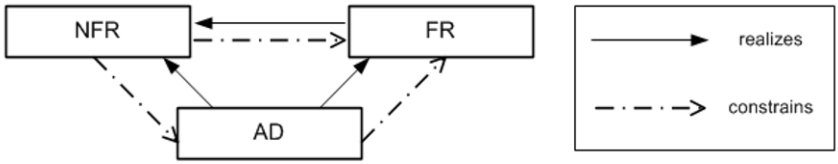

Figure 20-1 Interactions between nonfunctional, functional requirements and architectural decisions [PDK+02]

The simple linking concept indirectly assumes that requirements and design are mostly interconnected by linear relationships. As the author tried to elicit in part II and ch. I.6 of the thesis, the transitions from requirements to design is often nonlinear[237] but more a creative mental transfer process of a problem description (requirements) to a solution, where the taken decisions build the foundation of these transitions (also cf. [TKT+07]). The path from the requirements to its realizing design can be described as a sequence of decisions constraining the *solution space*. This circumstance induces that design does not only depend on its requirements to be fulfilled, but it depends to a higher extent from the decisions taken before. Now, this observation leads to the following two points to consider:

- Decisions and their effects must be communicated to other designers, developers and testers within the project. As ch. II.9 shows, approaches for decision documentation exist. In practice however, if any decision documentation is done, the information will be documented in some design documents (as, e.g., propose by Clements, Bass et al. in connection with *SW architecture documentation* [BCN+06], [CBB+03]). By such an unstructured way, problems can then arise then, when this information must be propagated to other stakeholders or even is to be processed in the further by other stakeholders.
- Later requirement changes not only influence the design but can also lead to the need to reassess formerly taken decisions and – if necessary – to revise them leading to new *impacts* on the design.

These considerations suggest the inclusion of a decision model in the *traceability* information helping to document the origin of new *design constraints* in a

[237] Also interesting in this connection is what Kruchten says about the design process he proposes associated with his *"4+1 View Model" architecture* approach: "Finally, this is not a linear, deterministic process leading to an optimal process view; it requires a few iterations to reach an acceptable compromise. There are numerous other ways to proceed" [Kr95; p.48]. As a consequence the question arises, why the traces of such a process should be linear.

lightweight and need-oriented way. Fig. 20-2 shows this concept extending today's *traceability* models by an explicit decision model. The diagram sketches a concrete situation, where a conflict between three requirements (Req_1, Req_2 and Req_3) and two *design model elements* (Class1, Class2) is resolved by a design decision (Decision1), resulting in two new *design contraints* (DesConstraint1, DesConstraint2).

The conventional scheme of relating requirements to realizing model elements is extended by a dialog allowing the capture of documented decisions. In this dialog, elements of the requirement model and the *design model* which are conflicting, i.e., causing a problem, can be chosen. Equally, diagrams describing aspects of the conflicting situation shall be attached as additional information (<<*documenting diagrams*>>).

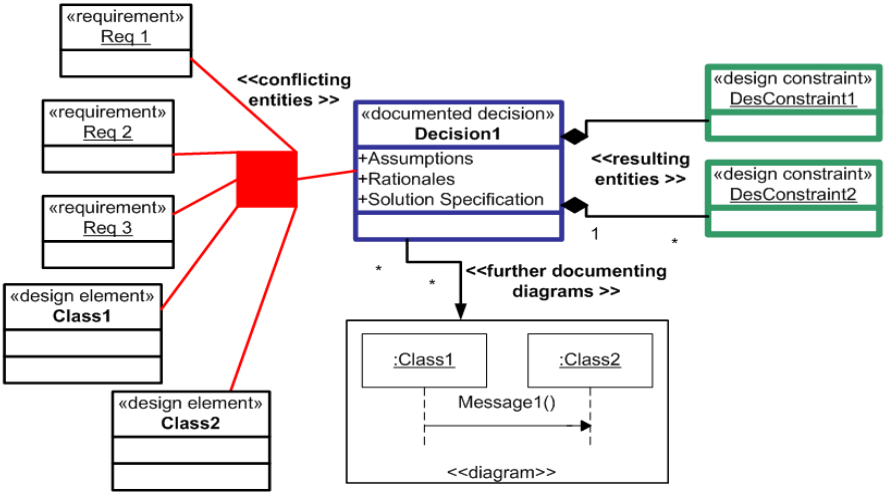

Figure 20-2 Documented decisions build the connection between *requirements*, *design elements* and resulting *design constraints*

Furthermore, the decision can be specified on demand via a text component. The text component accepts unstructured text, but – when needed – can give adequate templates to support the decision documentation. A possible way to structure – the user should choose these freely – is given in fig. 20-2 with the decision's attributes *assumptions, rationales* and *solution specification*.

III.20.1 Relation to Approaches of Rationale Management

The *decision model* presented here is strongly connected to *RatMan* (see ch. II.9), since both deal with decisions during SE processes. In classical *RatMan*, the focus lies on documenting, recovering, further usage and reuse of justifications (= *rationale*) behind design decisions. *RatMan* mainly targets on the information about the 'Why' of design decisions in order to alleviate the knowledge transfer of decision makers to other involved stakeholders.

However, existing approaches could not succeed in practice [DMM+06a], even though documenting design decisions is regularly called for in literature (cf. [IEEE1471], [CBB+03], [BCN+06], [Ri06], [PBG04], [GP04], [Bo94]) and practitioners acknowledge the importance of this type of documentation [TAG+05]. Diverse causes for this negligence have been identified, but the problem of *capturing* the *rationale* seems to be the main obstacle (cf. [DMM+06a], [HA06a]):

1. Most approaches are highly *intrusive* (bothersome and interfering) to the design process with extra effort for capturing (ch. II.9.1.4, ch. II.9.4.2, [Gr96b], [HA06a]).
2. The approaches tend to have negative impact on the decision process, since not all (aspects of) decisions can be rationally justified but arise from intuitive considerations (Schön's "*Theory of Reflective Practice*" [Sch83] adopted by Fischer et al. [FLM+96], [DMM+06a]) basing on diffuse experiences (e.g., *tacit knowledge* [Po66]; also cf. [DMM+06a], [HA06a], [SM99a]).
3. Decisions must be made despite of unclear circumstances and it is impossible to include all relevant information (*bounded rationality* [Si96], [HA06a]). Thus satisfactory solutions must be found although problem knowledge is clearly limited [LF06].
4. *Grudin's principle* [Gr96b] suggests that collaborative systems fail if the invested value is not returned to the information bearers (ch. II.9.4.2, [DMM+06a], [Sch06]).

The problem mentioned in point one implies that *not all* decisions can be treated exhaustively in any case. For example, Clements, Bass et al. only refer to the documentation of the most important decisions ([CBB+03], [BCN+06]). Booch [Bo94] gives another lead by dividing decisions[238] into *strategic* (i.e., with striking *impact* on *architecture*, mostly made on the early stage of a project) and *tactical* (i.e., locally limited *impact* on the *architecture*).

[238] Also cf. Canfora et al. [CCL00] distinguishing maintenance *rationale* into two parts: *Rationale* in the large (*rationale* for higher-level decisions) and *rationale* in the small (*rationale* for implementing a change and testing).

In this context, strategic decisions must/should be thought through carefully and should –if possible– be made on explicit *rationale* grounding. For this relatively small fraction, the investment in more intensive analyzes is highly valuable, as discussed by most approaches on rational management ([RJ01], [CBB+03], [BCN+06], [TA05]). These issues may be analyzed in a *prescriptive schema* as *IBIS* [KR70], or the *Rationale Model* of Ramesh and Jarke [RJ01], or REMAP [RD92], or Clements and Bass [CBB+03], [BCN+06]. R2A's decision model (see fig. 20-2) supports this by additionally allowing defining a project individual template for the textual description component of the decision (in fig. 20-2 shortly sketched by the bullets "*Assumptions*", "*Rationales*" and "*Solution Specification*").

On the other hand, Booch [Bo94] also demands that *tactical* decisions should be documented. At that time, Booch thought both kinds would disclose themselves by applying adequate modeling. Today's experiences show that such modeling just documents the *how* but not the *why* of decisions. In this context, Dutoit et al. [DMM+06a; p.39] provide the heuristic to concentrate on documenting decisions that are not obvious or *impact* other decisions. Referring back to Booch's view, it can be said that modeling captures a certain part of the decisions and the R2A decision mechanisms help to document the not obvious and especially influential decisions.

In the author's opinion, the developers should at least get the possibility to document decisions on demand, but considering aspects mentioned in point 2 and 3, the *intrusion* on the development process must be minimized ([Sch06], [HA06a], [DMM+06a], [SM99a]).

Keeping this in mind, a key goal of this decision model approach is to lower the barriers to making design decisions explicit as much as possible: Therefore, this decision model mechanism offers to designers a simple, *semi-formal model* as a *skeletal structure* to easily add basic information[239]. For this, the proposed decision model provides a minimal notational framework to identify the conflicting elements (*requiremental* and *design*) and to derive the resulting consequences as *DCs*. Thus, the conflicting elements define the area of conflict with the counteracting forces, automatically documenting the basic *rationale* behind a decision as a *by-product*.

In that case, however, the model is minimalistic and of a purely *descriptive* nature. Any further users of such minimalistically documented decisions must at first derive the actual knowledge about the decision on their own. But at least the fact that the context (the conflicting items and the results of the decision as *DCs*) is present for each decision provides evidence to later users: They can infer that a

[239] In this way, the approach resembles to the *QOC* approach (see ch. II.9).

decision has been made consciously and first clues are given for recovering the *rationale* (cf. [RLV06]). Further, this modeling of consequences pays tribute[240] to Horner and Atwood's claim that designers must consider the "holistic affects" of problems, their *rationale* and solutions [HA06a; p.84], (also cf. ch. II.9.1.4 and ch. II.9.4.2).

In that way, not all decisions can be reconstructed. Since the tool discussed here shall also automatically record such *meta-data* like the author(s) of a decision, the later users of a decision (*rationale seekers*) can consult the author(s) about unclear aspects. Additionally to tool usage, a *process rule* shall prescribe that the *rationale seekers* must document the results of this *decision recovery* in the decision's textual description to further improve the decision's documentation.

This procedure –inspired by Schneider ([Sch06; p.97]: "Put as little extra burden as possible on the *bearer* of *rationale*") – helps to cope with the problem in point four (see above), because by deferring the documentation work to the inexperienced *rationale seekers*, the experienced *know-how bearers* are significantly disburdened from communication resp. documentation work. As a positive side-effect, the transferred knowledge is consolidated in the *rationale seeker* during his documentation work.

On the other side, only unclear decisions will go through this further *rationale* request and documentation process. Therefore, the approach indirectly minimizes the documentation overhead by orienting itself on the selective information need of the further *rationale seekers*.

Van der Ven et al. express the observation that design decisions spark these new requirements, which then also must be satisfied by an *architecture* [VJN+06; p.340]. Van der Ven et al. [VJN+06] therefore also propagate to capture information about design decisions, because this helps to address central problems in design [VJN+06; p.332, p.341]:

- *"Design decisions are cross cutting and intertwined"* [VJN+06; p.341]: Many design decisions affect multiple parts of a design. As usual design processes do not explicitly represent design decisions, this knowledge is often fragmented across various parts. The designer himself knows these connections at first but always is in danger to forget it. Also Dutoit et al. [DMM+06; p.86] emphasize that much of design is done through evolutionary redesign and therefore long-term collaboration is essential. An adequate design decision representation can help to preserve the knowledge about the intercon-

[240] Even though, this tribute is far from being holistic, the *decision model* approach described here is a first try to establish *rationale* in practice. If the *decision model* concept proves to be sustainably successful in design practice, the model can be enhanced by modeling further more holistic connections.

nections. Later, designers can again be made aware of such cross-cutting and intertwined connections. If then some of the interconnections are no longer desirable (e.g., due to newly discovered facts), the structure can be refactored more easily.

- *"Design rules and constraints are violated"* [VJN+06; p.341]: "During design evolution, designers can easily violate design rules and constraints arising from previous decisions" [VJN+06; p.332]. Such violations are usually the source of architectural drift. Through an adequate design decision representation, designers can be made aware of design rules and constraints imposed by former decisions. In this way, architectural drift can be avoided better.
- "*Obsolete design decisions are not removed*" [VJN+06; p.341]: During evolution of design, some previously taken decisions become obsolete. Recorded information about decisions helps to "predict the *impact* of the decision and the effort required for removal" [VJN+06; p.341].

The *DC* and *decision model concept* proposed here has potential to alleviate these issues. Thus, concerning *RatMan*, R2A tries to balance and connect *descriptive* pragmatism and structured *prescriptive* methodologies. *RatMan* is not R2A's central issue, but this chapter shows that *requirements traceability* and *RatMan* are very closely related to each other and complement one another.

A further general problem of *RatMan* not yet discussed here is the *retrieval* of documented decisions. Horner and Atwood [HA06a] argue that fixed schemes –in contrast to unstructured text– offer better possibilities for indexing according to *retrieval*. The following chapter shows how the *retrieval* problem can be avoided through usage of the gathered *traceability* information of this approach.

III.20.2 Effects on the Traceability Model

The idea of including decisions into the *traceability* models has already been proposed by Ramesh with his REMAP tool [RD92]. In a later empirical study on *traceability* (see ch. II.10.4.2.3), Ramesh and Jarke ([RJ01]) detected a real need by experienced users. Therefore they include a separate *traceability* sub-model (*rationale sub-model*) for decisions, which is oriented on the former works with REMAP.

The decision model being proposed here has been inspired by the *rationale sub model*, but in the author's view Ramesh and Jarke's [RJ01] solution lacks making concrete proposals for implementation and thus, the *RM* component appears loosely connected to the other *traceability* sub models. Besides, the *ra-*

tionale sub model (orienting on REMAP) extends *IBIS* [KR70], which is a *prescriptive* and *intrusive* method (cf. ch. II.9.1.4, [LL00; p.202ff]).

In contrast, this decision model directly fits into the *schema* for *traceability* to design. In that way, a *semi-formal model* has evolved which provides easy handling and which has the following characteristics:

- A constellation (combination) of requirements and *design elements* leads to conflicts.
- Decisions do not directly influence dedicated design objects, but they bear *design constraints* that can be flexibly assigned to *design elements* during the project.
- All other important information for documenting a decision can be added on demand as unstructured descriptive text.
- For important strategic decisions, a template can provide *prescriptive* elements to assure these decisions have been made thoroughly.

The usage of the decision model has effects on existing *traceability* models. The *traceability* model of simple linkage described in ch. III.18 is extended to a model briefly sketched in fig. 20-3. Since *design elements* influence the decision process as well, the requirement dimension migrates to a close coupling with the design. Simple <<*satisfy*>> relationships can occur next to (as *Req.1* maps to *DesignElement1*) more complex *traceability* networks. Thus, e.g., *Req.2* only *impacts* the design by the decisions *Dec.1* and *Dec.2*.

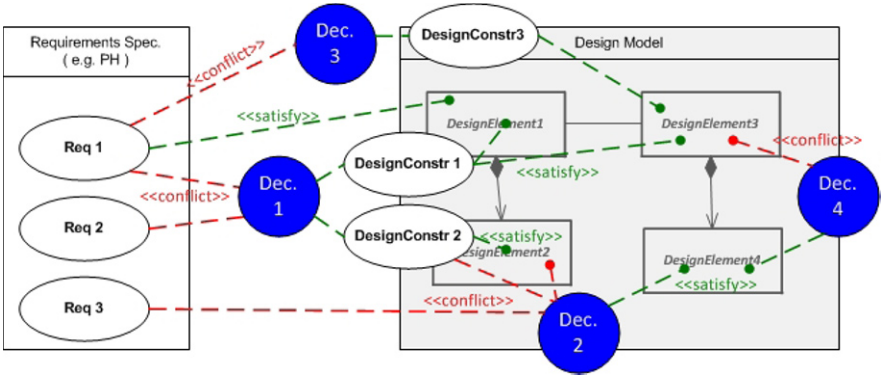

Figure 20-3 The newly emerged and more detailed *traceability* information scheme

Dec.2[241] arises from the conflict situation of *Req.3, DesignConstr.2* and *DesignElement2*, whereas *Dec.3* is only derived from requirement *Req.1* (which then corresponds to a *<<derive>>*-relationship as described in [Li94; p.33]). Consequently, *design elements* (alone without *RIs* involved) should also be able to invoke a decision (*Req1→Dec.3→DesignConstr3*). This way, chains of decision sequences can be modeled corresponding to experiences of Lewis et al. [LRB96] describing design as a *suite of problems* (ch. II.9.3.2).

With adequate tool support, these *traceability* relationships indicated in fig. 20-3 could be visualized as a *traceability* tree. A kind of browser should support:

- Detailed *IA*: Starting with a *starting impact set*, all subsequent paths would firstly be classified as *impacted*. During the following detailed check, the tool should allow to take out paths identified as none-relevant and adding paths detected as relevant (cf. ch. III.22.1).
- An adequate context for the simple *retrieval* of documented decisions. The following chapters show how R2A supports this.

III.20.3 Example How to Tame the Development Process Model of SPICE

In ch. I.7.3.2, problems of the SPICE *process model* concerning artifact handling and *traceability* are sketched. The major problems are unnecessary redundancies and lacking abilities to make implications between different model artifacts transparent (in the example case discussed here between the HW and the SW). The process artifact strategy described by fig. 7-3 (see ch. I.7.3.2) could improve the redundancy problem, whereas the second problem is still open.

Directly relating to fig. 7-3 (see ch. I.7.3.2), fig. 20-4 shows how this problem can be solved by using the *decision model* described here. The architect discovers the same problem concerning watchdog and EEPROM. He (she) opens a decision wizard and marks Req.1 and Req.3 as conflicting and links the decision to the "HW design" *AN* with the diagram documenting the conflict. As further *rationale*, the architect textually documents "synchronization conflict at SPI between time intensive EEPROM application and time critical watchdog application". A further click helps the architect to put the conflict into the risk list. As resulting *DC*, the architect sketches the cooperative handshake and links the *DC* to the EEPROM and watchdog *design elements* in the *SW design*.

---

[241] *Dec.2* is directly mapped to *DesignElement4*. This may also be possible, when no further information for understanding the decision is needed.

Our implementation follows the ideas described in the previous chapter. In the further project progress necessary changes are early detected by *IAs* (see ch. III.22.1) and the additional costs can be compared to the cost savings of the rejected HW change.

The artifacts *HW_RS* and *SW_RS*, which have not been realized, can be generated out of the model on demand by summing up all requirements related to the corresponding design (*HW design model* for the *HW_RS*, *SW design model* for the *SW_RS*). Ch. III.23.2 describes this in detail.

As it is a known problem in embedded design [Gr05; p.415], this example further shows how the *decision model* improves the design processes by making the strong influence of HW design on SW more transparent.

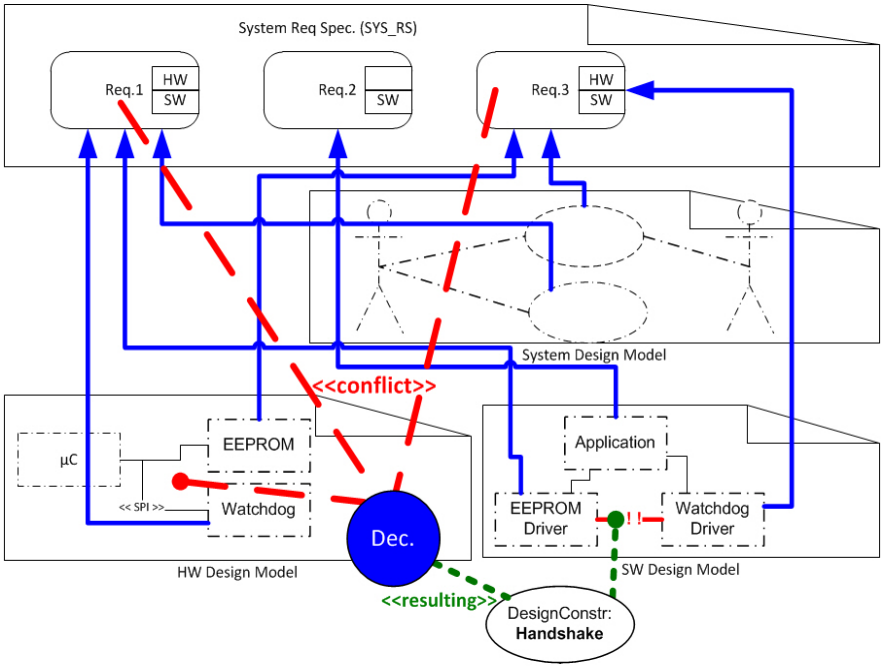

Figure 20-4 The example of SPICE conforming design processes in the new way

III.20.4 Implementation of the Decision Model in R2A

After the decision model has been theoretically discussed, this chapter will now outline how the decision model is implemented in R2A. Fig. 20-5 shows a decision modeled in R2A's decision dialog (left side). Additionally, fig. 20-5 shows possible *drag-and-drop operations* to relate information between the decision dialog and R2A's main window (right side). The modeled decision deals with how the *NFR* "ReqSpec_14: The system must be flexible to change." can be realized concerning HW and SW.

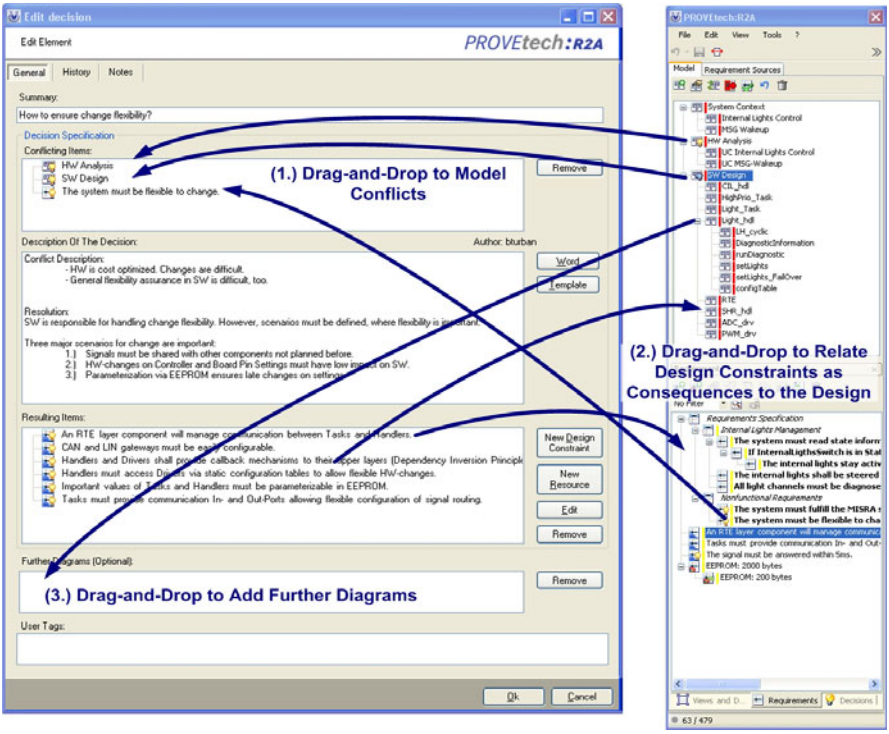

Figure 20-5 Decision dialog in R2A

The dialog implements the decision model described in fig. 20-2 (p.318), and has the following sections (see fig. 20-5):

- At the top, a summary or topic of the decision must be provided. The summary is displayed as the decision's item text in all other controls (e.g., see fig. 20-6).
- In the “Conflicting items:” section, all R2A items being in conflict with each other (and thus need to be decided about) can be added via *drag-and-drop operation* (1.). Once this decision is then saved, the items are related to the decision through <<conflicting entities>> relation described in fig. 20-2 (p.318). In the example, these items are the design *AN*s representing HW and SW in combination with ReqSpec_14.
- Further *assumptions*, *arguments*, and *rationale*, as well as any other information can be added in textual form in the “Description of the Decision” part. The approach does not prescribe any information provided here. Through the button “Word”, the description can be performed using Microsoft Word, thus allowing using formatted text. The “Template” button allows loading specification templates if some more structured (*prescriptive*) *rationale* approaches shall be used. The approach does not rely on a specific *rationale* structuring method. Correspondingly, the conflicts, and results parts form a kind of *semi-formal skeleton* for structuring the *rationale*. But, for further documentation of the *rationale*, the approach does not rely on any specific style documentation as *IBIS*, *QOC*, *DRL* etc. Instead a word style documentation is possible, where a template can be prescribed that could be in any *rationale* structuring template[242]. This can be seen as an advantage, because the *rationale* documenter can choose a best-suited structuring *schema*. As Dutoit et al. emphasize [DMM+06a; p.7], schemes differing from the way the *rationale* documenter would intuitively structure it create “a *cognitive dissonance*” imposing additional cognitive strains to the documenters. Freedom of choice can here provide a decisive difference alleviating the burdens encountered at *rationale* documentation.
- To derive consequences from the decision, *DCs* can be created in section “Resulting Items”. Afterward, these newly created *DCs* can be *assigned* as *RIs* to any *AN* via *drag-and-drop operations* (2.). Correspondingly, *DCs* could also be termed as 'requirements emerging from the design and decision processes'.

[242] It would even be possible to combine the model described here with other *rationale* capturing tools as gIBIS or Compendium.

- For further decision specification any diagrams showing important information can be added via *drag-and-drop operation* (3.) into the 'Further diagrams' section.

The decision modeled in fig. 20-5 is visualized in R2A as shown in fig. 20-6. Ch. III.22.1 and ch. III.22.2 describe how this decision structure and visualization are used to improve *IA* and *consistency management*.

The *DC* "Handlers and Drivers shall provide callback mechanisms to their upper layers (Dependency Inversion Principle)" indicates another aspect to consider. Callback mechanisms can be seen as *patterns* (or *idioms*) to decouple modules. In this way, the decision mechanism can be seen as a way to document *pattern* usage, where a designer can even prescribe the application of *patterns* for a specific situation through *decisions* and *DC*s. This is further discussed in ch. III.20.5.1.

Besides this aspect, the example also shows a situation, where a *NFR* (ReqSpec_14) is reexpressed through several more functional *DCs*. The strategy of taming *NFRs* by concrete scenarios or reexpress them by more concrete *FRs* has been already discussed in ch. I.5.1, ch. II.9.5, ch. II.10.4.2.2, and ch. III.18.2.2.

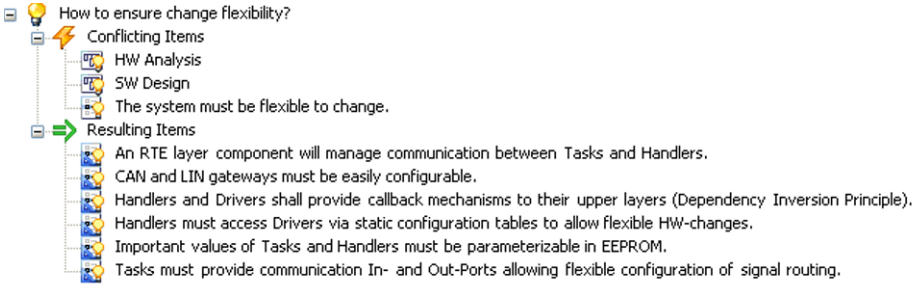

Figure 20-6 R2A's visualization of the decision taken above

Theory of *SW architecture* development has developed the so called *influence factors assessment* described in ch. II.9.5. This can be seen as a more general view on this topic in the context of design.

Table 20.1 shows an example of an *influence factors assessment* on the case study described here, orienting itself on findings of [PBG04; p.79], [CBB+03], [BCK03], [Bo00b], [HNS00], and [BCN+06]. The tabular presentation is taken over from Hofmeister et al. [HNS00]. In the first column, the factor is described, the second column discusses the priority and flexibility of the factor, the third

column identifies the influences and risks that may be involved when the factor takes effect, whereas the fourth column describes handling strategies to proactively reduce negative influences and risks of the factor.

The following factors have been identified and discussed:

1. Some requests for the ECU must be responded within 5 milliseconds (ms) (*nonfunctional timing requirements*). As these requests must be fulfilled within this timing to ensure that the controlled processes work properly, the priority is high and the influence of not fulfilling the timing restrictions can lead to complete failure of the ECU. Fortunately, the timing restrictions are not completely fixed but can exceed by 0.5 ms in 5 % of the cases, but 5 ms are still difficult to achieve. Correspondingly, continuous measuring and monitoring, or schedulability analyses as provided by *rate monotonic analysis* [KRP+93] can be an adequate strategy to ensure that all timing restrictions can be fulfilled.
2. A *NFR* requires minimizing power consumption in order to reduce problematic battery work load. This issue also has high priority, but only when ignition is off. As consequence, a sleep-wake-up manager in SW must manage that the ECU goes into a sleep mode when ignition is off.
3. Current *HW design* requires reading input signals of shift registers. This issue results from internal *HW design* decisions for cost optimization and is not demanded by the customer. Correspondingly, priority is low and flexibility is high. As major drawback, the input provided by shift registers must be polled continuously. This imposes a direct risk for factor 1. This also induces a high risk for factor 2, because some of the input signals are dedicated to wake up the ECU, when it is in sleep mode (see factor 2). When shift registers are used for these pins, the ECU must wake up continuously and poll these shift registers during sleep mode, which leads to higher power consumption in sleep mode. To fulfill the wake up requirements in the current *HW design*, the *SW design* for the current SW version must provide an extra timer with a time slice of 2.5 ms for polling the shift registers (2.5 ms in order to handle requests concerned with factor 1). Nevertheless, as this again imposes high risk for factor 1, the *HW design* must be changed for the next release to employ multiplexers instead of shift registers, because multiplexers allow wake-up-able pin interrupts at the micro controller to be directly triggered, thus avoiding polling for input signals and reducing risks of not fulfilling factor 1 and 2.
4. Factor four addresses change flexibility in software as it has been discussed above in fig. 20-5. As change flexibility is rather abstract, the *NFR* is concreted by defining three concrete scenarios for change flexibility:
   a. Scenario one discusses what will happen if input signals currently measured by the environment are sent from another ECU over CAN. In this

project, the scenario could be identified as low priority and is thus not further considered.

b. In scenario two a situation is addressed in which it is not quite clear whether some output signals currently sent via CAN may not also be provided via other out pins. Due to limited output pins of the micro controller, the usage of multiplexers (MUX) will then be necessary. The probability of this problem is medium and the change must be applicable within one month. Consequences would be that these output signals should be configurable by EEPROM parameters, HW must be changed, and a new SW component (MUX_hdl) handling these MUXs must be included. Negative *impacts* of the factor can be addressed by a HW reserve[243] that allows easily integrating the multiplexers on HW and an integration point to easily integrate a potential MUX_hdl to be easier integrated in SW.

c. The third scenario discusses the potential that internal SW signals within the ECU may have to be propagated to other parts of the ECU's SW. This is very likely and must be realizable within a few days, because otherwise implementation of other features needing the signals will get retarded. An extension of signal propagation imposes new efforts on the different SW tasks (processes) and may impose a risk for factor 1. To avoid these risks, an *RTE*[244] component as a decoupling layer between *tasks* and *handlers* may provide a standardized communication mechanism with configurable signal propagation through function pointers combined with asynchronous messaging mechanisms to decouple processes.

5. Factor five addresses the effects when development processes with SPICE *maturity level 2* (*ML2*) must be employed. The priority is high, because the customer demands for high quality and a scalable development process. On the other side, SPICE *ML*2 demands high administrative and bureaucratic effort for documentation inducing high risks for factor six. This requires a good tool support in order to diminish unnecessary effort; but in the same way it may be acceptable to use development processes capable for SPICE *ML*1, as SPICE *ML*1 also requires that all necessary processes are fulfilled; but it does not require extensive documentation.

---

243 German: HW-Vorhalt

244 RTE is inspired by the run-time environment (RTE) component of the AUTOSAR architecture. AUTOSAR (Automotive Open System Architecture) is a standardization en–deavor with the goal to define an open standard for automotive *SW architectures* [We07; p.18]. The design case study introduced here is not an AUTOSAR conforming design, because it would unnecessarily complicate the case study. However, the RTE concept proved a good idea to be integrated into this example about SW architectural design decisions.

6. Concerning the project resources, budget for three developers for two years is available. At first sight, this issue seems not so important, because HW part costs are at the end the dominating cost factor in the end. On the other side, risks to achieve the goal of factor 6 are significantly imposed by factor 5. This issue may at first also just seem to be a matter of planning in the sense that the project manager just performed wrong effort estimations, because he did not consider the extra effort of SPICE *ML* 2. In this sense, project staff simply must be increased; but on the other side it may also be the case that budget requirements imposed by the customer or management do not allow an increase in budget and other strategies must be taken. Generally, it is to say that factors 5 and 6 seem not to be directly connected to the design; however, as indicated by Posch et al. [PBG04; p.74f], the scope of factors to be considered should include a wider perspective in which especially organizational factors[245] should be considered. The example shown here is only a snapshot of the factor analysis at a very early state of the project, where factor 6 is in conflict with factor 5, but the effects on the *architecture* are not yet obvious. Now, in the further project progress it may become apparent that the customer insists on SPICE *ML* 2 processes and that project budget is very tight preventing to call in further developers. It may turn out at this later point, however, that two former projects are existing handling partially similar issues as the example project and parts of their SW components can be adapted to the new problem. As this promises to significantly reduce development effort and staff needs, it is then decided to reuse parts of these projects. In this case, both factors would significantly raise their influence on the design, leading to the effect that the whole character of the design may change (e.g., the design may then rather become an integrative patchwork to integrate the old components with adapter components to fulfill the new needs).

[245] Organizational factors such as staff size, staff skill levels, development organization, or available budget often impose significant restrictions on which solution is possible and thus significantly influence on the outcome of a design [PBG04; p.74]. Especially economic and development process contexts play an important role, because they soon become an important factor about the feasibility of an intended solution.

Table 20.1 Example of an *architectural influence factors assessment*

Factor	Priority/Flexibility	Influence/Risk	Handling Strategies
1. Response time < 5ms	HIGH; in 95% +- 0.5ms → soft deadline	K.O.-criterion	Rate Monotonic Analysis + continuous measurements of prototypes and release candidates.
2. Minimize Power Consumption	HIGH; at least when ignition is off.	ECU must go into a sleep mode.	Sleep-wake-up manager in SW.
3. Input signals over Shift Register Handler	LOW; High flexibility as not prescribed by customer	Through needed polling induced risk for 1 and 2.	Timer with t+2.5ms; HW change from shift-register to multiplexer in next release.
4. Flexibility to change	MEDIUM		
4.1 Scenario: Input signals change to CAN.	LOW; Rather low probability	-	-
4.2 Scenario: Output signal via CAN or multiplexer	MEDIUM; must be realizable within one month	Configuration parameter in EEPROM; HW change; Multiplexer handler (MUX_hdl) in SW necessary.	HW reserve; Integration point for MUX_hdl in SW.
4.3 Scenario: Internal signal processing must notify other parts of the system.	MEDIUM; very likely → must be realizable within a few days	New communication effort with other tasks → Risk for point 1.	RTE-layer with configurable function pointers and asynchronous messaging
5. SPICE *ML*2	HIGH, the customer demands for high quality, but also wants a scalable development process.	Increased administrative effort → Risk for Point 6	a) Usage of adequate tools. b) Negotiations whether SPICE *ML*1 may also be adequate. c) Adding 50 % additional developer resources
6. Project resources: Three developers for two years	LOW, costs are mainly driven by HW costs.	-	-

Following the current design theory the *influence factor assessment* example above described would be part of a design description only loosely connected to the *design model*. With the *decision model* described here, the decision can be directly integrated into the *design model* (cf. fig. 20-7).

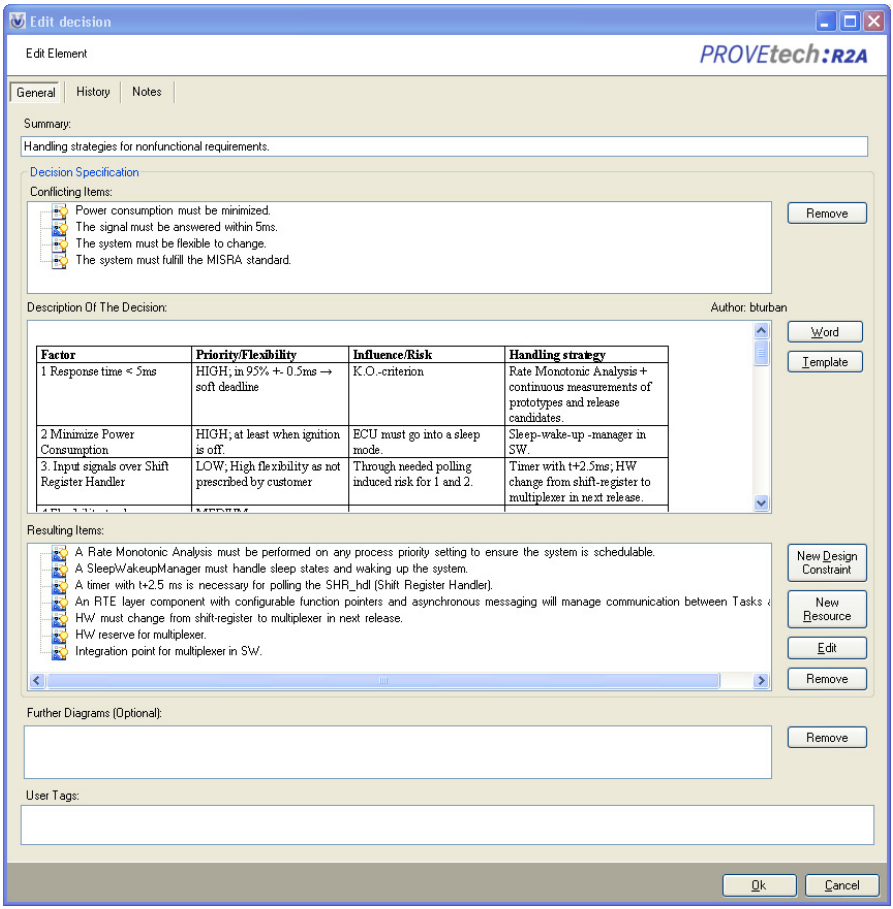

Figure 20-7 *Architectural influence factors assessment* with R2A's decision model

All *requiremental items* (*RIs*) or design related elements (*ANs*) present in R2A and being considered as influence factors can be added to the "Conflicting Items:" section. The assessment description can be documented in the "Description Of The Decision" section in an equal way as shown in table 20.1 above. The arising consequences (column "Handling Strategies" in table 20.1), can again be derived as *DCs* thus allowing directly assigning the *DCs* to the *AN*s needing to realize the consequences. At first, this helps to ensure that the designers of the

corresponding *AN*s become aware of these demands and thus ensures that these demands are considered by the considered in the design. Secondly, this also ensures that this information is made directly traceable and thus ensures that this information is present later in *IA*s for change assessment during *change management processes* (cf. ch. III.22.1 and ch. III.22.2).

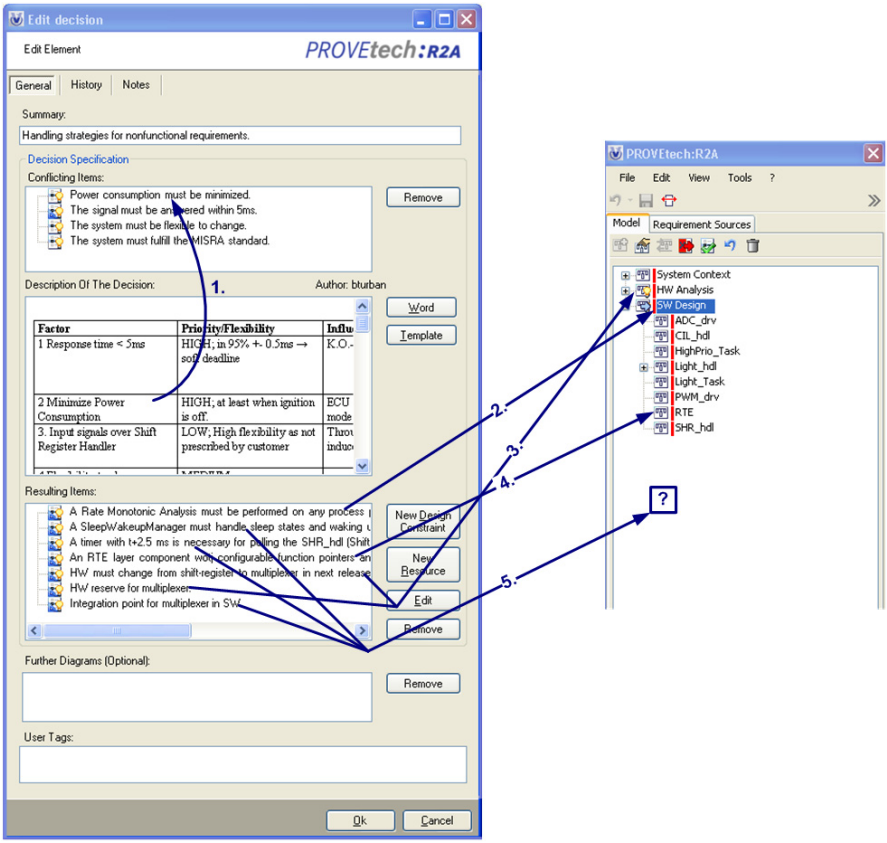

Figure 20-8 Consequences of the *architectural influence factors assessment* of fig. 20-7

Fig. 20-8 shows in more detail how the *influence factors assessment* of fig. 20-7 could impose consequences (see the different arrows) on the design and how they currently can be made explicit in R2A.

Arrow '1.' indicates a fact not yet directly discussed but possibly often occurring in design: The process of discovering *rationale* about a decision can also impose backlashes on the original sources of the decision such as the requirements involved. During the decision process of the example, the designer discovered that the requirement about minimized power consumption is only important if ignition is off, because otherwise the running motor drives the power generator generating enough energy to not strain the battery. This discovery could lead to the conclusion that the requirement itself should best be adapted to 'Power consumption must be minimized if ignition (KL15[246]) is off'. Currently, R2A does not provide dedicated support for this situation, because the situation can be managed by current state-of-the-art tooling. If, e.g., a *change management tool* with a *change proposal system* is used, the designer can initiate a change request describing the situation and the designer can directly textually refer to the decision via its unique identifier in R2A (cf. ch. III.17.4). Otherwise, if only an *REM-tool* is used, the textual reference to the decision's identifier can be added to the information about the requirement (e.g., in a comment attribute or 'Origin' attribute as described in II.10.4.2.1). However, as ch. IV.26 outlines, further perspectives of research about R2A could be supporting a dedicated integration with *change management tools*.

The first *DC* in the “Resulting Items” section demands to perform a *Rate Monotonic Analysis*. Arrow '2.' shows how this can be modeled as a nonfunctional consequence for the complete *SW design*. By assigning the *DC* to the *AN* “SW Design” via a *drag-and-drop* operation the *DC* becomes a new *nonfunctional RI* for the *SW design*. Arrow '4.' indicates a similar situation for a part of the *SW design*.

On the other side, arrow '3.' imposes consequences on the HW. As indicated in fig. 20-8, if the HW *design* is also somehow represented in the R2A *design model*, this can be performed by a *drag-and-drop* operation to the corresponding *AN* representing the HW. Currently R2A does not support a modeling tool for dedicated *HW design*, but the *product line* concept with dedicated support for integrating different modeling tools as a *variation point* should, in principle, equally allow connecting any *HW design tool*.

[246] In automotive terminology, ignition is coded by the term “KL15” (In German: Klemme15).

In the current state of R2A with lacking direct support of a HW modeling tool, two alternative strategies are possible to allow tighter integration of *HW design*:

- A place-holder *AN* for the HW model can be created, where all issues arising[247] from a design process performed in R2A possibly relevant for the HW can be assigned. As further described in ch. III.23.2, this place-holder *AN* can then be used in R2A to generate a *requirements specification* for HW resulting out of the design processes performed in R2A.
- In the author's experience, any embedded system or *SW design model* must integrate certain HW aspects anyway in order to model certain cross influences. As these models might need some aspects of HW in their models anyway, a certain low detailed *HW model* could be collaboratively maintained (resp. sketched) by *system designers*, *HW designers* and *SW designers* together to improve a common understanding at this core interface, in which the three domains have their significant overlap. If this *HW model* could be maintained in UML, the system and *SE* activities could seamlessly integrate the model. As a side-effect this model could also be an interface communicating effects of design processes performed in R2A to the *HW designers*. In fig. 20-8, the author indicates this idea by including an *AN* 'HW Analysis'[248].

Last but not least, arrow '5.' indicates that new *DCs* might also spark the need for modeling new *ANs* in the *design*. In fig. 20-8, for example, a *DC*[249] demands a SleepWakeupManager. This SleepWakeupManager must be modeled as a new *AN* in the design. These situations sparking new *ANs* are indicated in fig. 20-8 by a square containing a question mark. The question mark indicates that it is not yet quite exactly sure in the current design situation whether these possibly new arising *ANs* really come to existence and how they might then exactly look like, because creating any new *AN* would then be some following deci-

---

[247] These are at first *DCs* as consequences of decisions as described here in this example, but perhaps also other items in R2A as, e.g., the *budgeted resource constraint* concept introduced in the next chapter, might be relevant.

[248] The author has chosen this name, because such a model concept – in the author's opinion – rather resembles to the *SW analysis* concept as such a model might not really anticipate the *HW design* but might help to analyze certain HW parts that are of cross-cutting interest for all three design domains. A real *HW design* might only make sense with a dedicated *HW design tool* allowing modeling of the HW circuits. An alternative name for such a model might be 'HW intermediate model'.

[249] The reader should note that in the situation described here actually three *DCs* might spark new *ANs*. The author has grouped the three items together to one arrow '5.' to avoid unnecessary clutter in fig. 20-8. It is very highly possible that the three items might spark the existence of three different new *ANs*.

sions of the designer, where other factors may also influence the final decisions. As described in ch. II.10.4.2.2, such consequences as indicated by arrow '5.' are connected to what Knethen and Paech [KP02; p.14] call '*applicability links*' meaning that an item can derive its justification from another item. From this perspective, the decision and *DCs* concept might probably also be seen as a special form of '*applicability links*'.

In [PKD+03; p.145], Paech et al. indicate that some *NFRs* can be specified via *FRs*. This is possible with the *decision mode*l, in which a *NFR* can spark a decision about handling strategies for the *NFRs* leading to new *DCs* as consequences[250]. Thus, it could be said that this approach is good way to cope with nonfunctional restrictions that can be split into some numerical expression as it is often the case in embedded systems.

Chung et al. [CNY+00] developed a *NFR framework*, where *NFRs* drive design creating *rationale*. The approach allows graphically modeling trade-offs and synergies between *NFRs* (also cf. ch. II.9.5). This can also be achieved by R2A's decision model, where the *NFRs* are referred to as conflicting items. Via the "Further Diagrams" section, a model graph can be modeled in the *design tool* and referred to in the decision. In a similar direction, Egyed et al. [EG04] discuss an approach, where they map *FRs* to *nonfunctional aspects* (or *software attributes*) to identify conflicting and supporting situations. This approach should be equally manageable by R2A's *decision model*.

III.20.5 Additional Support of the Decision Model for Designers[251]

In the following, additional connections and advantages of the proposed decision model in relation to design-related issues are discussed.

---

[250] As an example, a *NFRs* demanding code flexibility could be handled by a decision to employ the *visitor pattern* [GHJ+95; p.301-318] to alleviate adding new operations to the data model. As consequences, *DCs* can be derived defining that data model classes must fulfill the characteristics (operations to accept a visitor) of concrete (visited) elements, whereas operations must fulfill characteristics (operations to visit the different elements) of a visitor. This example can also be seen as an example for the claims made in the following ch. III.20.5.1 that the decision model of R2A has close connections to the *pattern* concept.

[251] Extended parts of this chapter have been published in [TKT+09; ch.5].

III.20.5.1 Patterns

"*Patterns*, as used in software engineering, constitute one of the most heavily used approaches for organizing reusable knowledge" [DMM+06a; p.19]. *Patterns* (ch. I.6.2.4) define the abstract core of a solution for a continuously recurring problem, thus allowing the solution tailored to the concrete problem to be reapplied [GHJ+95]. *Patterns* are described using a structure *template*. Even though different authors use slightly different *templates*, the description of the problem (often referred to as forces), the solution and its consequences are part of all *pattern templates*. The *decision model* discussed here can be described in terms of such a *pattern template* (see also [HAZ07; table 1]): The *conflict situation* of the *decision model* corresponds to the *problem description* part in *patterns*, whereas the description of consequences in a *pattern* description could be modeled by resulting new *DCs* in R2A's *decision model*. Due to this analogy, the author believes that this approach can provide valuable support in selecting *patterns* (e.g., the *conflict situation* of a decision can indicate the usage of a specific *pattern*). At the same time it can help knowledge engineers to identify interesting solutions as new *patterns* (for the relationship between design decisions and *patterns* also refer to [HAZ07], [PBG04; p.209]). A *pattern library* for decisions in modeling embedded systems could be the ultimate goal of such an effort.

Horner and Atwood [HA06a; p.76] characterize *patterns* (ch. I.6.2.4) as common solutions resolving conflicting tendencies. The *decision model* proclaimed here also supports analyzing conflicts and results. In the author's eyes, the decision model supports identifying matching *patterns* and identifying new *patterns* as described in [TKT+09]. In this way, the R2A has certain resemblances to the DRIMER tool [PV96] (see ch II.9.3.1).

Cleland-Huang and Schmelzer [CS03] (see also [GG07; p.315]) introduce another connected approach. Their concern is to improve *traceability* of *NFRs* to design. Due to the often global and far reaching effects of *NFRs* on design, *traceability* of *NFRs* to design is difficult to handle adequately. As a solution, they propose to use *design patterns* as an intermediary model between *NFRs* and the design. This means that *NFRs* are not directly mapped to design. Instead, *NFRs* are mapped to a *design pattern*, which then again is mapped to design. In this way, the number of *traceability* links to be manually captured is reduced. The approach then uses this information to automatically derive the relations between *NFRs* and the design through the manually captured relations.

In the author's opinion, however, this approach has the following shortcomings:

- Not all *NFRs* can be directly mapped to specific design *patterns*. Some *NFRs* may also be handled through other strategies[252].
- The approach does not consider crossinteractions between *NFRs* or other *FRs*.

Correspondingly, ch. III.20.4 shows that R2A's mechanism may be more powerful as it also allows describing handling strategies apart from *patterns* and also allows describing crossinteractions (see, e.g., the described *influence factors assessment* in ch. III.20.4).

III.20.5.2 Ensuring Adequate Realization of Design and Decisions

As Posch et al. [PBG04; p.38] underline, architects also have to ensure that their design settings are adequately considered and realized by other designers or coders. Using this *decision model*, designers can model the consequences of a decision as *DCs* and assign the *DCs* as new "requirements" (in *R2A* terminology: *RIs*) to *design elements* that must then fulfill the *DCs*. Besides usage in further design or coding processes, the list of assigned *RIs* to a design item can also be used as basis for reviews on design and implementation of the item.

III.20.5.3 Support for Architecture Evaluation

The *R2A* approach can also provide valuable support in maintenance and *evaluating architectures* [CKK02]. Moro [Mo04; p.321] points out that the usage of patterns and other decisions must be documented for later maintenance and *architecture evaluation* issues. According to Reißing 80% of change effort is caused by wrong *architectural decisions* [Mo04; p.90]. With documented decisions and *rationale* at hand, potentially wrongly made *architectural decisions* may be easier and earlier identified in *architecture evaluation*. In this way, implementation of wrong decisions and thus later costly changes may be avoided.

When evaluating *design documentation* during design evaluation meetings, Karsenty [Ka96] found out that questions about *rationale* have been the most

[252] E.g., *NFR* about security may also be handled by a login and password component (prevents unauthorized access) in connection with cryptography mechanisms (prevents eavesdropping) and intensified quality assurance methods (prevents bugs susceptible for hacking).

frequent questions (approx. 50%), but only 41% of these questions could be answered (also cf. [HA06a; p.83], [BB06; p.275]).

The idea of the *decision model* is to allow *DCs* (and *budgeted resource constraints* see ch. III.21) as consequences and attaching them to sub elements also provides direct benefit for the designer himself, because he can clearly model his demands for components and in later reviews these demands can be assessed directly. Through the structure of the *decision model*, further *rationale* is already present, where designers might even have used the description text to document further *rationale*.

As already addressed in ch. II.9.4.1, a further helpful concept in this relation is the identification and tracking of *neuralgic points* in design [Mo04; p.310-330]. As Moro found out, developers are often aware of *neuralgic points* by themselves, because *neuralgic points* often recur back as issue of discussion. R2A's decision mechanism gives designers a means at hand to document new discovered *rationale* at those recurring discussions. Further, the author believes that it may also be possible to discover *neuralgic points* through the sheer amount of documentation attached to a decision. In most cases, the most extensive documentation may thus be provided to decisions touching *neuralgic points*, because the developers are often anyway aware of the *neuralgic* nature of an issue.

Other possibilities to identify *neuralgic points* through documented decisions may be to identify a metric for measuring the complexity of decisions. As a start, e.g., it may be possible to assess the number of items identified as part of the conflicting area of a decision. If this number exceeds a certain number (e.g., 15 to 20) the decision can be considered as especially complex. However, this topic should be further researched and be filled with experiences from practical usage. A further idea might be to implement a mechanism to analyze the click behavior of the designers. If certain decisions are often clicked at and further analyzed (e.g., when the properties of the decision are opened), it may indicate that this decision is more critical than decisions seldom being clicked at.

III.21 Resource Allocation as a Special Decision Making Case[253]

The requirements for design conflict and cannot be reconciled. All designs for devices are in some degree failures, either because they flout one or another of the requirements or because they are compromises, and compromise implies a degree of failure. ... It follows that all designs for use are arbitrary. The designer or his client has to choose in what degree and where there shall be failure.
[Py78; p.70]

In design activities for embedded systems an additional decision type can be identified dealing with *non-functional aspects* of limited resources such as memory resources (e.g., Read Only Memory (ROM), Random Access Memory (RAM), Electrically Erasable Programmable Read Only Memory (EEPROM)) or timing restrictions.

A core goal of embedded design is the effective administration and distribution of such resources[254] and different strategies for handling this problem exist:

1. The allocation is a more or less unconscious or uncontrolled process (i. e., no explicit strategy is established).
2. A *resource estimation* is performed as part of the design and estimations are checked and adapted at each development cycle.
3. Resource allocation is explicitly modeled in the *design model* (e.g., by using UML profiles such as the *UML Profile for Schedulability, Performance, and Timing* [Do04, ch.4] or *MARTE* [EDG+06].

With respect to collaboration in complex development teams or organizations, approaches 2 and 3 have limitations in the following aspects:

- Propagation and communication of changes to all team members involved in the change can be cumbersome.
- Minimizing redundancies as a major source of inconsistencies can result in communication errors.
- The seamless adoption and refinement of other designers' design results can be extremely difficult.

---

253 This chapter bases mainly on [TWT+08].

254 In fact, also Simon acknowledges *resource allocation* to be an important aspect of design [Si96; p.124-125]. Correspondingly, *resource allocation* can be considered as an important aspect of every design, but in embedded design its importance is highly more significant. When the engineering standard Automotive SPICE is applied, ENG.5 BP5 ("Define goals for resource consumption") even requests that resource consumption for each software module is explicitly planned and tracked [MHD+07; p.64].

- Sharing project knowledge in general will become more difficult.

The following example, basing on the accompanying case study (see ch. III.12), illustrates these shortcomings in more detail. The design shown in fig. 12-3 (see ch. III.12) may lead to the following estimation of RAM consumption (table 21.1) documented as a separate chapter in the design document of the high level designer.

Table 21.1 Example resource estimation of RAM consumption in design

Module	Light _Task	Light _hdl	CIL_ hdl	CAN-drv	PWM_ drv	ADC _drv	Buffer
RAM (1500 Bytes available)	600 Bytes	250 Bytes	100 Bytes	300 Bytes	100 Bytes	100 Bytes	50 Bytes

Such tables are a common format for documenting resource assignments in design documents (cf. [Mu04]). The tabular format has the main advantage that it easily gives an overview, but it has important weaknesses when collaborative aspects are considered:

- First of all, even though these assignments are typically called estimations, they should rather be treated as *RIs*. This implies that a mechanism must be in place to communicate these *RIs* on time to all interested stakeholders – especially if changes occur during project progress.
- Further, the allocation settings are estimated at a certain design stage and thus are an integral part of the design documents at this stage. Therefore, further processing of this information by other designers is difficult. In the case study, the estimations are made at the level of modules and included into the documentation of the high-level design. If the module designer of the complex *Light_Task* wants to refine the resource estimation into a more detailed estimation, a problem arises. In this case he would have to copy the information "*Light_Task == 600 Bytes*" into some document of his responsibility. This leads to unnecessary redundancy causing consistency problems when this setting changes later in the project.
- These problems are even more critical if some parts of the project are delivered by a subcontractor – as it happens to be the case in the example. In this case, all relevant requirements for the item to supply must be provided (as required by SPICE process ACQ.4 Supplier Monitoring, see [MHD+07]). In this case, the RAM estimations, since they are *RIs*, must be communicated as requirements to the supplier. This also leads to a high degree of redundancy with even worse effects if changes are not communicated.

III.21.1 *Budgeted Resource Constraints* as further *Requiremental Items*

In consideration of this problem a way to perform such resource allocation decisions in a handy fashion is needed, which also allows communication of the results for each considered *design element* throughout the entire project in an efficient way. An additional aspect here is the fact that the results of a decision act as new *RIs* on the design elements they are assigned to. As literature shows (cf. [BGT+04], [CBS+02], [FGS+01], [Do04; p.317], [Do03; p.169], [Mu04], [Gu03]), most resource allocation activities consist of numerically truncating a larger resource amount into smaller subsets –more or less in analogy with the *abstraction hierarchy* of a system's resp. software's design (see ch. III.15, fig. 21-4 resp. fig. 21-5 in ch. III.21.2.4 below). Obviously, this can be compared to the process of preparing and distributing budgets in business administration or project management area [HHS64]. Therefore, the taxonomy of *requiremental items* is enhanced by an additional type of *RI* called *budgeted resource constraint* (*BRC*) as shown in fig.21-1.

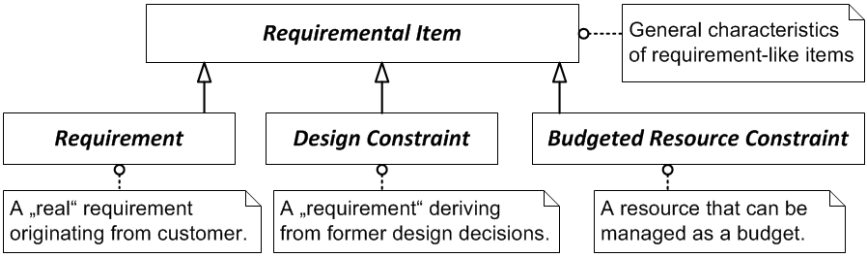

Figure 21-1 *Requiremental items taxonomy* with *budgeted resource constraints*

BRCs are similar to *design constraints (DCs)* as they represent the results of a decision making process and can be assigned as *RIs* to any *design element*.

However, there are the following differences when compared with other *RIs* (such as *DCs*):

- *BRCs* represent numerical values, whose associated *design elements* may not exceed the maximum value of the assigned *BRC*.
- A *BRC* can be subdivided into sub *BRCs*. Thus, *BRCs* at the same time represent a decision-making process as well as its results.

- As *BRCs* represent numerical values, whose sub *BRCs* divide resource amounts into smaller budgets for more detailed parts of the design, automatic consistency checks (e.g., tests for budget overruns) can avoid wrong allocations. Budget overruns may be detected at an early project stage.
- Individual *BRCs* can be added to one design item only, whereas requirements and design constraints may be added to several items.

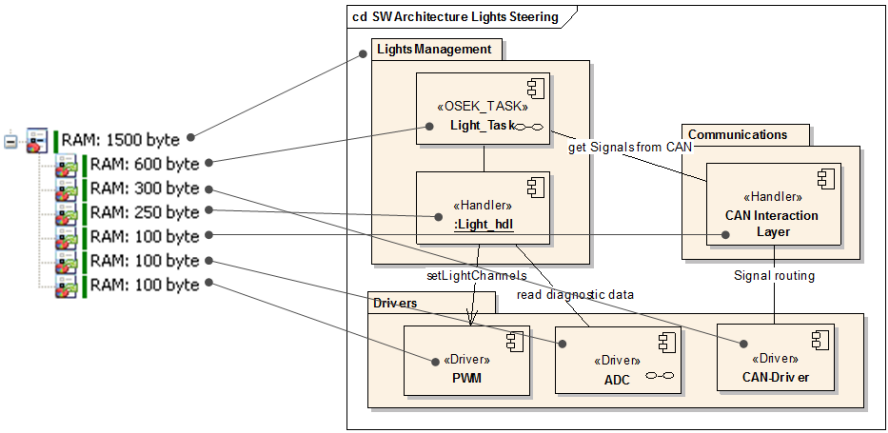

Figure 21-2 Resource allocation example with *budgeted resource constraints*

Resuming the example described above, fig. 21-2 illustrates the resource allocation problem presented using *BRCs* as implemented in R2A. The connections to the *design elements* illustrate so-called assigned to or *satisfy-link* types used in R2A to relate *RIs* to *design elements* (see description in ch. III.18.2). In R2A, all *RIs* assigned to an *AN* (thus, also *BRCs*) are displayed via the "Requirements" tab (fig. 15-4 in ch. III.15), but for better understanding they are here directly mapped on the design diagrams, where the shown elements on the diagram are *ANs* in R2A.

In this situation, the *SW architecture* is assigned to fit in a total budget of 1500 bytes of RAM. This *BRC* is subdivided into six sub *BRCs* assigned to the six modules in the *SW architecture*, thus showing a more detailed partitioning of the RAM budget.

Comparing fig. 21-2 with table 21.1 above, it can be seen that both representations have an equivalent meaning. In fact, the idea of budgets in HW and SW engineering is not new (cf. [FGS+01], [Do04; p.317], [Do03; p.169], [Mu04], [Gu03]). What this wants to point out beyond the appealing (and well-known)

aspect of a more or less easy mathematical model enabling consistency checks are the advantages of the budget concept itself, when it comes to collaboration and sharing project knowledge between project members. In this sense, the budget concept is used as a means of communication during software design. The following chapters will provide more details on this.

III.21.2 Advantages for Collaboration and Sharing Project Knowledge

The following situations of this example project show the value of the *BRC concept* for the following communication situations:

- Within project refinement,
- Communicating information over organizational boundaries,
- Change management,
- Different views on the same problem;

III.21.2.1 Within Project Refinement

During the first design cycle of the Light_hdl, the Light_hdl is forecast to have a very tight RAM budget. Therefore the designer identifies several specific aspects for which he arranges budgets according to his current information and needs (see fig. 21-3):

- In normal mode, the module uses the settings in EEPROM mirrored to RAM for steering the lights. RAM consumption depends on the number of steered channels and the number of bytes needed for each channel.
- The diagnostic part supervises regular checks of the electrical current between the ECU and the connected lights to detect malfunctions as short circuit or open drain. Malfunctions lead to the deactivation of a light channel.
- In the case of severe error conditions, e.g., loss of EEPROM data, the fail over mode assures that at least essential functions like brake lights and indicators work. The code and configurations are fixed in ROM, thus no particular portion of RAM is needed.

With the type of *BRCs* proposed here, designers of sub levels can directly continue to process results produced in previous design decision processes.

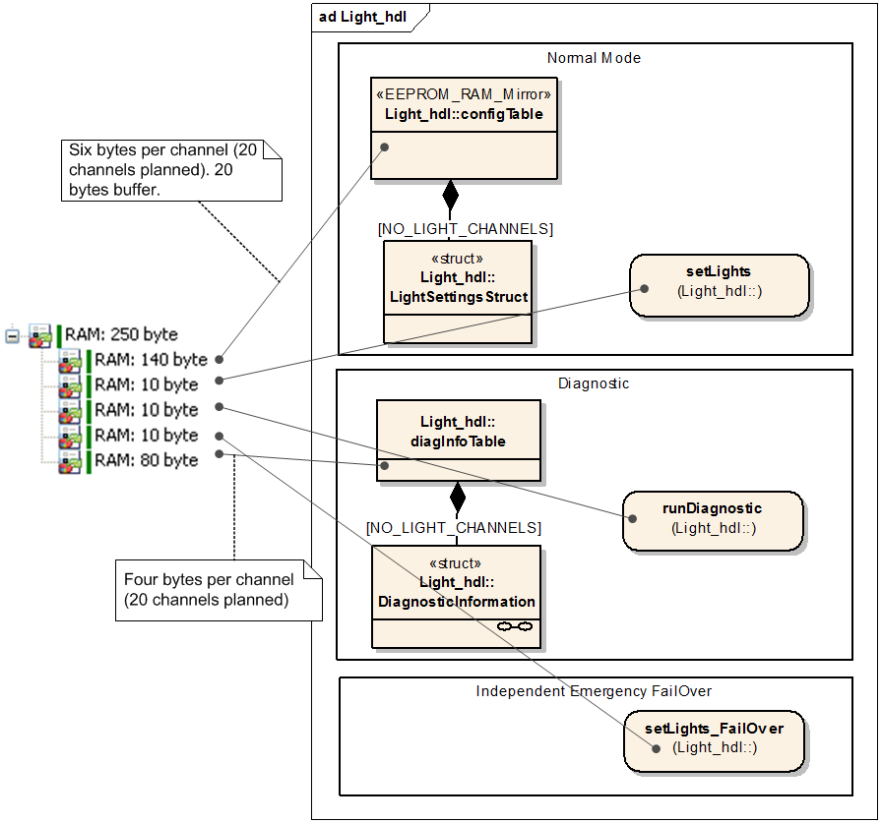

Figure 21-3 Sub budgeting of the Light_hdl module

III.21.2.2 Communicating Information across Organizational Boundaries

Information must often be provided across organizational boundaries. Such boundaries can be sub projects within the same company or between different companies. In the case study, drivers are provided by different subcontractors. This implies that all requirements for the drivers must be provided throughout all

parties involved. In the author's experience, *functional aspects* are communicated in a quite complete fashion, but such *nonfunctional aspects* (e.g., restrictions on memory, timing, etc.), resulting from former design decisions, are often forgotten.

The solution described here supports exporting all types of *RIs* associated with a *design element* as a new *requirements specification* into requirements management tools like IBM Rational DOORS, which can be delivered to the subcontractor. Since *BRCs* are treated as normal *RIs*, they are directly propagated to the subcontractors via automatically generated *requirements specifications*. In later development phases, these *requirements specifications* can be continuously synchronized with the settings in the *design element*, thus ensuring proper propagation of requirements to subcontractors.

III.21.2.3 Change Management

During project progress changes occur that force designers to change decisions and assumptions. Managing those changes efficiently is essential to avoid project deviations. Two heuristics should be considered:

- Changes should be kept as local as possible to avoid unnecessary complexity.
- Changes must be implemented in a consistent way.

Our model supports handling changes of *BRCs* as local as possible. Continuing with the example, it might happen that the “runDiagnostic” function needs more than 10 bytes of RAM (see fig. 21-3 above). In this case, the designer can first try to find an internal solution for the problem (e.g., find a way to cut down on some bytes in the “diagInfoTable”). If this is not possible, the designer can escalate the problem to a higher-level designer.

In another situation, new requirements from the customer could make the creation of a new, additional module necessary. This case has effects on the design as a whole since most of the modules already present might suffer a budget cut in their *BRCs* as a consequence. R2A visualizes changed *BRCs* (in a red color coding; cf. ch. III.22.2) to alert designers of sub-layers to analyze the *impacts* on their assignments.

If the sub designer has made his changes and consistency checks (e.g., detecting budget overruns) pass, the designer can mark the change as implemented. After this, the *BRC* is shown in normal mode.

III.21.2.4 Different Views on the Same Problem

In software design theory, the idea that different aspects of SW can be modeled by different views has been proposed (cf. [Kr95]). The same can be claimed for *non-functional aspects* modeled by *BRCs*. Besides the direct allocation view (see fig. 21-2 and fig. 21-3 above), R2A supports creating an enhanced table representation. Fig. 21-4 shows this tabular lineup between *BRCs* and their allocated *design elements*. Both columns additionally show their hierarchical break down.

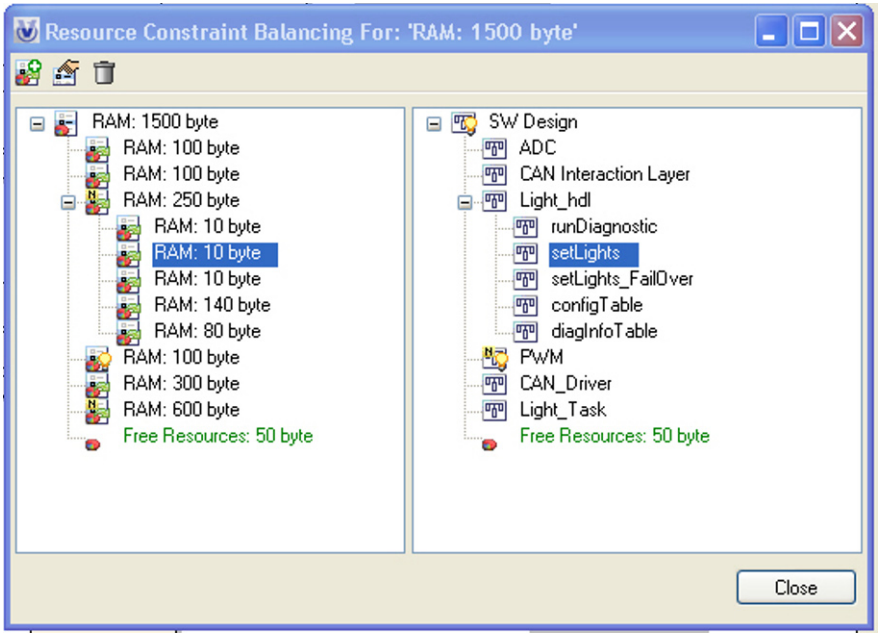

Figure 21-4 Tabular view with corresponding abstraction hierarchies.

Since the structure of the *BRCs* break down has a strong analogy with the breakdown of their associated *design elements*, design flaws of the assignment be can easily detected. Fig. 21-5 shows this situation, where a wrongly associated item disturbs the analogy, helping the designers to detect those problems easily.

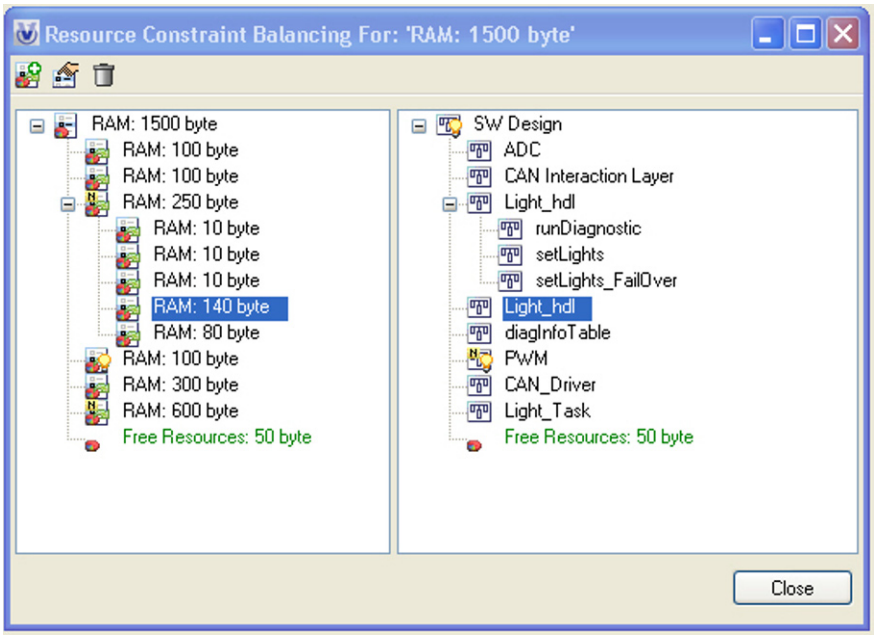

Figure 21-5 Tabular view with assignment inconsistency (selected line)

III.21.3 Representing Budgeted Resource Constraints in SysML

Another frequently used possibility of modeling resource allocations in UML[255]-design is to use UML profiles (e.g., timing constraints can be modeled in the *UML Profile for Schedulability Performance and Time* [Do04; ch. 4]).

[255] This statement does not refer to any specific version of UML as profiling is a general feature of UML.

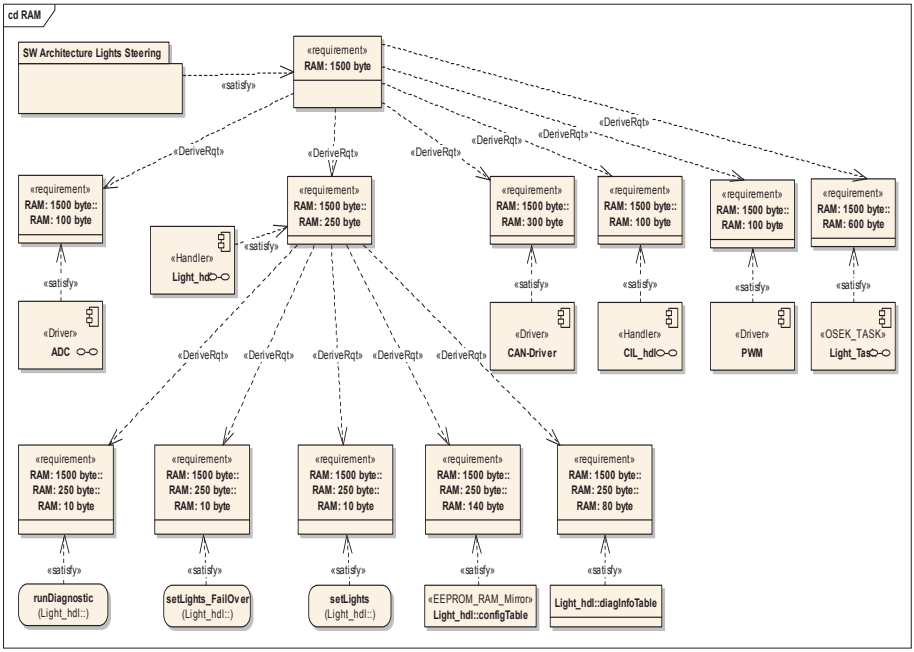

Figure 21-6 Representation of the same information as fig. 21-4 but in SysML view

In 2006, the Object Management Group (OMG) adopted an extension of UML called Systems Modeling Language (SysML; cf.[We06] and ch. I.6). SysML extends UML to improve support for Systems Engineering activities. A goal of SysML was to provide support for modeling dependencies between requirements and *design elements*.

R2A's model is compatible to SysML through the following definitions:

- *BRCs* are represented by the <<*Requirement*>> stereotype,
- Sub *BRCs* can be derived from the <<*DeriveRqt*>> relationship,
- *BRCs* are assigned to *design elements* via <<*Satisfy*>> relationships;

As a proof of this claim, R2A supports automatic generation of SysML diagrams from the *BRC*-model. Fig. 21-6 shows a SysML diagram generated from the model of the case study. However, it shows that such SysML-diagrams seem to have only limited value since they quickly can get very complex and cluttered. Thus, the real value of SysML might not be in the diagrams but the *meta model* behind it, being shown in different representations as R2A does in fig. 21-4.

Similar generation functions could be employed for timing budgets using the *UML Profile for Schedulability, Performance and Time* or the *MARTE profile* ([EDG+06]).

Except for prototypical implementation of the transformation between *BRC*-model and SysML described here, these topics have not been further pursued because R2A aims to embrace design methodologies and tools beyond the UML paradigm (as, e.g., Matlab Simulink or Stateflow).

III.21.4 Combining both *Decision Models*

As already described in [TWT+08], implementing a small change on the first decision model described in ch. III.20 allows making both decision models compatible with each other. If *BRCs* are allowed as possible results in decision model one, both models support compatible types as their major in- and out-comes (since all are *RIs* (cf. fig. 21-1)).

The following example described illustrates this in detail (see fig. 21-7). A *documented decision* "Dec1*"* determines the use of a specific micro-controller. This decision also determines a *BRC* "RAM:1500 byte*"*. Through several decision steps, a sub *BRC* "RAM: 10 byte" is derived that is satisfied by the "setLights_FailOver*"* function in design. Both conflict with a *requirement* "Req3", resolved by a new *documented decision* "Dec2*"*.

As the example shows, both decision models complement each other and allow modeling of more difficult decision problems.

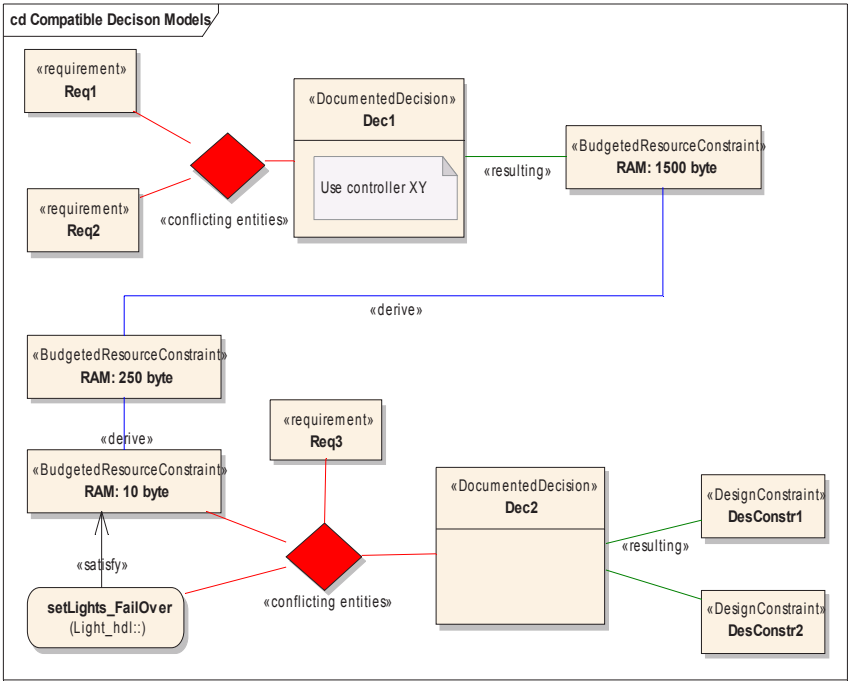

Figure 21-7 Example for combining both decision models together

III.22 Managing Changes and Consistency

Complexity is the path of growth. On the other hand, complication is the path of degradation, loss of control, evanescence of order.
Lem O. Ejiogu

Nuseibeh et al. [NER00] describe that it is not always viable resp. advisable to resolve all inconsistencies immediately. Even though resolving inconsistencies can only imply adding, changing or removing information, it more often involves balancing conflicts and taking design decisions. Correspondingly, “the choice of an inconsistency-handling strategy depends on the context and the impact it has on other aspects of the development process” [NER00; p.26].

The R2A mechanism allows keeping inconsistencies for a certain time but keeps also track of the inconsistencies so that they can be resolved later.

III.22.1 Usage of Traces – Managing *Requiremental* Changes

Ch. II.10.4.3.3 discusses the usage of traces recorded in *traceability* approaches. Pinheiro terms the usage of traces as *trace extraction*. Concerning *trace extraction* processes, Pinheiro [Pi04; p.105] describes three different tracing modes that should be supported. The following listing describes features provided by R2A to support the modes described in ch. II.10.4.3.3:

- *Selective tracing* is supported by the *impact analysis dialog*, where each element can be selectively applied to an analysis or deactivated. *IA* with the *impact analysis dialog* is described in the following sub ch. III.22.1.1.
- *Interactive tracing* is directly supported by a *model browser* described in the following second sub chapter III.22.1.2.
- *Non-guided tracing* is supported by the *model browser* as well as by other features described in the following third sub chapter III.22.1.3.

III.22.1.1 Selective Tracing: Impact Analysis[256]

As illustrated in ch. I.5.6, requirements changes occur in project practice. Thus, their consequences for the development process must be directly tracked in detail to avoid continuous drift between artifacts. For this, so-called *impact analyses* (*IA*) as described in ch. II.10.3 are the intended means for addressing these problems.

R2A offers the possibility to perform *IAs*, where *impacts* of requirements change on design can be easily made understandable for project members as well as for project outsiders (e.g., the customers) via iconographic highlighting.

Fig 22-1 shows two examples of how the *impact* results can look like during *IAs*, highlighting the *ANH* tree in R2A. The left tree shows a very local *impact* (red cross at 'Light_hdl' *AN*). Oppositional to this, the right situation shows *direct impacts* (red crosses) on the complete 'SW Design' as well as to the modules 'Light_hdl' and 'RTE'. Here, also *inherited impacts* (arrows with grey shade pointing at the bottom at 'ADC_drv', 'HighPrio_Task', and 'PWM_drv') and *indirect*

[256] Parts of this chapter have been published in [TKT+08].

impacts via decisions (yellow crosses at 'CIL_hdl', 'Light_Task', and 'SHR_hdl') are visible.

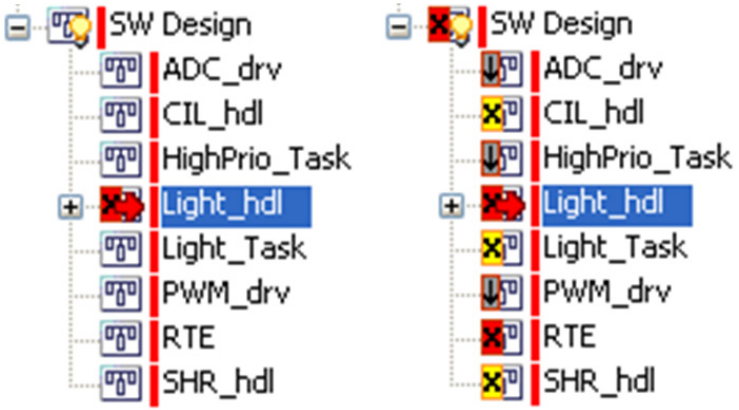

Figure 22-1 Two examples for visualizing *impact* on the *abstraction nodes hierarchy*

These opposed examples show the indisputable advantages of clear iconographic highlighting. Even though, engineering theory concentrates on reproducible results, the author is convinced that the developers' intuition (see also [LL07; ch.2]) is more often a factor of success than usually admitted. The graphical aspect of *R2A's IA* approach supports the intuition of the developers. This means even if no complex and detailed *IA*s are performed, the ease of just identifying a few items will also improve the working quality. A second major improvement of graphical *IAs* is that *impacts* of changes demanded by certain stakeholders can be better communicated to these stakeholders, as they can also more intuitively grasp the effects of the demanded change. Ebert emphasizes that "lots of changes are proposed because the corresponding interest groups think that the change is done by only changing a few lines of code or a parameter" [Eb05; p.188 (*)].

Via R2A's graphical highlighting of *impacts* such misunderstanding can be easily cleared and thus unnecessary change efforts, where change costs do not outweigh the change gains, are avoided.

However, development is not that easy that all effects of a change can be directly discovered. Often, changes can trigger a *dominoes effect* [VSH01; p.83] or *ripple effects* (cf. ch. II.10.3). To discover these effects earlier, project members must be able to perform more complex *IA*s because simply following the link chain only helps to find the *primary change* but neglects to identify the *second-*

ary change often leading to the *dominoes effect*. Thus, besides the simple graphical representation, the following characteristics allow significantly more precise *IA*s:

- Often several *requirements* in combination are affected by a meaningful change. R2A allows to starting an initial *starting impact set* (*SIS*) with several *RIs* (*Requirements*, *DCs* or *BRCs*).
- The affected *RIs* often involve formerly taken decisions and consequences (as *DCs* or *BRCs*) that must be reassessed. Starting from the initial *SIS*, R2A automatically calculates direct and inherited *impacts* on *AN*s derived from the *RIS* (ch. III.18.2.2). Additionally in a next step, indirect *impacts* through modeled decisions and their consequences (*DCs* and *BRCs*) are calculated with their *impacts* on *AN*s.
- The *inherited* and *indirect impacts* are automatically calculated by R2A from the formerly gathered *traceability* information. In order to allow users to differentiate between *direct impacts* and calculated impacts, the different impact types have different iconifications.

After R2A has first calculated the impacts, R2A offers dedicated support to perform a more detailed assessment of the *IA* results:

- Automatically *calculated impact* can lead to overestimated *impacts*. For these cases, the user can again determine for all *calculated impacts*, whether they are actually real *impacts* or rather overestimated *impacts*.
- To each element in the *IS* notes can be attached, by which the user can tell the cause why an item is in the *IS*, or what has to be performed in order to implement the change *impact* on the item.
- Performed *IA*s can be saved and shared with other users. This allows already performing rough *IA*s during meetings with the customer (ideally even at the site of the customer), early sparking concrete discussions with the customer if the customer expresses a change need. In combination with the possibility mentioned above to document notes on items in the *IS*, concretely identified steps to be performed on the change or other important information can already be documented and saved. This helps to capture early *rationale* on changes to perform. In the aftermath of such a meeting the developers then can refine the captured information. Estimations on costs and duration are one of the important information possible to be added are, thus extending the sheer *IA* to a detailed effort estimation.
- Once *impacts* are identified, a decision must be taken whether a proposed change is really performed on the project (e.g., by a *change control board* (*CCB*), [PR09; p.144f], [VSH01; p.184f, p.216]). As basis of such a decision the saved detailed *IA* results can be loaded and viewed in R2A again.

- Once a change has been approved, the gathered *IA* information about the change can again be loaded in R2A, providing now a detailed road-map for the designer to perform the changes.

These described actions and information can be steered via R2A's *impact analysis dialog* shown at the left side in fig. 22-2.

Fig. 22-2 shows the complete set of information displayed in R2A during an *IA*. At the left side, the *impact analysis dialog* is shown, whereas the right side shows an excerpt of R2A's main window with the *ANH* at the top and the "Requirements" tab at the bottom.

Impact highlighting on the *ANH* has already been discussed in the context of fig. 22-1. The "Requirements" tab shows the *RIs* of the selected *AN* (here 'SW Design'), where *RIs* being in the *impact* set are correspondingly highlighted to provide the user with information about the concrete *impact* on the *AN*.

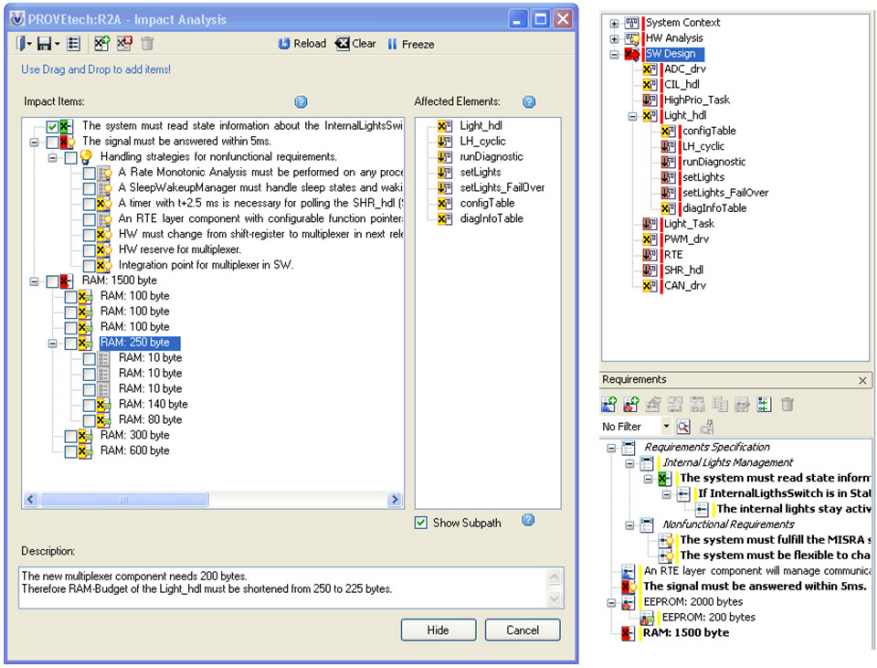

Figure 22-2 *Impact analysis dialog* and R2A's main window with an *impact set* taking *decisions* into account

The *impact analysis dialog* is divided into the left part showing the *impact* situation for *RIs* in connection with *impact* derived through decisions. The figure shows a situation of a planned change, affecting in the first instance requirement ReqSpec_2 (“The system must read ...”), requirement “The signal must be answered within 5ms” (taken from the documented decision concerning the *architectural influence factors assessment* shown in fig. 20-7 (see ch. III.20.4)) and *BRC* “RAM:1500 byte” (taken from the resource estimation described in ch. III.21 (see fig. 21-4 in ch. III.21.2.4)). All made decisions and all *DCs* or *BRCs* derived from the decisions are taken into account and shown beneath the elements identified in *SIS*. The right side shows the direct consequences on design (*ANs*) of an item selected in the left side (the complete *impact* on the *ANs* is shown in the *ANH*). Via the textual component at the bottom, notes can be edited and viewed describing additional information on the need for change of an item selected at the left side. Above, the author also mentioned that the dialog can be used as a detailed road-map to perform the changes. This is indicated in the figure by item ReqSpec_2, being checked and being highlighted via a green cross. Via this checking mechanism, changes already performed can be checked. In this way, the dialog turns into a checklist for the change to be performed showing the current status the designer is in during change implementation.

IA support is helpful to assess potential influences of changes and the captured *rationale*; during assessment it can give important guidance to how these changes must be performed. However, most probably not all *requiremental change* will run through a cycle of detailed *IA* and *CCB*. Often 'minor' changes influx into a *requirements specification* from all kinds of sources, though. For these cases the *change management mechanisms* described in ch. III.22.2 help to keep changes transparent in order to maintain changes to consistently propagate to all relevant parts of the *design models*.

III.22.1.2 Interactive Tracing: The *Model Browser*

Interactive tracing means to allow an interactive browsing mechanism to navigate backward and forward in the model.

Fig. 22-3 shows the *model browser* integrated in R2A for fulfilling *interactive tracing* needs. The *model browser* can be opened for any item present in R2A. Fig. 22-3 shows the *model browser* opened for the *NFR* “ReqSpec_14: The system must be flexible to change.”. In the left part of the *model browser*, direct information on the item (e.g., the text) and several *meta-data* (e.g., author and date of the last change, version, baseline, and internal item id) are shown. At the right side, all traceable relationships to other items are shown.

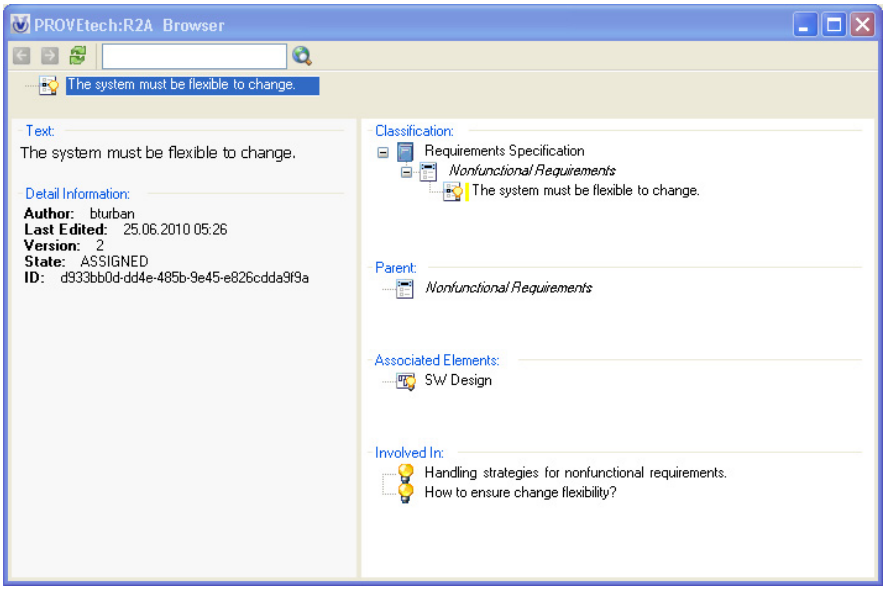

Figure 22-3 The *model browser* in R2A

There, the user can double-click on any item. Then, the *model browser* changes to this item, thus allowing navigating through the complete model present in R2A. The user can also open several *model browsers* in parallel, allowing keeping information on some items currently important to the user open; meanwhile he still can navigate further through model.

III.22.1.3 Non-Guided Tracing: Additional Features for Fast Look-Up

Non-guided tracing shall allow the user to arbitrarily step from entity to entity analyzing contents as demanded. This shall ensure convenient tracing when little information on what or how to trace is available.

Besides *IA* features and the *model browser* described in the chapters above, being also able to fulfill *non-guided tracing* needs, the following features provide possibilities for fast looking up some information:

- When the '*quick view*' option is activated, a slim version of the *model browser* automatically appears when the user works with R2A showing the current-

ly selected item in R2A. When the user changes to the *design tool*, the *quick view* automatically disappears. In this way, the user can on one side easily gather important information on an item. On the other side, the *quick view* can be arranged in a way overlaying the *design tool* but not overlaying any other information in R2A, when the user works with R2A. But when the user works with the *design tool*, again no disturbing window of R2A hinders the designer in working with the *design tool*.

- On any *RI*, the '*locate origin'* action can be performed opening the *requirement source document* of the *RI* and selecting the *RI* in the *requirement source document*.
- In the same way as '*locate origin'* opens the corresponding *requirement source document* in R2A, the '*locate in REM-tool*' action can be performed on any *RI* originating (being synchronized) from an *REM-tool* such as IBM Rational DOORS, opening the corresponding document containing the *RI* in the *REM-tool* and selecting the *RI*.
- Vice versa, R2A also integrates a button into the *REM-tool* environment allowing a '*locate in R2A*' action, where a requirement selected in the *REM-tool* is then again shown in R2A.
- As described later in ch. III.23.1, parts of an R2A-model can be again exported into a *REM-tool* to support supplier management. Similarly to the two points above, R2A also allows navigating into such a generated document in the *REM-tool* and back.
- 'The *'Show related decisions'* action can be performed on any item in R2A. When performed, a window opens showing all decisions the item is involved with (either as conflicting or resulting item) in the style shown in fig. 20-6 (see ch. III.20.4).

Through the different *locate* actions, *bidirectional traceability* (see ch. I.5.7.1) is ensured, where *RIs* can be traced in the backward and forward direction.

III.22.2 Consistency Maintenance of Requirements, *Traceability* and Design[257]

In ch. II.10.4.3, establishing *traceability* has been identified as an important aspect to consider because it means significant effort to be spent. This is only one facet of the problem. A second equal problematic facet is that later changes must be efficiently and consequently inferred throughout the whole development effort

[257] This chapter bases on parts of [TKT+08].

in order to ensure consistency throughout the whole development project (see ch. II.10.4.3.4). Otherwise, the best *traceability* establishment processes will be in vain if the traces significantly degrade in short time. On the other side, a certain degradation of traces may be inevitable even under best support for trace maintenance.

To ensure *traceability* information is maintained best possible, obstacles for *traceability* maintenance must be as low as possible. In R2A, maintenance of *traceability* information is easy and intuitive because of the overall *drag-and-drop* support as well as operations as *dribble-up, dribble-down* and *copy*, and the concept to present only the information relevant in the given design situational context.

A main concern addressed in maintenance of *traceability* is ensuring consistency. The following now shows how R2A supports that requirement related changes are consistently inferred to design.

If a proposed *requiremental* change is decided to be performed[258], it must be possible to propagate the changes in a controlled way, ensuring a consistent implementation of the change in all artifacts. For each *RI*, R2A is able to visualize its status by using a colored status bar at the left side of each *RI* (see fig. 22-4), where each *RI* runs through the life-cycle sketched in fig. 22-4.

[258] The *CCB* can also decide not to perform a change. (e.g., if the effort detected via an *IA* is higher than the change's value gain).

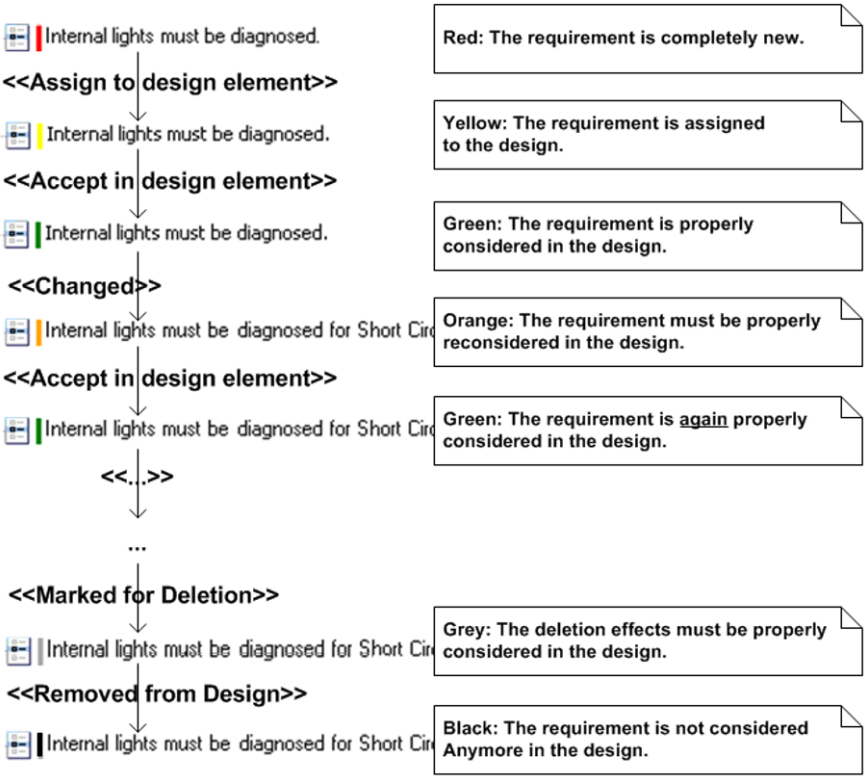

Figure 22-4 Life-cycle of a *requiremental item* and its color coding in R2A

Each *RI* not yet considered in the R2A design (status 'red') must be assigned to the design (change to status 'yellow'). Yellow means that an *RI* is considered, but it did not yet reach its final state of realization in design (see *RDP* heuristic in ch. III.18.2.4). If a designer decides that an *RI* has reached its final state of realization, the designer can perform an *accept* operation on for the *RI* at the corresponding *AN*, indicating that the designer considers the *RI* has reached the adequate location in the design. When the *RI* is *accepted* at all *AN*s[259], the *RI* auto-

[259] An *RI* can also be assigned to several *AN*s. Of course, it should be avoided that the realization of an *RI* is performed by several *ANs;* however in certain cases this will happen.

matically changes to status 'green' meaning that the *RI* has generally reached the adequate consideration at all parts of design it must be considered. Later changes on the *RI* (e.g., after a new synchronization of the *requirements specification*; see ch. III.18.1) may require a reassessment of the RI's current realization in design. Therefore, the status of the *RI* changes to 'orange' until the designers have performed the necessary changes on the design to again adequately consider the *RI*. This can also involve that the *RI* may be relocated to another part of the design (assigned to another *AN*). Once the *RI* is again *accepted* by the designers, at all assigned locations, it is again promoted to status 'green'. This handling recurs every time the *RI* is changed.

If an *RI* becomes obsolete during project progress, the *RI* can be *marked for deletion* by the designers (change to status 'gray'). As soon as the designers have considered the marked *RI* in design, it can be finally deleted (change to status 'black'). In this way, it is can be ensured that design settings having become obsolete can be removed, thus avoiding clutter and architectural erosion.

III.23 Aspects of Embedding R2A in a Process Environment

Getting the formula right entails knowledge, patience, foresight, and communication.
[BT04; p.99]

A tool alone is not a solution for a problem. Instead, a tool must also be embedded into a process landscape (see beginning of ch. II.10.4.4). After the chapters above described the tool R2A and how its integration supports the transition processes from requirements to design, this chapter widens the scope of considered processes in the sense that the requirement and design processes may be again embedded in a higher-level process environment, where tight integration is essential. These aspects can be that parts of a designed system are supplied by a supplier. In this case, design must be tightly integrated with *supplier management* to propagate important information to the supplier. Ch. III.23.1 describes how the information gathered during a design process with R2A can be directly used to generate a *requirements specification* for suppliers dedicated to deliver parts of the designed system (resp. SW). This helps to avoid redundancies and thus significantly improves supplier management.

Another, issue may be that several requirement and design processes may occur on different levels of abstraction, where the results on one abstraction level induce requirements and design on another level of abstraction (see ch. I.7.3.2).

Ch. III.23.2 discusses how this can be achieved best with R2A. Again, the *requirements specification* generation feature described in ch. III.23.1 also proves helpful in this case.

III.23.1 Avoiding Redundancies in *Supplier Management*

"In the development of complex embedded systems, often several companies work together on the development. At such an interconnected development, often partnerships are built, where mostly one supplier is engaged as the system supplier, having – besides other tasks – the responsibility to coordinate the other suppliers. Therefore, selection and coordination of suppliers is of special importance in embedded development. Often, even a hierarchy of client-supplier-relationships emerges, meaning that a supplier (*second tier*) acquires further sub components of the system from his own suppliers (so called *third tier*) and coordinates the collaboration. Additionally, the customer often prescribes the supplier certain *third tier-suppliers*" [HDH+06; p.65 (*)].

If a partial component of a system or software must be supplied by a supplier, a reliable and efficient *supplier management* must be installed (see ACQ.1 and ACQ.4 process in SPICE [HDH+06]).

For this, at minimum a *supplier requirements specification* (*SuppRS*) must be continuously administered. Such a partial component, however, must be included in the design of the higher-level (more abstract) component the partial component shall be integrated into[260]. In the further this design is called the *customer's system design* (*CusSysDes*).

As a main problem, high content redundancies arise between the information created during design and the writing of the *SuppRS* leading to high extra effort spent on creation, keeping the *traceability* and applying changes. Especially applying changes can be seen as a critical issue because redundancies are often accompanied by the danger that the changes are not propagated to all redundancies, leading to growing inconsistencies between the redundancies.

R2A tackles this problem by using the information about the component created in the *CusSysDes* directly to automatically create the *SuppRS*. This means that the partial component is included as *AN* in the design of the higher-level component. The requirements for the component emerge from:

[260] For example, a complete system, a sub system, a complete SW system, a partial SW sub system

- The previously found requirements for the higher-level component that are assigned to the partial component as requirements (r*equirements* in R2A-terminology).
- The constraints for the partial component, resulting from the decision processes during design of the higher-level component (in R2A-terminology *DECs*, *DCs*, *BRCs*).
- Inherited *RIs* from parent *AN*s (see ch. III.18.2.2) as they also may be important for the component.

R2A offers the possibility to export all this information concerning an *AN* to an *REM-tool* as a new *requirements specification* artifact for the supplier. Later changes in the R2A can be synchronized into the artifact. However, a *SuppRS* usually should not just include the *requiremental* information. Instead, the context of the component to supply (embedded in the higher-level system) is important. Thus, besides this *requiremental* information mentioned above, R2A can also export the following information:

- Modeled diagrams showing how the component collaborates with the other parts of the system.
- The textual description of the component performed in R2A.

Of course, not all information created during *CusSysDes*, concerning the component of a supplier need be propagated to the supplier. In fact, often the customer must decide which information is necessary to propagate and which information must not be propagated in order to protect the customer's know how. Thus, R2A's *SuppRS* generation mechanism contains a wizard, in which it is possible for each item to set whether to propagate to the *SuppRS* or not. After the *SuppRS* is once created, the synchronization mechanism also detects later *edit changes* (i.e., changes through later editing or formatting) in the *SuppRS*. When afterwards the next synchronization with the *SuppRS* occurs, the changes in R2A and the *edit changes* performed in the *SuppRS* are equally considered. Besides allowing *edit changes* of the *SuppRS*, *R2A* mainly allows covering two other points important for the *SuppRS*:

1. The *SuppRS* as a requirement artifact read by humans also must obey the rules for a human readable document. Thus, the document must provide a continuous reading flow. In most cases, this means the raw version of the synchronized *SuppRS* must be reedited. For these reasons, also new items can be added to the *SuppRS* manually. These items are then handled outside of the R2A approach and the development team must use other mechanisms to keep these elements up to date. Besides adding new elements not managed by R2A, a *SuppRS* requirement artifact can also be restructured at will in order to improve reading by humans. This works properly when the order or hierarchy of the requirements is changed; but it involves some problems if

also the text of a requirement must be changed. In principle, changing the text of a synchronized element (e.g., to improve readability) is possible but this makes the following synchronizations more difficult to manage because then both sides to be synchronized (the R2A side and the *REM-tool* side) may have changed. In these cases, it is indicated that the designer must manually merge the texts. Thus, the author rather recommends to perform the textual change already within R2A and then again to synchronize the *SuppRS*.

2. Decisions not to propagate certain information elements to the customer may just occur during the editing phase of the *SuppRS*. In these situations, it would be very long-winded if the synchronization mechanism had to be performed again in order to select information not to propagate in the wizard. Instead, it is easier to just delete the elements in the *SuppRS*. Then the synchronization mechanism detects that these elements are deleted and will not again synchronize these elements.

Such an emerging *SuppRS* can then be used as *user requirements specification*[261] for the supplier. As the information is directly generated out of the previous design processes by R2A, the *single-source-principle* ensures that redundancies are avoided.

III.23.2 *Traceability* over Several Artifact Models without Redundancies

As discussed in ch. I.7.2.4, the topic *traceability* between requirements and design involves different artifacts at different levels of abstraction in *process models* such as SPICE. After having all pieces together now, this chapter discusses this topic from the process chain and artifacts viewpoint.

Fig. 23-1 describes the *process* and *artifact model*, when *system design*, *SW design* and perhaps even *HW design* are performed in one *design model*. Only a common *requirements specification* with the real requirements from the customer (corresponds to the *SYS_RS* in SPICE) are imported from a *REM-tool* and are related to the corresponding *ANs* in the *system design*, *SW design* and *HW design*, being responsible for fulfilling the requirements. During the design processes new 'requirements' arise in the form of *DCs* and *BRCs* from design decisions made. These 'requirements' enrich the original requirements. In this way, the *SW_RS*, *HW_RS* and *module requirements specifications* are all *RI*s assigned to

[261] In German: 'Lastenheft' (see ch. I.7.2.2.1)

the *ANs,* representing the *SW design*, *HW design* and *module designs* and are only metaphorically present in development.

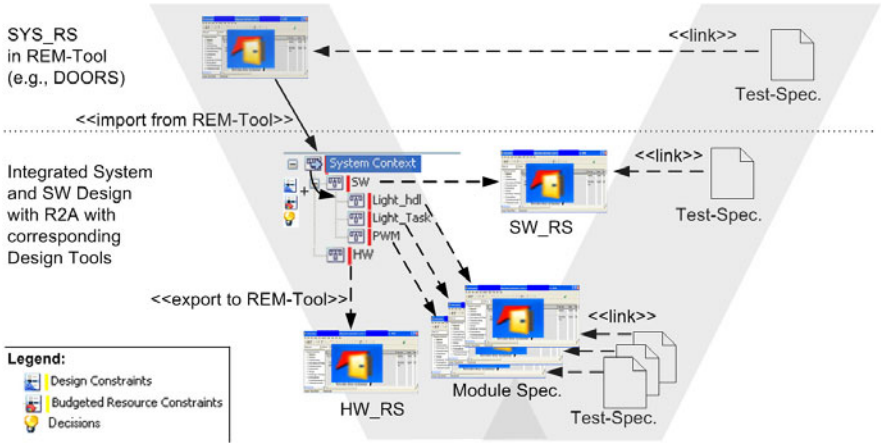

Figure 23-1 Process chain of an integrated *design model* for *system*, *HW* and *SW design*

However, the SPICE standard also demands that testing procedures must be performed on the corresponding *requirements specifications*. This can be achieved through R2A's feature for creating a *requirements specification* from a partial *design model* (originally intended for *supplier requirements specifications*; see ch. III.23.1). Now, these created *requirements specifications* can be used to create and link test specifications to the corresponding *requirements specifications*.

The author recommends using this *process model* because it provides optimal communication for designers, reduces redundancies to a minimum, and provides best support of R2A's *consistency management* mechanisms. As described in ch. I.7.2.4, Hörmann et al. emphasize that in practice the transition between these processes mentioned are anyway mostly fluent and are rather of iterative and recursive nature [HDH+06; p.103]. Correspondingly, this model also is closer to practice than the original SPICE *process model* is.

However, as mentioned in ch. III.19, a *process model* deviating from the original SPICE model is allowed in principle but requires higher efforts for organizations to prove that the *process model* corresponds to the original ideas of the SPICE *process model*. It may even be possible that the *process model* has

lower acceptance by SPICE assessors (the power of assessors assigning negative assessment results should not be underestimated). These factors may push organizations to the decision to rather exactly follow the SPICE *process model* to avoid such problems.

Fig. 23-2 shows how such a process chain may look like when R2A is employed in an organization using the original SPICE *process model*. At start, the requirements of the customer are collected in the *SYS_ RS* in the same manner as above. Via R2A in connection with a *design tool* adequate for *system design*, the *system design* is created. During *system design* as well as in the other design phases described a few lines later, new *DCs*, *BRCs* and *Decs* emerge (emphasized in fig. 23-2 by a '+'). In the *system design* artifact, a placeholder "SW" is created, collecting all relevant requirements and other items resulting from the design (*DCs, Decs* and *BRCs*) having influence on the SW. This placeholder can then be used to generate the requirements specification for the SW forming the basis for the *SW design*, again performed in R2A in connection with a *design tool* adequate for *SW design*. If needed, the same procedure can be applied to modules in the *SW design* if a dedicated *module specification* is needed (in most cases this may be especially interesting, when the realization of modules is delegated to a supplier). Through these controlled import and export actions via R2A, controlled copies emerge, whose redundancies are in most cases maintained under automation support.

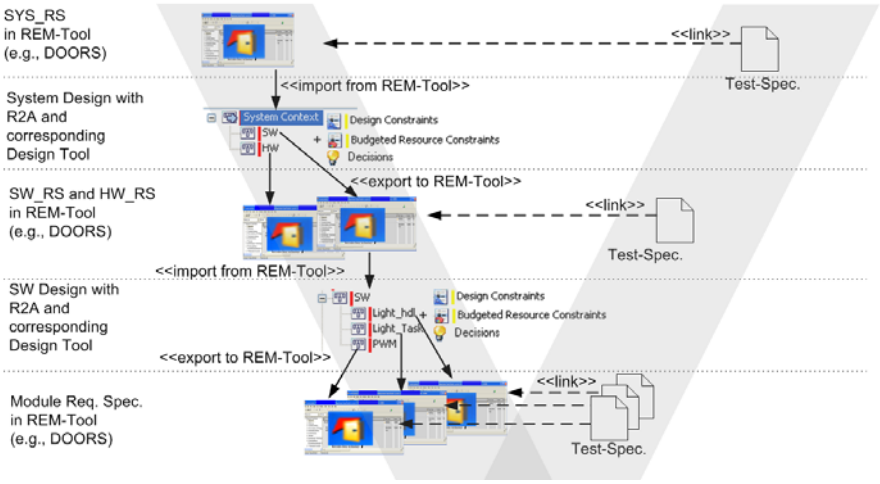

Figure 23-2 Process chain of multi-layered requirements and design artifacts

In this *process model* implementation, R2A also provides advantages of minimized redundancies because *SW_RS*, *HW_RS* and the *module requirements specifications* are generated from the *design models* made earlier with included *DCs, Decs* and *BRCs.* On the other side, *IAs* and *consistency management* become significantly more difficult because the tool barriers between artifacts in *REM-tools* and R2A must be crossed permanently. This leads to friction losses.

III.23.3 Decoupled Development of Requirement and Design Artifacts

The process chain introduced in the previous chapter still leaves one central point uncovered: Often, different artifacts are developed with a certain time-lag in parallel. Thus, after the *SYS_RS*, the *system design* is developed with a time-delay, and after the *system design* again the requirements specification and design of the SW are developed with a certain time delay. During this process, requirement changes already occur in the *SYS_RS*.

In simple link concepts, the link chain now can be paced off by an *IA*, but controlling a consistent maintenance through all artifacts proofs difficult[262].

R2A addresses this problem by an interplay of synchronization, consistency propagation (ch. III.22), and export (ch. III.23.1) mechanisms.

Fig. 23-3 shows the effects of these mechanisms in cooperation, in which the R2A process artifact chain of fig. 23-2[263] is extended by a temporal dimension, showing change deltas (horizontal dimension). From top to bottom, different requirement and design artifacts are shown at different levels of abstraction (*system design*, *HW design* and *SW design*). R2A is able to perform the synchronization mechanism on different version baselines of requirement artifacts. Thus, it is possible to synchronize the requirements according to an existing version baseline of the requirement artifact.

---

[262] Current *REM-tools* such as IBM Rational DOORS provide mechanisms to mark such links. In IBM Rational DOORS, e.g., these links are marked as 'suspect links'. However, after a baseline is made in a certain artifact all suspect links are cleared, making it unfeasible to perform baselines in a time-delayed development for a certain artifact. Moreover, the problem increases when tool gaps as the problem of an essential tool gap between *REM-* and *design tools* as exposed here are involved.

[263] The statements are analogously valid for fig. 23-1 in ch. III.23.2.

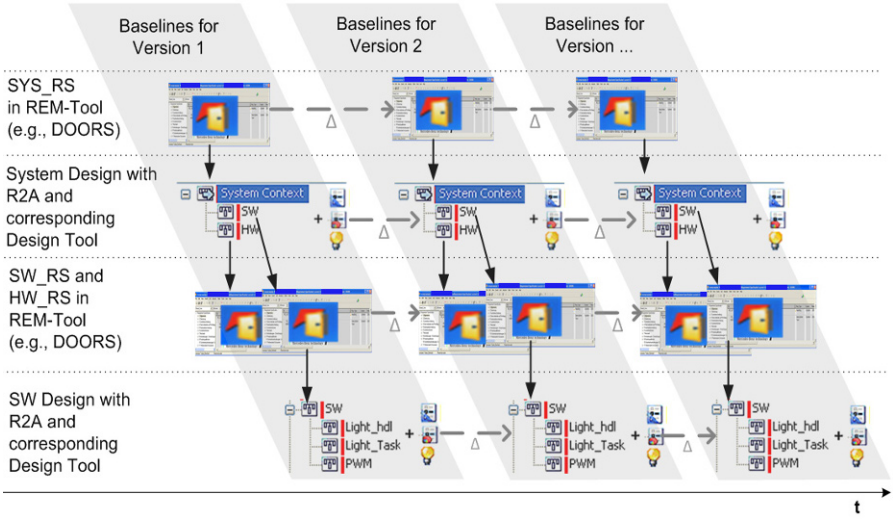

Figure 23-3 Consistent integration of changes (Δ) beyond version barriers

Through the consistency mechanism, this requirement artifact version can be propagated through the designs (with new *Decs*, *DCs* and *BRCs*) and the export mechanism then propagates this baseline version state to the requirement artifacts at lower levels of abstraction. In the meantime, the requirement changes (Δ) for the next version can already be performed, being again propagated downward to the artifacts at lower levels of abstraction within the following version baseline.

Subsuming, it is to say that R2A conducts requirement changes into controlled, consistent version pathways (gray pathways in fig. 23-3), but at the same time it allows a decoupled, further development of requirement changes for subsequent versions in parallel.

III.24 Overall Architecture of R2A

Designers have occasionally been urged to seek for 'ideal solutions of design problems' or words to the same effect. There can be no ideal solutions.... Design is not like that. There are, however, occasions when it is possible to determine temporarily what is the best practicable balance between opposing requirements....
The fact that compromise is inevitable in so many kinds of design has led theorists to classify design as a 'Problem-solving activity', as though it were nothing more than that. In is a partial and inadequate view.
Most design problems are essentially similar no matter what the subject of design is....
[Py78; p.74f]

After the chapters before have described the features of R2A with their innovative potential, this chapter describes the technical background of the R2A solution. At first, the general *architecture* of R2A is described. The core of the R2A tool is the *conceptual meta-model* described in the second sub chapter. Afterward, other additional interfaces are described.

III.24.1 General Architecture

Fig. 24-1 describes the high-level *architecture* with the most important packages and their interdependencies. The overall structure is divided into three parts:

- The "*General Reusable Libraries"* part subsumes libraries with general supportive tool (resp. utility) libraries that can also be used in other development projects, thus generating significant alleviations for new development projects. In the *Infrastructure* package, general solutions for *cross-cutting concerns* as error logging, threading support, or integration of unit testing, etc. are developed. As it provides very basic support, the *Infrastructure* library is used by all other packages in R2A. Basing on *Infrastructure*, the *GuiFramework* package is the equivalent of *Infrastructure* but for GUI[264] support. The *GuiFramework* provides better support for user messaging (a framework, where user vocabulary and messages to the user can be defined in a general way), encapsulates important GUI-controls to make them exchangeable and more stable. Further, the framework provides a general implementation of the *model-view-controller pattern* allowing easily creating new user controls with support of the *model-view-controller pattern*. Several other smaller reusable libraries addressing more special *cross-cutting aspects* exist, not explicitly mentioned here.

[264] Graphical User Interface

- The "*Product Line Core*" is the actual core of R2A. Its *architecture* follows the *three layer architecture pattern* [BMR+00; p.31ff]:
  - The *Gui* package contains all program elements directly related to the *graphical user interface*.
  - The *ProgramCore* package contains the data model and its operations of the R2A application. In *ProgramCore*, the *MetaModel* package contains the data model, whereas the *ModelController* package contains and controls operations on the data model. R2A's data model classes have detailed knowledge about their own structure. In this way this data model is more a *meta-model* about the entities represented in the R2A-model. This *meta-model* is described in the following ch. III.24.2.
  - The *Opf* package is an *object persistence framework* (*OPF*) responsible for mapping the R2A data from the *meta-model* to its representation in the database. The *OPF* also can automatically handle the *cross-cutting concerns* of *versioning* and *baselining* realizing the features described in ch. III.17.5. As the *OPF* realizes any data changes, it also contains a collaboration framework allowing other R2A instances of other developers connected to a project to be notified about data changes. These notifications then trigger the collaboration framework in the other R2A instances to update the changed model parts, thus allowing direct synchronous collaborative work between the designers.
- The "*Variation Points*" part contains the packages *RemInterface* and *MdlInterface*. *RemInterface* is the *variation point* to connect different *REM-tools*, whereas *MdlInterface* is the equivalent *variation point* to connect to different modeling tools. As both packages have equivalent responsibilities but for different tool types, the internal structure of both packages is equivalent. Both contain a general part and a tool specific implementation part. The general part shall encapsulate the tool specific part from access of the *ProgramCore* package. The general part contains an abstract interface definition each specific tool implementation must implement, a factory class that uses the *abstract factory pattern* to create a specific tool object with the implementation of the abstract interface, and objects representing items present in a connected tool. These objects (*TMdlObject* in *MdlInterface*, *TReDocumentItem* and *TReDocument* in *RemInterface*) are used to connect information of a connected tool with the data model (see ch. III.24.2).

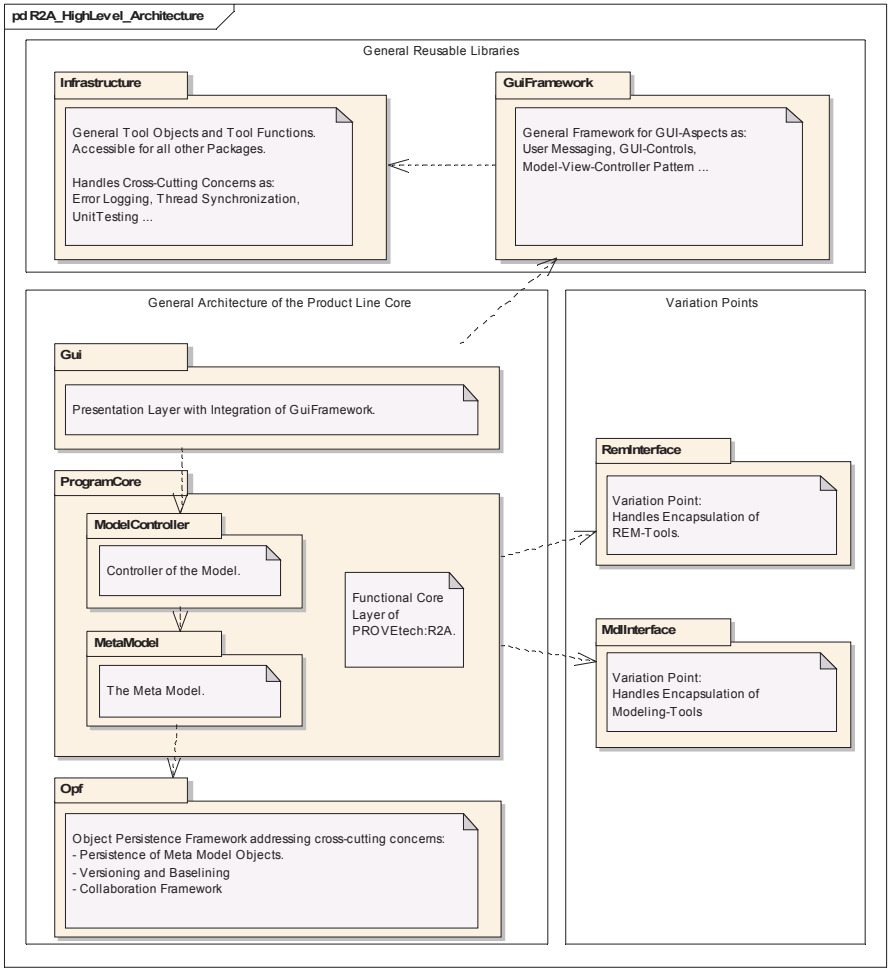

Figure 24-1 High-level *architecture* of R2A

III.24.2 The *Meta-Model*

The concepts mentioned above are embedded in R2A in a *meta-model*. The *meta-model* can be seen as the *traceability reference model* or *conceptual trace model* (ch. II.10.4.3) of the approach.

Although a certain overlapping with concepts of the *meta-model* of UML (with SysML; in the further just referred to as UML) exists, R2A's *meta-model* is not basing on an implementation of UML, because:

- The UML *meta-model* did not yet exist as a standard, when research on the *meta-model* of R2A began.
- The UML *meta-model* is substantially more complex since it is very generic and it is designed to cover all aspects and concepts of design, whereas R2A only uses some specific concepts important for design structuring, *traceability* and design documentation.
- R2A aims to be open to integrate other design modeling approaches. Thus R2A must avoid a too strong concentration on one modeling approach.
- The usage of the UML *meta-model* would demand to be conforming to UML. R2A involves research on new concepts and ways to establish *traceability* in an easy to use fashion. Strong orientation on a standard could predetermine the researcher's thinking in an unfavorable way, preventing to find a good solution. Or, probably new concepts are necessary that cannot be adequately mapped to the UML *meta-model*. Such cases of mismatch can be seen in the *DC* concept[265] or the decision model concepts.

Nevertheless, the UML and SysML concepts have been analyzed and inspired certain concepts of R2A and its *meta-model*.

Fig. 24-2 shows the R2A *meta-model* with the most important[266] classes, its properties and relationships. As a convention of the R2A-project, all type names start with a capital T as abbreviation for the word 'type'. Through this notation, inspired by the hungarian notation, types created within the R2A-project can be

---

[265] The UML has a constraint concept but with very different semantic to what is called a *design constraint* in R2A. However a certain connection between both concepts exists in the form that the UML constraint semantic can be seen as a special case of the *design constraint* semantic. As the UML constraint semantic bases on a *formal* language concept (called *Object Constraint Language* (*OCL*)), it is designed to describe very specific design issues in design diagrams in an annotation format. In contrast, a *design constraint* aims to describe all kinds of constraining effects of a design in natural language, thus providing significantly higher flexibility for description.

[266] The reader should note that the *meta-model* shown here is idealized to be understandable for the reader. In reality, the *meta-model* contains a few more classes, and the classes have significantly more properties and relationships. E.g., the access to TMdlObject objects (associations (5.) and (7.)) is in reality controlled through a proxy object *TToolsObjectProxy* to improve encapsulation of the *MdlInterface variation point*. This is important for the real tool implementation but is an implementation detail not necessary for understanding the fundamental concepts of the *meta-model* to be introduced in connection to this thesis.

easily differentiated from original types provided by the Microsoft C# .Net environment.

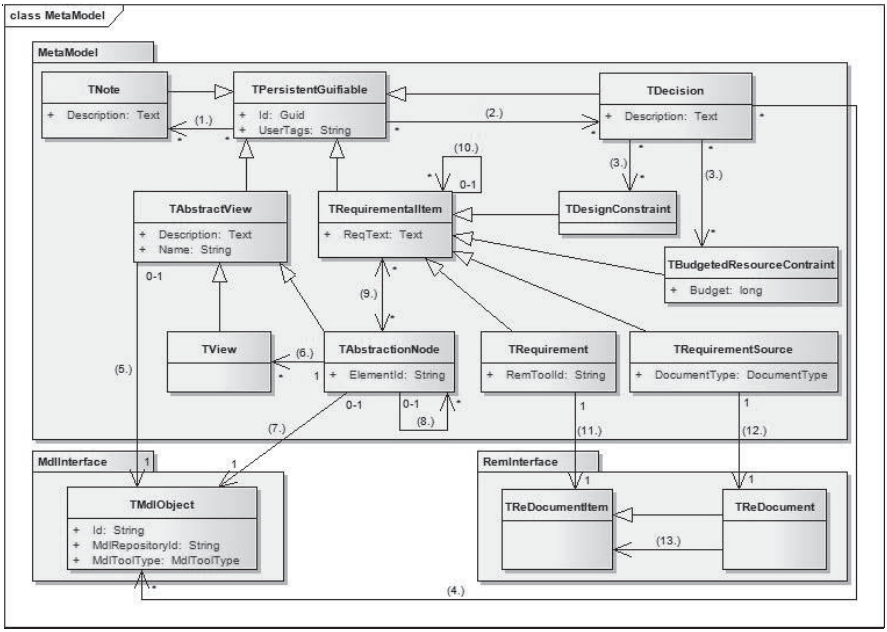

Figure 24-2 The *meta-model* of R2A

As mentioned in ch. III.16.1, R2A consists of a *core* and the two *variation points* for integrating *REM*- and *modeling tools*. In fig. 24-2, the *REM-tool variation point* is described by the *RemInterface* package and the modeling tool *variation point* by the *MdlInterface* package. The *meta-model* is located in the *core*, but information located in the connected *REM*- or *modeling tools* must be referenced through proxy objects in the *variation points*, abstracting from a specific implementation in a specific tool. In the case of the *RemInterface*, the class *TReDocument* represents a document in an *REM-tool* and *TReDocumentItem* represents an item (e.g., a requirement) within an *REM-tool's* document. *TMdlObject*, on the other side, represents any item available in a modeling tool.

Concerning the *core's meta-model*, any item inherits from *TPersistentGuifiable*. In *TPersistentGuifiable*, central characteristics necessary for any item to be part of the *meta-model* are realized.

Their characteristics are:

- *Persistence*: The item can be stored in R2A's data base through being a persistent item managed by a persistence framework (*OPF*).
- *History and baselining*: To fulfill demands of *evolutionary traceability*, the change versions' history must be recorded and it must be possible to include a version state into a baseline. Both are also accomplished by being a persistent item managed by the persistence framework.
- *Representablity in R2A's GUI*: *TPersistentGuifiable* implements all necessary characteristics for representation to be integrated into R2A's GUI concept.
- *Unique identifier* (cf. ch. III.17.4): Through the *Id* property, any item has one general unique identifier (GUID), through which the item can be referenced.
- *User tagging* (cf. ch. III.17.3): Through the *UserTags* property, any item can be tagged by users.
- *Notes* (cf. ch. III.17.2): Through association (1.), any item can be assigned to *TNote* objects representing notes. It is possible to assign several notes to an item as well as to assign several items to one note. As *TNote* is also part of the *meta-model* and inherits from *TPersistentGuifiable,* it is in principle possible to make notes of notes.
- *Being part of a conflict based decision* (cf. ch. III.20): Association (2.) represents the *conflicting* relationship in fig. 20-2 (see ch. III.20). Through this association, it is possible that any item of the *meta-model* can take part on a conflict, where a decision to solve the conflict can be modeled. This even includes notes or other decisions.

Design aspects are expressed through the concepts *TAbstractView, TAbstractionNode* and *TView. TAbstractView* represents general principles any view concepts in R2A have in common. The general principles are that a view has a name, can have a textual description and is expressed through a diagram in a modeling tool linked to through association (5.). *TAbstractionNode* represents the *AN* concept as described in ch. III.15. An *AN* consists of a *design element* in a modeling tool expressed through association (7.), a diagram in a modeling tool expressed through association (5.) and a description inherited by *TAbstractView*. The *ANH* concept is built up through association (8.). *TView* represents further related *views* that can be added to an *AN* (see description to fig. 15-4 (in ch. III.15)). An *AN* knows its related views through association (6.).

Requiremental aspects are expressed through the inheritance hierarchy starting from *TRequirementalItem*. This inheritance hierarchy resembles the *requiremental items taxonomy* introduced in fig. 21-1 (see ch. III.21.1), except for the fact that the inheritance hierarchy also contains *TRequirementSourceDocument*, representing requirement source documents described in. ch. III.18.1. This can be considered as a kind of artifice to create a thorough *requiremental decomposition hierarchy* in R2A. As described in ch. II.10.4.2.2 and ch. III.18.1, decomposition

of requirements is a common principle in *REM*. Association (10.) from *TRequirementalItem* to *TRequirementalItem* is a *parent-child-relationship* used to build up requirement decomposition hierarchies. This decomposition hierarchy relationship can be used in principle for any *RI*. In fact, the hierarchy is used by requirements to reproduce the decomposition hierarchy present in *requirements specifications* from *REM-tools*, and it is used for the decomposition discussed in the course of *BRCs* sub budgeting (ch. III.21). Association (10.) is also used for *requirement source documents* (*RSDs*) to refer to the root *RIs* being at the start of the decomposition structure of a *RSD* (in fig. 18-2 (see ch. III.18.1), e.g., the *TRequirementSourceDocument* "PH" refers to the *requirements* "MSG Wakeup","Internal Lights Control", "Nonfunctional Requirements" and "HW" through association (10.)). In this way, a *RSD* is a parent of the root *RIs* in the document. This view is not wrong because a *RSD* as a container of *RIs* is itself an *RI* in the sense that the *RSD* demands that all containing *RIs* must be fulfilled. Through the *Type* property, the *TRequirementSourceDocument* specifies whether it is a free-edit document or whether it origins from a *REM-tool*. In the latter case, association (12.) refers to the corresponding document in the *REM-tool*. In a similar way, association (11.) refers *TRequirements* originating from an *REM-tool* to the original item representation in the *REM-tool*.

As *requirements traceability* to *design elements* is the core scope of R2A, *RI* must be linked to the design. This is expressed by association (9.) representing the '*assigned to'* or resp. '*satisfy'* relationship between *ANs* and *RIs* described in ch. III.18.2.

The *decision model* described in fig. 20-2 (see III.20) is realized by the class *TDecision* and its associations. As mentioned above, association (2.) represents the conflicting entities relationship. Association (3.), however, refers to the resulting consequences derived as *TDesignConstraints* or *TBudgetedResourceConstraints*. Association (4.) realizes references to further documenting design diagrams in a modeling tool.

III.24.3 Further Interfaces

Additionally to the user interface, R2A has the following other interfaces:

- *REM-tool integration*: As described in ch. III.13, R2A provides a *variation point* to integrate *REM-tools* as source for requirements (ch. III.18.1) and as target to export requirements for *supplier management* (ch. III.23.1).
- *Modeling-tool integration*: R2A provides an integration interface for modeling tools as *variation point* described in ch. III.13.

- *Word interface*: For documentation of the design and design decisions, Microsoft Word is integrated into R2A. The Word documents are saved in the R2A database in *rich text format* (*RTF*) and are integrated in R2A's other information *meta model* through a *persistence framework* (e.g., the *meta model* items *TAbstractionNode*, or *TDecision* contain a *persistent* property "Description" referring to *RTF* documents editable with Word).
- *Standard report*: A standard report interface allows to generating a HTML-report of the generated model in R2A. The report includes diagrams modeled in the connected modeling-tools, thus enabling to generate extensive design documentation.
- *XML-export*: Ch. III.17.3 describes the XML-export feature allowing the complete model gathered in R2A to be exported for organizations to reuse the gathered information in other tools or to develop own special purpose tools working on the information.
- *Rule engine*: *Consistency management* is a decisive issue for ensuring quality of developed artifacts. Ch. III.22 has described the standard features for *consistency management* in R2A. However, often projects have individual characteristics influencing the consistency. To cover this, R2A provides a *rule engine*, where projects can specify individual rules for *consistency checking*. In this way, projects can ensure that the R2A model fulfills consistency criteria defined in the project. At the moment, the current *rule engine* concept implemented in R2A is only a prototypical implementation, showing a proof of concept. This point can be seen as a promising perspective for further research and improvement of the R2A concept. As an example of the possible uses of R2A's *rule engine* concept, it is possible for designers to define rules that any *design element* with a certain characteristic must obey. When the example of the decision modeled in fig. 20-6 (see ch. III.20.4) is considered, the *DC* "Handlers and Drivers shall provide callback mechanisms to their upper layers (*Dependency Inversion Principle*)." exists that must be obeyed by any handler or driver in the *SW design*. With the *rule engine*, a designer can define a rule that ensures that the *DC* is automatically assigned to any handler and driver *design element* currently present in R2A. The rule also ensures, that the *DC* is added to any *design element* with handler or driver characteristic, added later to the R2A model.
- *Special reports with the rule engine*: The reporting mechanisms can be combined with the *rule engine*. In this way, customized reports can be created in R2A for special reporting needs of a project. With the *rule engine*, scripts can be written to extract data from the data collected in an R2A-project specially prepared for the customized report. Through customized reports, e.g., it is possible to create reports about statistical data of a project to report it to man-

agement (e.g., to report how many *RIs* are not yet considered in design, partially considered in design and how many *RIs* have reached the final state in design).