Any system design and development activity should always begin with a phase where requirements and technical specifications are collected and captured. System requirements should clarify what the system is supposed to do (what is its behavior), what properties and constraints it should satisfy, and also environmental and regulatory conditions that apply to the system, its design or development. The product of this process is a model or a set of models, possibly included in one or more formal documents. Requirements may be the technical annex to a contract binding the system developer(s) and its users or purchasers and/or specifying what system (parts) needs to be provided by a supplier in an integrator-supplier relationship. In addition, requirements are typically part of the internal documentation of the company developing the embedded system. In all cases, they describe the (sub)system and provide a reference point shared between those who understand and collect the customer’s needs, those who implement systems and programs to satisfy those needs, those who test the results and those who write instruction manuals. In the product lifetime, requirements also provide a reference for maintenance and extensions.

Requirements engineering is a key problem area in the development of complex, software-intensive systems "The hardest single part of building a software system is deciding what to build. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later." [Brooks 87]

4.1 Introduction

Traditionally, requirements have been given informally with many pages of natural language descriptions, sometimes flanked with manually derived diagrams that represent the structure of the design. In the documents, a lengthy elaboration of what is expected of the design is accompanied by a set of properties the design has to satisfy and a set of requirements on the implementation, possibly including indications on the components to be used. For example, in the automotive domain, when a subsystem is specified by a car maker for a Tier 1 supplier, e.g., an engine controller, the characteristics of the subsystems may include the prescription of what microprocessor platform to use, e.g., the part number of an IC supplier.

We argue that this approach amounts to far-from-optimal designs since the choice of the implementation platform should be best decided by the Tier 1 supplier who can optimize with respect to his own business criteria while delivering a product that satisfies its function definition and the performance constraints. In addition, requirements and specifications often change during the lifetime of the design project and keeping track of these variations is a constant nightmare. Not unexpectedly, the informality of the description also leads to misunderstandings, delays and cost overrun.

The issues of specification capture and management is considered today as a crucial aspect of system design and one that needs much research and tool development. There is evidence that a ratio of 15 to 1 in design quality and efficiency can be credited to a good specification entry and analysis process. The DoD Software Technology Plan [DoD91] states that "early defect fixes are typically two orders of magnitude cheaper than late defect fixes, and the early requirements and design defects typically leave more serious operational consequences."

In this chapter, we briefly outline the main issues and approaches to the problem and then present some general principles that typically apply to the development of embedded systems.
4.2 Requirements Engineering

Software engineering is a discipline born out of the need to streamline and formalize wherever possible the software design process. In this domain, two approaches have had significant impact on the way software has been specified (and designed): Requirements Engineering (RE) and Object Oriented Analysis (OOA). The two came remarkably together on several principles, but were independently derived by separate communities.

These efforts have not been widely used in the embedded system design community due to a cultural and a practical mismatch between the languages and the needs of the two communities.

In traditional software engineering, the methods proposed for the requirements capture and analysis focus on the definition of the system functionality (the set of reactions or computations), as well as the expression of the concurrency, communication and synchronization properties of the system. Also, the definition of the data model or the domain ontology gets a great deal of attention, given the importance these aspects may have for many general purpose systems (think of databases and other IT systems above all).

However, traditional software engineering often relies on the principle that “no assumption is made on the absolute or relative processing speed of the computations or the events processed by the system”. The dependency of the behavior of the system to be specified on the dynamics of the controlled system is often not considered, given that control of physical systems is indeed a characteristic trait of embedded systems. Also, availability of computation resources (processor speeds), communication bandwidth and other platform resources (like memory, power) is not a concern. These issues are factored out and abstracted in favor of modeling methods that focus on causality, concurrency and synchronization among the procedure composing the system to be developed.

In addition, Object-oriented analysis, as traditionally arising from the programmers’ domain concentrates on the definition of abstract interfaces encapsulating implementation concerns (including the HW platform implementation) and consisting of operations to be invoked asynchronously. Most embedded (control) systems, are however best represented in a modeling framework in which events occur synchronously and a signal interface is defined among computing systems, given that this is the typical model assumed by control theory and systems theory.

Recently, however, there have been great efforts in deriving specialized extensions (or profiles) for system design out of object oriented formalisms such as UML and SysML.

Unified Modeling Language (UML) is a standardized general-purpose modeling language for software engineering. UML includes a set of graphical notation techniques to create abstract models of specific systems. UML has succeeded the concepts of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) and combine them under one framework. It can model concurrent and distributed systems, and targets to be a standard in software engineering.

The SysML (Systems Modeling Language) is the union of an extension and a dialect (Profile) of UML 2. It supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. SysML was developed to provide systems engineers with a domain-specific modeling language to specify complex systems that include non-software components (e.g., hardware, information, processes, personnel, and facilities), which UML could not satisfy because of its software bias. SysML also reduces UML’s size while extending its semantics to model requirements and parametric constraints. These latter capabilities are essential to support requirements engineering and performance analysis.
UML extensions, such as the MARTE profile [Marte09] and SysML allow modeling platforms and resources as well as time and time-related events and also signal-oriented definitions of interfaces (flow ports in SysML). In the intention of their developers, these profiles are the foundation for the quantitative analysis of time and reliability properties and the guarantee of these constraints.

In reality, while SysML and UML profiles should be credited for an attempt at establishing a formal semantics for modeling systems and describing functional as well as non-functional (time) properties and constraints, they hardly represent a complete and accessible solution, as discussed in the following sections.

RE is most relevant to our discussion in this chapter. It can be decomposed into the activities of requirements elicitation, specification, and validation.

- **Requirements elicitation** is the stage in which, by several means, including interviews, focused groups, and product analysis, the requirements are agreed among the stakeholders. Success of the elicitation stage depends on the use of a language that promotes communication and understanding between (among others) developers, marketing analysts, application domain experts, customers and users.

- **Requirements specification** is the stage in which requirements and the information in them are examined, assimilated and then represented, either formally or informally, in a document and/or a set of diagrams or any other means.

- **Requirements validation** amounts to checking that requirements allow for a feasible solution (i.e. they are not inconsistent and possibly allow for an implementation) and that the system model emerging from requirements subsumes the properties of interest for the system. The specifications document should be validated and verified to ensure that it is complete, consistent and correct.

The process is seldom a straightforward cascade among these three steps. Quite often, designers proceed iteratively and, at the end of each iteration, test whether enough information has been gathered and all requirements identified. The process almost invariably starts as informal in the elicitation stage, but it can be formalized in the specification step assuming availability of languages, methods and tools for describing the identified requirements.

While this is seldom the case in practice, a formal language with a well-defined semantics provides several advantages, including the possibility of automatic parsing and removal of ambiguity. Further, formal languages offer a basis for various ways of reasoning with models, either through consistency checking or simulation/prototyping. Indeed, the majority of research work focuses on the validation stage, assuming availability of requirements written in some formal language and especially on the verification problem, that is, to determine whether the expressed requirements are consistent and are implementable. Consistency checks and implementation feasibility can be performed using formal methods based on model checking, theorem proving or other automatic logical inference or satisfiability (SAT) checkers. In addition, a formal specification language can be executable and allow for early prototyping and simulation of behavior, or property verification.

RML [Greenspan94] was one of the first attempt at providing a formal foundation to requirement engineering, together with a number of other formal languages, including (among others) Z [Spivey09] and VDM [Fitzgerald05]. Today, the list of formal languages for the specification of requirements can be quite long. Most models of computation, as outlined in the following chapters, including Finite State Machines (or FSMs), Petri Nets and their timed variants, like the Timed Automata, Timed Petri Nets or even Hybrid Automata can be used to formally express
the desired behavior of the system. First order- or timed-logic predicates can be used to express desired properties and constraints.

On the other hand, informal techniques should not be quickly dismissed. The appeal and usability of some techniques may be largely due to their relative simplicity and flexibility derived from informality. The use of a formal requirements modeling language does not preclude the concurrent use of informal notations. In fact, the original RML proposal envisioned early use of an informal notation and a transformation process from the informal model into a formal RML one.

### 4.3 System Specification Principles

The PBD principles described in Chapter 3 imply that functionality and architecture may be considered as independent inputs to the design process, albeit any experienced designer would consider the reciprocal influence of the two arms of the Y chart when capturing his/her design. Indeed, when dealing with a system-level design problem, we must consider not only what the system is supposed to do but also what are the restrictions placed on the space of feasible solutions either by constraints posed by the application, by the need to cope with complexity, i.e., the "size" of the design space to be explored, or by constraints posed on cost, power, reliability and availability of the architectural components to be considered for implementation.

Thus, a specification, $S$, consists of a description of the functionality of the controller system to be realized, $\mathcal{F}$, and the controlled system or plant $\mathcal{P}$, together with a set of constraints, $\mathcal{C}$, that is partitioned into constraints on the behavior of the system, $\mathcal{C}_f$, and on the characteristics of the implementation, $\mathcal{C}_i$. The controlled system is typically associated to (or even abstracted by) a set of assumptions $\mathcal{A}$ on its characteristics.

$$S=\{\mathcal{F}, \mathcal{C}, \mathcal{P}, \mathcal{A}\}, \quad C=\{\mathcal{C}_f, \mathcal{C}_i\}.$$  

For example, if we are designing a communication protocol, then we may have constraints on the functionality (e.g., absence of deadlock) and on the implementation (e.g., a bound on the power consumed that the implemented system should not exceed).

The question is now how to capture the initial specification of the system. We argue that the functionality of the design should be captured at highest level of abstraction, i.e., at the level we "enter" the design, in an unambiguous way: given an input, the behavior of the system should have a unique response, i.e., it should be deterministic (or determinate). If the functional description is non deterministic, then we argue that either the specification is incomplete, i.e., the designer has forgotten to specify the behavior of the system for a particular input or that the designer is not interested in the output of the system for that particular value of the input. For example, assume we consider a sequential system whose input set has three elements. If we
implement that system in digital forms then we have to encode each symbol corresponding to the input set a Boolean representation. Given three objects, we need at least two bits to encode them. However, since there are four distinct elements in the Boolean space corresponding to two bits (e.g., \{0,0\}, \{1,0\}, \{0,1\}, \{1,1\}) there will be one combination that does not correspond to any input. The implemented digital system will have a behavior corresponding to the "meaningless" bit combination but the output value does not matter to the designer since, in the absence of faulty behaviors, that combination will never occur.

The constraints on the behavior of the system $C_P$ should be formalized as a set of input-output relationships. In this case then, the system is said to satisfy the requirements if all its behaviors are contained in the set of constraints. The requirements on the implemented system $C_P$ can also be called reaction requirements and are expressed in terms of "physical" quantities such as power or timing. These may be expressed in terms of equalities or inequalities involving the variables of the design and characteristics of the implementation. A common reaction requirement is response time, which bounds the worst-case or average-case delay between an external stimulus (input) of the system and its response (output). These requirements guide the selection of the architecture (platform instance) so that when we substitute for the free parameters of the requirements the actual values of the corresponding parameters of the architecture, we can verify whether the selected implementation is a feasible one.

We consider constraints (sometimes called execution requirements) such as limits on available machine cycles, memory space, battery capacity, and channel bandwidth as characteristics of the elements of the implementation platform, consequently, they will be discussed in Chapter 6.

We argued for deterministic behavior in the specification, yet we showed that nondeterminism has some important aspects in system design that we want to keep in our framework. Nondeterminism in system design refers to two different phenomena.

One that we discussed briefly above is about don't-care values in the output behavior of a system.

Let $u$ and $y$ be vectors of input and output variables respectively in an appropriate domain. In most cases, $u$ and $y$ take values in the space of real numbers, integer or Boolean but it is not infrequent to think of $u$ and $y$ as members of a functional space, for example when we deal with continuous time systems where the inputs and outputs are related by ordinary differential equations. Then the behavior of the system can be given as an explicit function, i.e., $\mathcal{F}: y=f(u)$ or as an implicit function $\mathcal{F}: g(x,u)=0$. Given an input $u$ a deterministic system has a unique $y$ that satisfies the behavior equations.

However, if for a given $u$, the corresponding $y$ is not specified uniquely, then we argue that we should augment the output space with a special element called the don't care value so that when the input variable is in the don't care set, the output takes that special value. In this augmented space then, the behavior is indeed unique. This approach has been followed for years in the domain of logic design and has allowed the development of powerful optimization techniques that can take advantage of the knowledge of the "don't care" set. In fact, a don't care does not mean a multiplicity of correct responses of the system with respect to a single input; it simply says that for a particular set of inputs, the response can be selected at will, and we can use this freedom to optimize the implementation of the design. In fact, after an implementation is chosen, the intended behavior of the system is indeed deterministic; given an input, the output is uniquely defined and satisfies all the equations describing the behavior of the system except for the set of don't cares where it always satisfies the equations.

The other form is about the environment of the system, which is in general free to behave in many different ways and we cannot predict which behavior is going to exhibit at any given time.
This nondeterminism is associated with the specification of the environment as seen by the system under design and so it is not a property of the design. Therefore, when we refer to a reactive system as deterministic in this context, what we mean is not that there is a unique stream of input and output values, but that for every stream of input values that is provided by the environment, the stream of output values that is computed by the system is unique.

For embedded systems, it is often the case that the input and output streams include "time stamps". More precisely, a timed input stream is a sequence of time-stamped input values, such as sensor readings or user commands; and a timed output stream is a sequence of time-stamped output values, such as actuator updates and other generated events that are observable by the environment.

We say that an embedded system is time-deterministic if for every timed input stream, the timed output stream that is computed by the system is unique [Henzinger03, Kopetz97]. Note that time-determinism refers not only to input and output values, but also to the times at which input values are given to the system and the times at which output values are made available to the environment.

If an embedded system computes a unique output value, but may make the value available to the environment (say, by updating an actuator) at different time instants, then the system is not time-deterministic.

Time determinism may be essential for safety-critical real-time systems such as those deployed to control automobiles or aircraft.

4.4 From Informal to Formal Requirements

The general framework outlined with the previous principles needs to be instantiated by an actual language that defines all elements of the description $F, P, C, A$.

In the early stage of requirements elicitation, informal, natural language (plain English) descriptions are often used. They are necessary to communicate with people who do not possess a background in mathematics or computer science, and possibly not even in engineering, such as many of those providing input to this stage and can be effective if properly structured and organized.

As already stated, natural language is subject to misinterpretation due to ambiguity, inconsistencies, omissions and redundancy and it is therefore desirable that requirements are refined or translated, until they eventually find an equivalent form in a formal language. Several proposals exist for how to conduct this translation and also on the formal requirements language that is the final destination of this activity.

At a minimum, an approach to requirements collection and analysis requires a methodology, tools, requirements documents formats and requirements writing rules. We argue that at the very least, between the elicitation and the (early) specification stages the following actions should be pursued:

- Elicit requirements in "structured" natural language, enforcing a writing style that identifies states or working modes, assumptions and assertions using a contract-based approach or, alternatively an explicit identification of pre-requisites, post-conditions and invariants to lower the probability of ambiguities and inconsistencies. This structure eases the transition from English language towards an FSM description.

- Enforce the definition of tests associated to each requirement item
• Enforce the early use of at least a data dictionary to minimize the chance of inconsistencies in the definition of the system names and system variables, including: I/O variable names, system parameters, and system states. The data dictionary can evolve into a true ontology if the complexity of the environment objects handled by the system requires it.

• Use of simple rules to avoid redundancy/inconsistency in the definition of the system reactions.

• Semi-formal or formal diagrams (context diagrams, sequence diagrams and state diagrams) should be eventually produced to complement/refine the text description.

Consider, for example, the following descriptions, extracted from early requirements for an electric water heater (boiler).

(error section) When an error condition is detected, the system is disconnected, power is taken off the load and the error is signalled to the user by blinking all yellow leds.

When the cause for the error condition ceases, in the case of volatile errors the error signalling ends and the system is unlocked. In the case of non-volatile errors the manual unlocking procedure is required, consisting in turning off and then on the apparatus. Unlocking is possible only if both sensors report less than 180°F.

... After detection of a non-volatile error, the error signalling persists even is power is discontinued and then restored, whether the cause of the error is still present or not.

After detection of a non-volatile error, the error signalling persists even when the cause of the error is removed or there is a power failure, until the system is unlocked.

When the cause of the error is removed, the error signalling disappears only after a reset, that is, a system shutdown and a new powerup. All other key presses shall not be considered.

... (diagnostics) when an error occurs, the load is disconnected and the error condition is signalled. To signal the error, all temperature leds, and panel lights will blink.

Later, in the interface specifications

When the boiler is on in thermostat mode, it is possible to set the working temperature that is the control target and the temperature actually measured by the two temperature sensors inside the boiler should be displayed.

To setup the temperature the setup mode must be activated. The setup mode is entered by pressing the “plus” key for two seconds. All leds indicating the temperature are off except the one that indicates the current setpoint.

... If no key is pressed for 5 seconds, the last setting of the desired temperature becomes the effective setpoint and the system goes back in control mode.

... If in setup mode the highest led is on and the plus key is pressed, the lowest led will be set.

...
When the boiler is not in setup mode or antibacterial mode or diagnostics mode, the led bar always shows information about the measured temperature. ...

The temperature control mode can be exited because of an error, in which case the resistor is powered off, and all leds blink, or when the user presses the plus key for two seconds to enter the setup mode.

The measured temperature is computed as the arithmetic average of the two NTC in the system. The interface is not updated immediately, the measured temperature is filtered to avoid sharp variations.

There are several issues with the previous description. First, inputs, outputs and working modes are defined with several terms, sometimes in an inconsistent way.

Examples are:
- temperature sensors/NTC
- load/resistor
- yellow leds/temperature leds
- desired temperature/current setpoint/effective setpoint
- thermostat mode/temperature control mode
- system disconnected/system shutdown/power discontinued/power failure

The enforcement of a data dictionary may help avoiding this redundancy of names and possible inconsistencies. A simple example of data dictionary may consist of a set of attributes associated to each Input, Output and Parameter (including Persistent data). For example, a possible set of attributes associated to each Output and Input signal are:

- A symbolic Identifier
- A short Name to be used in the requirements descriptions
- A Text description that helps clarify the semantics of the signal
- (possibly) an Event on the corresponding input, possibly associated with the availability of new data values
- A Flag indicating whether the signal shall be considered (and managed) as persistent
- The Direction (input/output) of the signal with respect to the (sub)system that is the subject of the description
- The Subsystem (or possibly the controlled system or plant) that is the source of the signal for inputs
- A Time interval or period indicating the validity or the maximum required update time for the inputs and the minimum
- The indication of the Data type: this is an abstract data type in the environment domain, not in the domain of the programming code, that is: date, integer or real rather than int16 or float.
- Units: The physical units for the signal
- The Minimum admissible Value for the signal
- The Maximum admissible Value for the signal
- The Resolution with which the (sub)system expects the value to be provided

The range of acceptable values for the input and output signals, the indication of the resolution and required rates allow simple consistency checks among the interfaces of the subsystems and between the controller and the controlled system.

A simple data dictionary like this may be insufficient in some cases, where user defined structured data types are required and where relations among data items need to be specified.
In this case, an ontology, expressed for example using UML class diagrams (which however require stereotyping the constraints section for adding value, resolution and rate constraints) or SysML (using block Value types).

Another possible issue with the previous description is at the end of the sentence. 

*When the cause of the error is removed, ... a new powerup. All other key presses shall not be considered.*

This is a case of a negative requirement. While negative requirements are an essential part of safety requirements, the previous is a regular functional description and, as often occurs, requirements could be simplified *avoiding negative requirements of this type*, by assuming that the system does not react to any other stimulus except those explicitly indicated in the requirements. In other cases, negative requirements can simply be rephrased in a positive form. The reason to express a requirement in a positive form is that most often, negative requirements are much more difficult to test.

The previous description is also prone to another type of inconsistency:

*(error section) When an error condition is detected, the system is disconnected, power is taken off the load and the error is signalled to the user by blinking all yellow leds.*

*(diagnostics section) when an error occurs, the load is disconnected and the error condition is signalled. To signal the error, all temperature leds, and panel lights will blink.*

*To setup the temperature the setup mode must be activated. The setup mode is entered by pressing the “plus” key for two seconds. All leds indicating the temperature are off except the one that indicates the current setpoint.*

*The temperature control mode can be exited because of an error, in which case the resistor is powered off, and all leds blink, or when the user presses the plus key for two seconds to enter the setup mode.*

Here, two possible mode transitions are described in several parts of the document. While this may not be an issue for the current set of requirements (the definition is consistent, at least within the limits of a natural language description), the replicated definitions open the door for inconsistency at the first update. If the logic of one of these transition changes, all replicated descriptions need to be updated in a consistent way.

To avoid these issues, it suffices to enforce the following rule: *mode transitions are only defined as outgoing transitions from a given mode* (avoiding incoming transition definitions).

Finally, contract-based definitions should be used. According to design-by-contract [Meyer91], system (subsystem) functioning may be considered as a contract between the system (subsystem) and the environment (other subsystems). In the definition based on Assumptions/Assertions pairs, the contract consists of a set of “assumptions” on the environment or the other subsystems (what the “users” of the subsystem promise to be or to behave).

If the environment and/or the subsystems satisfy the assumptions, the system (subsystem) under specification will have the duty to keep its side of the contract, that is, a set of assertions (what it promises to provide/how it promises to behave)

Assumptions and assertions can be specified formally or informally using several languages. In early elicitation stages, when plain English is used, conventional terms can be used to identify them. For example:
• The term **will** can be used to identify assumptions on the environment that derive from the immutable laws of physics.

• The term **must** can be used to describe other assumptions on the environment or the (sub)systems with which our system interoperates.

• The systems behavior and the constraints on it (the assertions of the system) are defined in sentences using the keyword **shall**.

• The keyword **should** is finally reserved for the description of optional requirements.

Going back to our example, requirements can be organized according to a document style in which the data dictionary description is used as a preface, identifying all inputs, outputs, states and parameters. Then, one section is dedicated to any major subsystem. Inside, the subsystem description is detailed with one subsection for each working mode. Finally, for each mode, three sets of reactions are identified: reactions that occur *when entering* the mode; reactions that take place *while* the (sub)system *remains in the working mode* and reactions that occur *when exiting from the mode*. One section of the previous boiler specifications example may be rewritten as follows:

6. Temperature setup mode

6.1 Entering mode

6.1.1 The temporary setpoint temperature $t_{\text{TemporarySetpoint}}$ shall be set to the current value of the temperature setpoint $t_{\text{UserSetpoint}}$.

6.2 While in mode

6.2.1 All LEDs shall be off except one corresponding to the value of $t_{\text{TemporarySetpoint}}$.

6.2.2 When pressing the button B\_UP the value of $t_{\text{TemporarySetpoint}}$ shall be set to the value immediately higher, when pressing B\_DOWN to the level immediately lower. If the temperature is at the maximum level, pressing the button B\_UP shall move $t_{\text{TemporarySetpoint}}$ to the minimum value. From the lowest position, pressing the button B\_DOWN shall bring the temperature $t_{\text{TemporarySetpoint}}$ to the maximum value.

6.3 Exit from mode

6.3.1 If no key is pressed for 5 seconds, $t_{\text{UserSetpoint}}$ shall assume the current value of $t_{\text{TemporarySetpoint}}$, the interface must exit from setup mode and go back to thermostat mode.

6.3.2 After pressing the button Button\_OnOff the interface shall exit from temperature Setup mode, the value of $t_{\text{UserSetpoint}}$ shall be set to the value of $t_{\text{TemporarySetpoint}}$, all LEDs shall be turned off and the interface shall go in Standby mode.

6.3.3 If power is disconnected and then restored, the value of $t_{\text{UserSetpoint}}$ shall remain the one previous to the disconnect.

Figure 1: A sample set of requirements in structured natural language.

Each requirement item is now clearly identified and a label is associated to any of them. This eases the definition of a relation associating one or more tests to each requirement. The definition of explicit links between requirements and tests has many purposes. It helps in the definition of the requirements themselves (when it is hard to find a test for a given requirement, it is very likely that there is something wrong with the way the requirement is specified) and provides support for checking coverage, that is, that at least one test checks any given requirement.

Starting from the set of descriptions in the previously listed sections, a refinement in more formal languages and diagrams will have to be produced resulting in a set of semi-formal or even formal diagrams including:

- State Chart Diagrams - depict the required behavior of the feature with the states, triggers, conditions and transitions (base for the definition of the system behavior)

- Functional Context Diagrams – this diagram highlights the system structure and topology, the main interfaces and the functional dependencies. It contains the listing of all the I/O signal connections at the system level and, after a first decomposition round,
the I/O dependencies among the main subsystems. (it is the base for the definition of the system architecture and the decomposition into subsystems)

- Sequence diagrams, scenarios - list the sequence of actions/events identified for several typical working cases of the system (it is the base per la definition of the test cases)

For example, the previous description translates easily into the portion of state machine of Figure 2. At this level, the indication of the actions to be performed, the transition conditions and the events is still informal (although only objects in the data dictionary should be referred), but the description can be later refined until the model is executable (for simulation or verification) and can be implemented.

**Figure 2: A specification-level state machine**

Figure 3 shows an example of context diagram (in practice, it is a SysML Internal Block Diagram, describing the simple interactions, at the system-level, of our example. Together with the following state and sequence diagrams, it has been produced using the open source UML/SysML tool Topcased [Topcased]).

**Figure 3: A context diagram, highlighting the functional structure and the main relationships among blocks.**

Finally, a sequence diagram describes sequences of reactions that occur in accordance with a protocol or complex behavior. Figure 4 describes (part of) the sequence of actions and events that are required from the detection of an excess temperature on one of the sensors to the generation of the error signal and the recovery from the error condition.
As stated in the previous paragraphs, in parallel with the requirements, the test plan should be developed and each requirements item should be associated to one or more tests.

One possible approach consists in the definition of a test plan document in which each requirements section is mirrored by a corresponding section (better if explicitly linked to it using some tool or simply hyperlinks between document sections). Once requirements are defined with explicit reference to the mode to which they apply, a test definition should consist of:

- The setup sequence that prepares the preconditions or the initial system state for the execution of the test item.
- The set of input data/events that should be applied to perform the test.
- The declaration of the expected outputs, with the values and the exact timing conditions that should be verified
- The expected state of the system at the end of the test (or postcondition), together with possibly all the invariants that must hold before and after the test run.
- (if needed) a teardown sequence to bring back the system in a known state.

A sample test case for the verification of the Requirement 6.3.2 of Figure 1 is:

6.3.2: test 1
Precondition: System in mode Temperature setup, tTemporarySetpoint has value $T_s$.
Input sequence: Press the Button_OnOff key
Expected output: All LEDs are set to off; tUserSetpoint has value $T_s$.
Postcondition: System in mode Standby

The previous example only serves the purpose of showing the association between requirement items and test cases, a full description of a good test methodology is clearly beyond the scope of this chapter.

The methodology developed by Nancy Leveson at MIT and based on Intent specifications [Leveson00a] and the SpecTRM (Specification Toolkit and Requirements Methodology) language [Leveson00b] goes further along the lines of providing support for the specification of safety-critical systems with the necessary degree of formalism, without necessarily require complex notations and retaining readability.

Intent specifications follow the entire development process (not only the requirements stage) and are meant to emphasize design rationale, that is, the why of system specification, as well as what the system is supposed to do. Intent specifications consist of six views of the system to be
specified (Figure 5). Each view is a complete model, with its audience, purpose, notation and language. Views are related and their objects should be connected by traceability links, but they are not to be intended as formally connected by refinement steps.

**View 0** (views are referred to as Levels in the referenced works, but the term level is typically associated with composition or refinement and might be misleading in this context) relates to project management and provides information on the project development plans, stages, resources and project scheduling. An important part of this view is the System Safety Plan. If the plan calls for the development of a hazard list, then the plan should link to the hazard list repository. A manager, customer, or regulatory body may use this section to ensure that the system safety plan is being implemented.

![Figure 5: The dimensions of the Intent specifications](image)

**View 1** is the customer view. Its purpose is to gather the user view on what should be built how to validate the result of the development. This view includes (high-level) system goals, user requirements, design constraints, hazards, environmental assumptions, and system limitations.

System goals are linked by an horizontal relation to the functional requirements that refine them. The implementation of these links, as well as of all the tracking links across layers is delegated to a requirements management tool (see following sections).

This view also includes the highest level (informal) description of the assumptions and constraints on the environment as well as on the operator. These assumptions and constraints are part of the Environment section but apply to the requirements for the system and the interfaces.

Finally, an important part of this view is the Preliminary hazard analysis, which consists of the identification of the system hazard and the association to each hazard of a hazard log entry. The template proposed in [Leveson00a] for each hazard item in the list and the log entries it is composed of is the following:

**H1** Hazard Identification
- **Subsystem:** set of subsystems to which the hazard description applies
- **Operation/Phase:** description of the operations involved in the hazard
High Level Causal Factors: list of possible causes  
Level and Effect: A1-2  
Safety Constraints:  
List of constraints imposed on the system to prevent the occurrence of the hazard  
Analyses Performed:  
Actions Taken:  
Status:  
Verification:  
Final Disposal (Closeout Status):  
Responsible Engineer:  
Remarks:  

The template for the description of system hazards requires the identification of safety constraints that eliminate or control the hazard. Each constraint requires the system not to perform actions that lead to hazardous behaviour and can be refined in several stages (an example from [Leveson00a]).  

SC1: The mobile base must move only when commanded by the operator or when commanded by the Planner and approved by the operator (H1)(FR2.5).  

SC1.1: MAPS must not enter Operator Mode unless the joystick is physically connected to the robot and the joystick is in the neutral position (2.4.6.1).  

SC1.2: The robot must not move unless the deadman switch is depressed (2.4.2.1, 2.4.5.4.1, 2.4.6.3.3).  

SC1.3: If the operator releases the deadman switch and then later depresses it again, all previous commands must be ignored and a new command must be issued before any robot movement occurs (2.4.2.5, 2.4.7.3, 2.4.5.4.1).  

Rationale: A long enough time may exist between releasing the deadman switch and depressing it again that the environment may have changed and previous commands may no longer be safe.  

Constraints must be traced down to design decisions in the lower level view and to the verification and validation tests at the same level in order to simplify the check that all safety constraints have been implemented in the system and tested by creating a direct path from the constraints to their realization.  

View 2 defines the System Design Principles, that is, the architecture-level view of the system(s) (controller and controlled). System architects address the definition of the system in terms of the physical principles and laws upon which the design is based.  

The actual content of this view may vary, but this architecture-level description typically includes:  

• An Interface Design describing the boundaries between subsystems and the controller and controlled systems.  

• A Controls and Displays section containing the models (or descriptions) of the interface devices between the system and the user.  

• An Operator Task Design Principles section listing the tasks required from the operator.  

• The Verification and Validation section includes information about the validation of the models included in this view. Verification and validation can be performed using several means including simulations, analyses, formal methods. The verification and validation procedures shall also apply to the system hazard analysis.
**View 3** is used to define the *Blackbox Behavior* level. This view provides models and notation for reasoning about the functional design at the system level with the main components or subsystems, their behavior and the interactions among them. This level provides the formal models that define the contract between systems engineering and component engineering, but also the behavior information that is used for informal review, formal analysis, and simulation. The models of this view are written using SpecTRM-RL (SpecTRM Requirements Language). SpecTRM-RL provides support for the description of the input and output dependencies between the controller and the controlled system, and also provides an extension of the standard Statechart mechanism for the description of the controller behavior, but also for the definition of the assumptions on the state of the controlled environment. This formal part of the description supports execution of the specification as well as automated safety analyses.

A SpecTRM-RL model is defined after view 2 has identified the main subsystem components and requires first the identification of the system functioning modes or macro-states. For each of them, specifies the conditions under which each mode becomes active, the conditions under which outputs are produced and their content, and how each inferred state variable assumes a value. This information is sufficient to form an executable and formally analyzable model.

Accordingly, a SpecTRM-RL model has three components: (1) a specification of the supervisory interface to the component, (2) a specification of the control modes for the component, and (3) a model of the controlled process or plant including relevant operating modes, state variables, and interface variables (measured and manipulated process variables as reflected by the inputs and outputs to the controller).

![Figure 6: The three components of a SpecTRM-RL description.](image-url)

The description consists of a set of requirements related to: Outputs, Inputs, Modes, State Values.

The **Output** description combines the description of the definitions and the constraints that (statically) apply to the output values and the description of the logic according to which outputs are updated.

The first part matches the concept of Data dictionary discussed in the previous section, with additional attributes that mostly relate to timing, such as the definition of an *initiation delay* and *completion deadlines*. The initiation delay is the time interval that elapses (at best), between the triggering event/conditions and the availability of the output value. Also, SpecTRM-RL prescribes the specification of not only the minimum guaranteed rate, but also of the maximum possible rate (minimum time between outputs) for capacity control.
The second part consists of the description of the system transitions in which the given output is assigned values for the computation of the output update part of a finite state machine reaction. The description of the transition that results in new output values is provided as a table. The table encodes the triggering condition for the transition, the value assigned to the output is provided separately as an assignment expression preceding the table. The table represents a disjunctive form. The condition is true if all rows (in AND) match the condition. Columns are considered in OR. Each entry in the table can assume the values T (true), F (false) or * (don’t care).

The authors argue that such table format is easier to understand than the other conventional notations for the expression of state machines.

The format for the description of the inputs is very similar to the SpecTRM-RL outputs. A set of attributes describes the input and a set of truth tables defines use of the input in the evaluation of the system behaviour. Among input attributes there are several referring to its timing behaviour. The minimum and maximum time between inputs is specified. If input messages arrive faster than the assumed maximum load, an exception signalling and recovery mechanism needs to be specified. Actions may go from dropping excess inputs to shedding other functionality to free up resources. The SpecTRM-RL format highlights the need for these decisions and provides assistance in making good choices.

Another input attribute is its obsolescence or the time interval after which the input value loses significance.

The appears in attribute, which is present in every model element, when associated to inputs allows performing several checks such as: inputs outside their expected range, inputs that are too early, too late, missing altogether, or having nonsensical values. The appears in attribute provides a navigational aid to verify that these criteria have been met.

Finally, a set of attributes links inputs to the corresponding relevant outputs as well as the states and modes that depend on the input. The latency attribute covers output-to-input feedback. An input should always arrive after the latency period of matching outputs, and should only be received in response to one (or more) such outputs, spontaneous or early receipt of inputs shall be treated as exceptions.

Modes represent a set of (controller) system-level behaviors. SpecTRM-RL specifies modes with a limited number of attributes followed by the specification of the set of transitions between any two modes. The conditions under which the mode changes are again expressed by means of tables.
State elements are used to represent the inferred model of the controlled system. Inputs (from sensors) are used to update the inferred state, and this may in turn change the controller mode or its outputs. An example SpecTRM-RL state specification from [Leveson00b] is included in figure 5.

Transitions among states are again specified using AND/OR transition tables. Of course, not only is it good practice to add “don’t care” to the possible values for the inputs, but similarly, an “unknown” state may be define to model conditions in which the estimate of the state of the modelled process is not considered to be reliable. Unknown gives the engineer writing the requirements a way to reason about situations in which the controller does

Finally, the last three views cover the design and implementation stages. As such, they fall outside the scope of this chapter. However, for completeness, we mention them shortly.

View 4 contains the design representation of the implementation of subsystem components. This level is written in a design notation appropriate to the component.

View 5 provides information necessary to reason about the physical implementation of a component. This may include code listings for short software modules, pointers to code repositories for larger software projects, or hardware schematics.
**View 6** provides all the information that is required to retain the system operational during its lifetime, to manage changes and corrections, including operator manuals, audit procedures, error reports, and change requests.

The Intent Specification and SpecTRM-RL provide means to bridge the gap between the informality that is required in the elicitation stage and the formal description (provided by attributes, attribute values and truth tables) that is required for automatic processing by tools and the evaluation of at least some inconsistencies, omissions and incompatible specifications.

Among its weaknesses, there is the need for a state based or at least operational description of most behavior constraints. This may be ill-suited to the description of constraints on dynamic systems.

Another set of languages goes all the way into the formal description of behavior and constraints. These languages are based on some logic and/or mathematical background and are typically specialized or at least best suited for the description of a given class of constraints.

Possible formal languages for expressing requirements include finite state machines (FSMs), Petri nets or other automata, including timed versions (timed Automata, timed Petri Nets) and other logics (first-order logic, timed logics) for the description of behaviors, LTL or CTL (for constraints description) the Z language, and many others.

### 4.4.1 Z

Z (pronounced Zed), is a specification language that works at a a high level of abstraction to describe formally complex behaviors. It is based on set theory and first-order predicate logic and was originally developed at the Oxford University Computing Laboratory (OUCL) in the late 70s, and used in non-trivial "real world" projects. Z is now defined by an ISO standard and is public domain.

In Z, a specification is decomposed in sections called *schemas*. Each schema can be linked with a commentary which explains informally the significance of the formal mathematics. Syntactically, a Z schema is represented as a named box partitioned into two parts, variable declarations and optional predicates relating the variables.

Schemas are used to describe both static and dynamic aspects of a system. The static aspects include:

- the states it can occupy;
- the invariant relationships that are maintained as the system moves from state to state.

The dynamic aspects include:

- the operations that are possible;
- the relationship between their inputs and outputs;
- the changes of state that happen.

A mathematical framework describes both the state space of the system and the operations that can be performed on it. The data objects in the system are described in terms of mathematical data types such as sets and functions. The description of the state space included an invariant relationship between the parts of the state.

The notation of predicate logic is used to describe abstractly the effect of each operation. The effects of the operations are described in terms of the relationship which must hold between the input and the output, abstracting from implementation details.
As an example of Z specifications, Andy [Andy94] shows an elevator operation model together with proofs on liveness and safety properties. It is particularly suited to non-constructive requirements specifications. Semantics-preserving refinement techniques allow formal translation of Z specifications to executable code (e.g., [King90]), and formal proof techniques are described in [Diller90].

4.4.2 UML/SysML with OCL

The Unified Modeling Language, UML represents a collection of engineering practices that have proven successful in the modeling of large and complex systems and has emerged as the software industry's dominant object oriented modeling language. UML has been designed as a wide-ranging, general-purpose modeling language for specifying, visualizing, constructing, and documenting the artifacts of software systems. Recently, it has been extended and specialized to cope with the needs of system engineering, giving rise to the SysML language.

UML comprises of a metamodel definition and a graphical representation of the formal language, but it intentionally refrains from including any design process. The UML language in its general form is deliberately semiformal and even its state diagrams (a variant of statecharts) retain sufficient semantics variation points in order to ease adaptability and customization.

The designers of UML realized that complex systems cannot be represented by a single design artifact. According to UML, a system model is seen under different views, representing different aspects. Each view corresponds to one or more of diagrams, which taken together, represent a unique model. Consistency of this multi-view representation is ensured by the UML metamodel definition. The diagram types included in the UML 2.0 and SysML specification are represented in Figure 9, as organized in the two main categories that relate to Structure and Behavior. SysML additional diagrams are outlined in red. Diagrams for which SysML has a different specification are grayed.

![Figure 9: the taxonomy of UML/SysML diagrams.](image)

The UML has metamodel elements that are suited to the representation of Requirements. For example, the language defines the Use Case diagram for the explicit purpose of representing a high-level (user requirements-level) description of the interaction of the system with external agents.

Use case diagrams are very informal and only represent very high-level behaviors, listing the main actors such as users (with their roles) and system components and the interaction scenario in which they appear (an example in Figure xx). While this description can be useful for
systems that require heavy interactions with human users and in which users must be classified by their role and the operations they are allowed to take part in because of their role, in embedded systems they are seldom useful.

Use cases can however be associated with other diagrams, such as state diagrams and sequence diagrams, which help clarify the intended behavior. In fact, as in our previous short introduction of the state and sequence diagrams, several UML/SysML diagrams can be used for the purpose of clarifying a system specification.

In addition, SysML provides for a new diagram, explicitly labeled as Requirement Diagram. A Requirement Diagram consists of blocks stereotyped as «requirement», each of which represents a simple text description with the an additional identifier property an possibly other user-defined properties such as the verification method to be applied.

Requirements blocks can also be further characterized by user-defined categories (e.g., functional, interface, performance) and organized in a hierarchy of containment or according to a set of predefined relationships, which include DeriveReqt, Satisfy, Verify, Refine, Trace, Copy. The Refinement and Tracing relationships are especially important since they allow the expression of requirements refinements using the aforementioned behavior diagrams (use cases, state diagrams, activity diagrams, sequence diagrams) and the establishment of tracing links between requirements blocks and testCase blocks (as shown in Figure 10: An example of SysML requirement diagram. Figure 10, extracted from the INCOSE SysML tutorial [SysML09].

Figure 10: An example of SysML requirement diagram.
Other relations that apply to «requirement» blocks are stereotyped as «satisfy» (an example in Figure 11, also from [SysML09]) and link a requirement to the (sub)system model that provides the design solution to the requirements problem.

Figure 11: An example of requirement diagram with refinement links.

Also, of particular importance for the purpose of describing the behavior of the controlled system is the Parametric diagram of SysML. This diagram can be used to express constraints (equations) between value properties associated to ports (the P and A parts of the specification).

The constraints are associated to the block representing the system dynamics and capture the dynamics equations and other constraints (Figure 12).

Figure 12: A block definition diagram linking constraints to block models.

A parameter diagrams defines the usage of the constraints in an analysis context and makes explicit the way in which constraints apply to the input and output values of the model components (an example in ). Constraints can be expressed informally, or using a formal language. The formal languages recommended by UML/SysML are MathML (a standard markup language for the definition of mathematical formulas [MathML]), or OCL (shortly described in the following).

As is typical of most UML tools, the computational engine is not defined by the standard and it is meant to be provided by applicable (external) analysis tool and not by the modeler.
Finally, the Object Constraint Language (OCL), developed by IBM, is a declarative language (a first-order predicate logic) used for describing rules that apply to UML models, initially defined as a formal specification language extension to UML. OCL is used to describe (constraint) expressions on UML models. An OCL expression is typically used to specify invariants or other type of constraint conditions that must hold for the system. OCL expressions refer to the contextual instance, that is the model element to which the expression applies, such as classifiers, e.g. types, classes, interfaces, associations (acting as types) and datatypes. Also all attributes, association-ends, methods, and operations without side-effects that are defined on these types can be used.

OCL can be used to specify invariants associated with a class of system objects (named as classifier in UML/SysML). In this case, it returns a boolean type and its evaluation must be true for each instance of the classifier at any moment in time (except when an instance is executing an operation).

Preconditions and Postconditions are other types of OCL constraints that can be possibly linked to an operation of a classifier and their purpose is to specify the conditions or contract under which the operation executes. If the caller fulfills the precondition before the operation is called, then the called object ensures the postcondition to hold after execution of the operation, but of course, only for the instance that executes the operation.

Figure 13: An example of SysML parametric diagram.
4.4.3 Analyzing the Specification with respect to Completeness, Robustness, Safety Criteria

As the software specification is developed, completeness and consistency criteria should be checked and enforced. The description of consistency checks in [Leveson95] provides a long list of 60 such criteria. Checking and enforcing these conditions with the help of automatic tools is possible provided the model is constructed according to formal specifications. The following examples refer to (hierarchical) finite state machine models, or equivalent formalism (including specTRM-RL).

A method for improving robustness consists in ensuring that a state transition or behavior is specified for any possible set (subset) of input values. Automating this type of completeness check is possible but possibly quite difficult to satisfy for large input alphabets.

Similarly, determinism should be enforced in the specification of the controller system. In this case, it is possible to detect inconsistency in the specification by identifying all transitions that can be made true at the same time by a set of input values.

Other possible checks include the analysis of reachability criteria for system states. In the context of hazard analysis we want to ensure that hazardous states are not reachable from any working state, possibly not only on a functioning system, but also under given fault assumptions. Reachability analysis techniques are typical of formal methods, using forward or backward search in the state tree or n-bounded model checking.

Software deviation analysis (SDA) is a technique for evaluating the robustness of a specification [Reese97]. It examines the response of the software to deviations in software inputs, exploring software operation in imperfect environments. The analyst provides a list of deviations in inputs and identifies those outputs that are safety critical. The result of the analysis is a list of scenarios, which are combinations of input deviations and system states sufficient to cause a deviation in a safety-critical output, obviously an important flaw in the system design. One application of deviation analysis is to determine what might result if particular inputs from the environment are lost.

4.5 Requirements in the Development Process

Requirements are "alive" and "volatile". Very often they are not completely known at the start of a system's development, but rather evolve during the analysis phases of a project and beyond. Users, developers, and customers, all improve their knowledge and acquire a better
understanding of needs, issues and opportunities during the system's development and maintenance. During the lifetime of a product, function updates, additions and possibly removal are quite common.

For example, in response to the rapid changes and strong competition in markets, products are often requested to change or be customized in order to meet customers' needs.

As for any other type of document or model that is subject to change and updates, it is important that requirements are managed by a tool that provides for change management, concurrent access control and versioning, access rights control, together with standard features like the association of metadata to requirements documents, ownership management and indexing.

Figure 15: Evolving requirements leads to a new set of evolving artefacts.

In addition, requirements live in the context of a development process where any change in them prompts a chain of updates in the design documents and models as well as in the hardware or code implementation of the functions. For example, as shown in Figure 15, a new set of requirements evolving from Version 1.0 to 2.0 triggers a series of updates along the entire development chain, with new design models and new code.

Also, given that each requirement item is associated with a set of tests, testing procedures are clearly affected by changes in the requirements and should be similarly subject to version control.

There are at least two sets of references that should be maintained from requirements items. One set of references/links should follow the refinement of the requirements into the design elements that are generated in response to the requirements, down to the code implementation. Another set of links should connect each requirements item to the corresponding set of (system-level or component-level) tests, that must be performed to verify satisfaction of the requirement by the system (or one of its subsystems/components).

4.5.1 Tracking Requirements into Design and Further Refinements

Being able to track requirements to design and implementation is beneficial and sometimes even necessary for several reasons:

- In case of requirements changes/updates such links allow to locate quickly the part of the design and implementation that needs to be changed/modified, without the need of going through heavy documentation.
- When system tests show failure to meet requirements, such capability rapidly provide the indication of the requirements that are affected.
In case updates are performed on the code, because of bug fixes or updates/adjustments at test time, reverse links from code to the requirement that originated the code allow to locate the requirement that can possibly be modified. When the requirement document is extensive, performing these tasks manually without the tracking between requirement and implementation becomes very difficult if not impossible.

![Diagram](image)

We need to know what part of the design/code has been produced in response to what requirement

Conversely, a set of backward links indicates what requirement originated the given design item or code section

**Figure 16**: forward and reverse links for tracking requirements refinement.

### 4.5.2 Requirements and Testing

A different set of references should be maintained between the requirement items and the tests that are defined to verify requirements on the final product. These links should not only be defined and maintained, but it is also important that they are defined early, at the same time requirements are produced. Following well-established practices (ESA-PSS European Space Agency process guidelines [ESAPSS], Extreme programming [Beck05], to name a few), white box or functional tests should be defined together with the requirement they are supposed to test. The enforcement of such a process and the maintenance of such links ensure the following.

- The definition of one or more tests contextually with the definition of a requirement item helps write better requirements, ensures that they are testable and the definition of the test procedure helps clarify the meaning of the requirement (if needed).
- Provides means to measure of the coverage of the functional tests with respect to requirements. Typically, the functional test plan is required to achieve 100% requirements coverage.
- The existence of links between requirement items and tests allows to quickly identify and change a test or set of tests for a requirement in case the requirement is modified or updated.
- Such links are also useful to identify the affected requirement whenever a test fails during the functional testing stage.

Finally, a fundamental part of requirements management includes tools and methods for change management and versioning. Even if requirements evolve, keeping track of requirement changes and how different versions of the requirements relate to different products or versions
of a product or its software is necessary for building the correct tracking management system between requirements and implementations (Figure 3).

Hence, a fundamental part of the project is the selection and use of a set of tools that enables requirements tracking, both with respect to design models and code implementations, and with respect to functional test descriptions.

Also, these tools should provide content management and versioning of requirements, design models and code.

Examples of tools that can be used for this purpose are DOORS (from IBM), Reqtify (from TNI), but also Word documents or HTML documents, possibly maintained by a content management platform (several solutions, also open-source exist), or stored and managed as a set of wiki pages.

The generation of links to and from requirements and code implementations is possible in Simulink models using the "Report Generator" package of Mathworks.

Similarly, for versioning, there are several options, including open source packages like svn or cvs or commercial solutions like ClearCase, Visual SourceSafe, Synergy.

4.6 Tools and Methods for Requirements Engineering

Among tools for the creation, management, tracking and versioning of requirements, IBM Rational (formerly Telelogic) DOORS is probably today the most popular in the industry. DOORS helps capture, track and manage user requirements. It is strongly oriented towards informal specification handling. It does not promote, provide or directly support any formal language for specification, but provides a framework in which formal definitions and propositions can be linked to specification objects and possibly handled by external tools.

DOORS manages a database in which information about Projects is stored and managed. Each Project has Folders, Formal Modules, Link Modules and Descriptive Modules.

Folders are provided for organization of Modules, which are used to organize requirements and specifications according to a user-defined taxonomy, but their content is not restricted to specification documents. For example, in Figure 17 (the DOORS screenshots are taken from the IBM website [IBM10]), the project Training Car Project consists of several Formal Modules, containing the requirements, but also user profiles, the definition of tests, and architecture design documents.
Each Module consists of Sections and each Section, in turn, contains Objects.

Each Object is an element of the specification, consisting of an Header or Text (possibly both, although it is not recommended), or a picture or a table. Each Object is characterized by a set of default attributes, including:

- A unique identifier, assigned by the Tool
- Information on who created the object and when, as well as the creation mode
- When it was last modified and by who
- The Heading, short text and Text information
- A picture info, in case the object is a picture
- Table attributes, in case it is a table

Further, users can customize objects by adding attributes. Each attribute is assigned a type. DOORS provides predefined types, but allows the user to derive custom types from them. Availability of a type checking feature helps enforcing the consistency of the value declarations for specification objects with respect to types.

Figure XX shows the objects that make the Formal Module *Car User Reqs* of Figure XX. The first column shows the identifier associated to each object. Some of them consist only of the header. Others contain text. Two attributes for Cost and Priority are also associated to Objects.

Requirements objects contain requirements items expressed (in this case) in plain English. This is often the case. However, it is in principle possible to use the text field for formal formulas, to be parsed and analyzed by external tools.
In DOORS, access to objects can be controlled in several ways:

- With the definition and assignment of access rights to users or class of users.
- Using the product features for managing concurrent access control and versioning.

Among the most notable feature of DOORS is probably the capability of managing the links among objects as well as between objects and external models or even code. This feature allows the realization of the relations for tracking the refinement of requirements in the design and implementation as in Figure 16 and the relation between requirements and functional tests. Figure 19 shows links that connect user requirements to their refinement into functional specifications.

In addition, DOORS can manage links to design elements managed by external tools or even sections of code. This capability can be used, for example, to link a requirement to a Simulink element (a subsystem, a block or even a stateflow transition) that provides the design-level solution to the requirement (as in Figure 8).

In Simulink/stateflow it is similarly possible to setup backward links to the requirements and to associate such links to model elements.
4.7 Concluding Remarks

Specifying a system is today an art more than a science but it does have an essential role in the final quality of the design both in terms of quality and performance. Several attempts have been done in the past in the area of software system design to formalize specification or requirement capture but these efforts have not made their way through to system level design. We have reviewed briefly some of the approaches to requirement engineering and we have derived some principles that can be used to understand the issues surrounding the topic and that may yield to a novel way of capturing and analyzing requirements and specifications.
4.8 References


[Marte09] Modeling and Analysis of Real-time and Embedded systems, The Official OMG MARTE Web Site


[Topcased] The Topcased project web site http://www.topcased.org


The King’s toaster

Once upon a time, in a kingdom not far from here, a king summoned two of his advisors for a test. He showed them both a shiny metal box with two slots in the top, a control knob, and a lever. "What do you think this is?"

One advisor, an Electrical Engineer, answered first. "It is a toaster," he said. The king asked, "How would you design an embedded computer for it?" The advisor: "Using a four-bit microcontroller, I would write a simple program that reads the darkness knob and quantifies its position to one of 16 shades of darkness, from snow white to coal black. The program would use that darkness level as the index to a 16-element table of initial timer values. Then it would turn on the heating elements and start the timer with the initial value selected from the table. At the end of the time delay, it would turn off the heat and pop up the toast. Come back next week, and I’ll show you a working prototype.

The second advisor, a computer scientist, immediately recognized the danger of such short-sighted thinking. He said, "Toasters don’t just turn bread into toast, they are also used to warm frozen waffles. What you see before you is really a breakfast food cooker. As the subjects of your kingdom become more sophisticated, they will demand more capabilities. They will need a breakfast food cooker that can also cook sausage, fry bacon, and make scrambled eggs. A toaster that only makes toast will soon be obsolete. If we don't look to the future, we will have to completely redesign the toaster in just a few years."

"With this in mind, we can formulate a more intelligent solution to the problem. First, create a class of breakfast foods. Specialize this class into subclasses: grains, pork, and poultry. The specialization process should be repeated with grains divided into toast, muffins, pancakes, and waffles; pork divided into sausage, links, and bacon; and poultry divided into scrambled eggs, hard-boiled eggs, poached eggs, fried eggs, and various omelette classes."

"The ham and cheese omelette class is worth special attention because it must inherit characteristics from the pork, dairy, and poultry classes. Thus, we see that the problem cannot be properly solved without multiple inheritances. At run time, the program must create the proper object and send a message to the object that says, 'Cook yourself.' The semantics of this message depend, of course, on the kind of object, so they have a different meaning to a piece of toast than to scrambled eggs."

"Reviewing the process so far, we see that the analysis phase has revealed that the primary requirement is to cook any kind of breakfast food. In the design phase, we have discovered some derived requirements.

Specifically, we need an object-oriented language with multiple inheritances. Of course, users don’t want the eggs to get cold while the bacon is frying, so concurrent processing is required, too."

"We must not forget the user interface. The lever that lowers the food lacks versatility, and the darkness knob is confusing. Users won’t buy the product unless it has a user-friendly, graphical interface. When the breakfast cooker is plugged in, users should see a cowboy boot on the screen. Users click on it, and the message 'Booting UNIX v.8.3' appears on the screen. (UNIX 8.3 should be out by the time the product gets to the market.) Users can pull down a menu and click on the foods they want to cook."
"Having made the wise decision of specifying the software first in the design phase, all that remains is to pick an adequate hardware platform for the implementation phase. An Intel Pentium with 48MB of memory, a 1.2GB hard disk, and a SVGA monitor should be sufficient. If you select a multitasking, object oriented language that supports multiple inheritance and has a built-in GUI, writing the program will be a snap."

The king wisely had the software developer beheaded, and they all lived happily ever after.

Of course we do not own this story in any way… it appears on several sites on the Internet and we could not figure out the original authorship (the story must have been around for quite some time, judging from the platform recommended by the computer scientist).

Also, in other versions on the Internet, the computer scientist only gets thrown in the moat …. While we do not condone violence, (least of all on computer scientists!) this is a fable for adults and the message is probably best delivered this way …. 