The Complexity Challenge in Embedded System Design

Invited Paper

Hermann Kopetz
TU Wien, Austria
hk@vmars.tuwien.ac.at

Abstract

The ever-increasing functionality and the non-functional constraints that must be satisfied by embedded systems lead to an enormous growth in the complexity at the system level. In this paper we investigate the notion of cognitive complexity and argue that it is not the embedded system, but the models of the embedded system that must be simple and understandable. The introduction of appropriate levels of abstraction in modeling and the associated concept formation help to reduce the emerging complexity by focusing on the relevant properties and omitting irrelevant detail, thus leading to a simpler representation of the evolving artifact. In the second part of the paper we present examples of basic-level concepts that we find essential in the design of distributed embedded systems and summarize the insights by presenting a set of concise design patterns that support the component-based design of embedded systems.

1. Introduction

A recent report on *Software for Dependable Systems: Sufficient Evidence?* [1] from the National Academies contains as one of its central recommendations . . *One key to achieving dependability at reasonable cost is a serious and sustained commitment to simplicity, including simplicity of critical functions and simplicity in system interactions. This commitment is often the mark of true expertise.*

We consider *simplicity* to be the antonym of *cognitive complexity*. Trying to analyze an embedded system design from the point of view of *cognitive complexity* and deriving design guidelines from this analysis is a novel approach. As with any new approach, the first try is most likely to be immature and will require further refinements. But we have to start somewhere.

It is impossible to reason about the behavior of an SoC (System on Chip) consisting of a billion transistors, switching with a frequency of 1 GHz at the level of activity of every transistor. Designing for *simplicity* therefore means that we have to build artifacts, the relevant properties of which can be analyzed by simple models at different levels of abstraction.

The major challenge of design is the building of a software/hardware artifact (an embedded computer system) that provides the specified services under given constraints and where relevant properties of this artifact can be modeled at different levels of abstraction by models of adequate simplicity.

Top-down computer-system design starts with a conceptualization of the intended *high-level behavior* of the planned system. For example, when we intend to build a computer-controlled braking system for a car, we start from a high-level behavioral specification that relates the inputs to the outputs in the domains of value and time: *When the brake pedal is pressed, the computer should initiate the braking action within one millisecond.*

A high-level specification can be expressed by a *high-level behavioral model* of the intended computer system. During a *top-down model-based design process,* this model is refined and transformed at multiple levels until an executable representation (again a *model*) that can be executed on the selected distributed hardware target platform has been generated. Given such a detailed behavioral model we might investigate other relevant properties of our evolving artifact by building appropriate analysis models in a *bottom-up fashion*. For example, we might develop a reliability model, or a power/energy model. Other properties, e.g., the *mechanical shape* or the *cost* of our artifact can be addressed by again other models, where each model looks at those properties of the artifact that are of interest for the given purpose. All these models are tied together by the fact that they are all concerned with the *same artifact*. Some of these models can be derived from a high-level specification of the system structure and of the solution algorithm,

978-0-7695-3132-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ISORC.2008.14

while others need to consider the very details of the specific software/hardware implementation. *Model building* is thus a pervasive activity during the design of an embedded computer system [2].

The rest of the paper is structured as follows: Section two deals with the essence of model building. Section three introduces the concept of *cognitive complexity* and deals with concept formation. Section four looks at the role of *determinism* and *simultaneity* in model building and system understanding. Section five elaborates on the notion of failure and argues that our abstractions must remain intact, even in the presence of failures. Section six presents some *basic-level concepts* that we have found *central* to the design of large distributed embedded systems. Section seven summarizes our finding by presenting *design patterns* that help to reduce the cognitive complexity of a design. The paper terminates with a conclusion in Section eight.

2. The Essence of Modeling

Given the rather limited cognitive capabilities of humans we can only understand the world around us if we build *simple models* of those properties that are of relevance and interest to us and disregard (abstract from) detail that we consider irrelevant for the given purpose. *A model is thus a deliberate simplification of reality with the objective of explaining a chosen property of reality that is relevant for a particular purpose*. For example, in Celestial mechanics a level of abstraction is introduced where the whole diversity of the world is reduced to a *mass point* in order that the interactions with other mass points (heavenly bodies) can be studied.

When a new level of abstraction (a new model) is introduced that conceptualizes the properties relevant for the given purpose and disregards the rest *simplicity* emerges. This simplicity, made possible by proper abstractions, give rise to new insights that are at the roots of the *laws of nature*. As Popper[3] points out, due to the inherent imperfection of the abstraction process, laws of nature can only be falsified, but never be proven to be absolutely correct.

The recursive application of the *principles of abstraction and refinement* leads to a hierarchy of models that Hayakawa[4] calls the *abstraction ladder*. Starting with *basic-level concepts* that are shaped early in the development of the human mind and are essential for understanding a domain, more general concepts are formed by *abstraction* and more concrete concepts are formed by *refinement*. Consider, e.g., the basic-level concept *chair* that is generalized in the more abstract concept of *furniture* and is refined in the lower-level concept of *armchair*.

At any given abstraction level, entities can interact and give rise to a scenario of *emergent complexity* until selected properties of this scenario that are relevant for a particular purpose are captured in a new conceptualization (model) at a higher level of abstraction resulting in an *abrupt simplification* [5, 6].

A novel structure or behavior may come into existence when the interactions of entities result in global properties that are not found at the level of the entities. We call these properties, which emerge from the interaction of the entities *emerging properties*. For example, the unique properties of a *diamond*, such as *brilliance* and *hardness*, which are caused by the coherent alignment of the Carbon-atoms, are substantially different from the properties of graphite (which consists of the *same atoms*). We can consider the diamond with its characteristic properties a new concept, a *new unit of thought*, and forget about its composition and internal structure. Simplicity has emerged as a result of the intricate interactions among the elements that help to generate a *new whole* with its new characteristic properties.

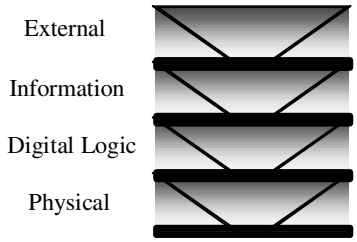

Fig. 1: Emerging Complexity versus Emerging Simplicity at different Abstraction Levels.

Fig. 1 shows this interrelationship between *emerging complexity* and *emerging simplicity* as we move up the abstraction ladder, taking the *Four Universe Model* of a computer as an example. This model, published by Avizienis[7] in 1982, introduces four levels of abstraction when modeling a computer system. At the lowest level, the *physical level*, the analog signals of the circuits are observed, such as the rise time of the voltages as a transistor performs a switching operation. The analysis of a circuit behavior at the analog level becomes difficult as soon as more and more transistors get involved (*emerging complexity).* The next higher level, the *digital logic level,* abstracts from the physical analog quantities and the dense time and introduces binary logic values (*high* or *low*) of signals at discrete instants, resulting in a much simpler representation of an elementary circuit, e.g. an AND gate (*emerging simplicity*). Complexity creeps in again as we

combine more and more logic circuits. The next higher level, the *information level*, lumps a (possible large) sequence of binary values into a meaningful data structure, (e.g., a pointer, a real-valued variable or a complete picture) and introduces powerful high-level operations on these data structures. Finally, at the *external level* only the services of the computer system to the environment, as seen by an outside observer, are of relevance.

Whereas the natural scientist must uncover the regularities of a given reality and find a suitable level of abstraction in order to formulate appropriate models and theories that explain the observed phenomena, the computer scientist is—at least theoretically—**in a much better position**: The computer scientist has the freedom to design the system—an artifact—which is the subject of his modeling. The requirement to build artifacts the properties of which can be analyzed by *simple models* should thus be an explicit design driver.

In many areas of computer science, this principle of building artifacts that can be modeled by *simple models* is violated. For example, the temporal behavior of a modern pipelined microprocessor cannot be captured in a *simple model*.

3. Cognitive Complexity

In the previous Section we have used the words *complexity* and *simplicity* assuming their established semantics. In this Section we elaborate in more detail on what makes a model *complex or simple*.

We take the view of Edmonds [8] that complexity can only be assigned to models of physical systems, but not to the physical systems themselves, no matter whether these physical systems are natural or man made. As mentioned before, a physical system has a nearly infinite number of properties—every single transistor of our billion-transistor SoC consists of a huge number of atoms that are placed in space and have their own identity. We need to abstract, to build models that leave out the seemingly irrelevant detail of the micro-level, in order to be able to reason about properties of interest at the macro-level

What, then is a good measure for *the cognitive complexity (c-complexity)* of a model in our context? We are looking for a quantity that measures the cognitive effort needed to understand a model by a client. *Understanding* means that the client can connect the model to his experience, which is recorded in his mind forming his personal *conceptual landscape*. *We consider the elapsed time needed to understand a model by a client a reasonable measure for the cognitive effort and thus for the c-complexity of a model relative to the client.*

The c-complexity of a model depends on the relationship between the existing *conceptual landscape* in the client's mind versus the concepts deployed in the representation of the model, the interrelations among these concepts and the notation used to represent the concepts. The conceptual landscape is the result of the client's *life-long* experience. If the client is already familiar with the concepts and the representation used to describe a model she/he is likely to judge the model as *simple*, since the mental effort to understand the model will be small.

3.1 Concept Formation

We are not always in the fortunate situation to have well-established and stable concepts at hand to describe our models. In some cases we have to form new concepts to capture the essence of an abstraction. This brings us into the realm of *concept formation*. *A thorough concept formation is an essential prerequisite for any formal analysis or formal verification.*

As a child grows up it continually builds and adds to its *conceptual landscape* by observing regularities in the perceptions and utility in grouping properties of perceptions into new categories [9]. These new categories must be interlinked with the already existing concepts in the child's mind to form a consistent *conceptual landscape*. By abstracting not only over perceptions, but also over already existing concepts, new concepts are formed. In the course of cognitive development and language acquisition *names* are associated with concepts. The *essence* of a concept associated with a name can be assumed to be the *same* within a natural language community (*denotation*), but different individuals may associate different *shades of meaning* with a concept (*connotation*), dependant on their *individual existing conceptual landscape* and the differing personal experiences in the acquisition of the concept.

In the world of science, new concepts are introduced in many publications in order to be able to express new *units of thought*. Often these concepts are named by a *mnemonic*, leading to, what is often called, *scientific jargon*. In order to make an exposition *understandable*, new concepts should be introduced sparingly and with utmost care. A new scientific concept should have the following properties:

Utility: the new concept should serve a useful well-defined purpose.

Abstraction and Refinement: the new concept should abstract from lower-level properties of the scenario under investigation. It should be clear what properties *are not part* of the concept. In the case of refinement of a *basic-level concept*, it should be

clearly stated what additional aspects are considered in the refined concept.

Precision: The characteristic properties of the new concept must be precisely defined.

Identity: The new concept should have a distinct identity and should be significantly different from other concepts in the domain.

Stability: The new concept should be usable uniformly in many different contexts without any qualification or modification.

Analogy: If there is any concept in the existing *conceptual landscape* that is, in some respects, analogous to the new concept, this similarity should be pointed out. The analogy helps to establish links to the *existing conceptual landscape* of a user and facilitates reasoning. According to [10], p.5: *Analogical reasoning mechanisms are important to virtually every area of higher cognition, including language comprehension, reasoning and creativity. Human reasoning appears to be based less on an application of formal laws of logic than on memory retrieval and analogy*.

3.2 Human Communication

A significant share of human communication is concerned with establishing agreement among the used concepts in the minds of the communicating parties and thus with the *alignment of the conceptual landscapes*. Different subjective connotations of a word can be the cause of confusion. Technical communication requires precise agreement on all levels of relevance. Different refinements of an agreed abstract concept can be incompatible with each other and thus lead to serious misunderstandings. It is therefore of utmost importance that the properties of a concept that are not specified at a given level of abstractions are made explicit, in order that incompatible implicit assumptions can be detected.

3.3 Associations vs. Relations among Concepts

The *conceptual landscape* of the human mind does not consist of isolated concepts, but of a network of concepts that are inter-linked by *associations* and *relations* [10] p.8:

*An **association** is a link between a cue and an associate. Activation of the cue evokes the associate, but the reverse is not necessarily the case.*

*A **relation** is a binding between a relation-symbol or predicate, and one or more arguments. . . Each argument constitutes a dimension in the space represented by the relation, and the number of arguments provides a metric for conceptual complexity. . . .*

One of the most important ways that associations differ from relational knowledge is that there is no symbol that represents the associative link. That is, a person can have an association between a cue and an associate without necessarily knowing that the link exists. They know something, but do not (necessarily) know that they know it. A further difference between associations and relations is that an association is *unidirectional*, while a relation is *omni-directional*.

While *associations* seem to be the main linking mechanism of concepts in *unconscious mental processing*, *relations* seem to be characteristic for *explicit thought* in higher cognitive processes.

According to the *Relational Complexity Theory of Halford [11],* there is an upper limit on the size of relations that the rational human mind can handle concurrently. Available evidence suggests that adult humans are capable to process at most a *quaternary relation* in parallel, i.e. a relation of rank five (that consists of five components: four arguments connected by one predicate). Children up to the age of four can handle relations of rank three and up to the age of eleven they have difficulty in handling relations higher than rank four (i.e., three arguments and one predicate). Whenever a scenario is faced that requires to process relations of a degree higher than rank five, humans deploy *simplification strategies* to handle the situation.

3.3 Simplification Strategies

We know of three strategies to simplify a complex scenario in order that it can be processed by the limited cognitive capabilities of humans: *abstraction, partitioning*, and *segmentation[10]*.

*Abstraction (*Halford calls it *conceptual chunking)* refers to the formation of a higher-level concept that captures the essence of the problem-at-hand and reduces the complexity of the scenario by omitting irrelevant detail. Abstraction has been discussed at length in the previous Sections.

Partitioning (also known as *separation of concerns*) refers to the *spatial division* of the problem scenario into nearly independent parts that can be studied in isolation. Partitioning is at the core of *reductionism*, the preferred simplification strategy in the natural sciences over the past three hundred years. Partitioning is not always possible. It has its limits when *emergent properties* are at stake, such as the above-mentioned example of the *diamond*.

Segmentation refers to the *temporal decomposition* of complex behavior into smaller parts that can be processed *sequentially*, one after the other.

Segmentation reduces the amount of information that has to be processed in parallel at any particular instant in order that the capacity limitations of the human mind are avoided. Segmentation is difficult or impossible if the behavior is formed by highly concurrent processes, depends on many interdependent variables and is strongly non-linear, caused by positive or negative feedback loops.

4. Determinism and Simultaneity

In this Section we investigate the role of determinism in understanding a system, and the relation between determinism and simultaneity. Let us start with a general definition of *determinism* which is derived from *[12]*: *A model behaves deterministically if and only if, given a full set of initial conditions (the initial state) at time t_o, and a sequence of future timed inputs, the outputs at any future instant t are entailed.*

Deterministic models of the natural phenomena are the basis of our technical civilization. The identification of abstraction levels and the development of corresponding deterministic models, where the non-determinism of the world at the lower levels does have only a negligible effect, are at the root of scientific discovery and engineering practice.

Deterministic models have many advantages over non-deterministic models since they enable the confident prediction of future behavior. *Determinism of behavior is closely related to the rational analysis of behavior,* since determinism is a sufficient precondition for logical reasoning. It is easier to abstract from deterministic models than from non-deterministic models. Furthermore it is difficult to validate a non-deterministic system, since repeated test cases do not have to produce identical results in a non-deterministic environment. *For these reasons we should try to build artifacts that can be modeled by deterministic models whenever possible.*

Under what circumstances are the emergent properties of a complex system *deterministic*? Given that the properties at the lower level are deterministic and care has been taken in the composition, particularly with regard to the handling of *simultaneity*, the properties at the higher level will be deterministic as well. Computer science is in the fortunate position that the behavior of the basic building block, the *synchronous digital logic*, is deterministic at the logical level (see Fig. l). But this determinism is often sacrificed when yielding to the pressure of building high-performance processors.

What happens if the lower-level abstractions behave *non-deterministically*? If this non-determinism of the lower level has a negligible effect on the properties of interest in the higher-level models (e.g. random voltage fluctuations of signals at the physical level that have an irrelevant effect on the logic value of a Boolean variable) then the non-determinism of the lower level can be neglected. In other cases special algorithms must be put into place in order to reinstate the required deterministic behavior (e.g., *consensus* or *agreement* algorithms[13]). As mentioned in the introduction, in many embedded applications a deterministic behavior is required at the *external level*.

The proper handling of *simultaneous* events poses a special challenge to the designer of a deterministic distributed system. Simultaneity is at the root of a number of difficult problems in computer science: the *meta-stability problem* in the hardware, the *mutual exclusion problem* in operating systems, and the *consistent message ordering problem* in distributed systems. Whenever two distributed events are perceived to occur simultaneously at a site of a physically distributed system, an *additional rule* (e.g., the rule that in case of simultaneity of the events on channel A and B, the event on channel *A* will be processed before the event on channel *B*) must be put into effect to establish a uniform temporal order of the seemingly simultaneous events. This additional rule must be applied consistently to the two considered events at all sites of the distributed system, implying that the *simultaneity of events* has to be established *consistently* at all sites. Otherwise the two events will be ordered by the *perceived (non-simultaneous) temporal order* at one site and by the *additional rule* at another site (that perceives the two events simultaneously), possibly resulting in differing orders at the two sites. It is principally *impossible* to always arrive at a system-wide *consistent notion of simultaneity* in a distributed system that allows the occurrence of physically distributed events at any instant of the *dense* timeline. The *sparse time model* [14] that restricts the occurrence of events that are in the *sphere of control* of the computer system to the *activity intervals* of a *sparse time base,* solves this problem, as explained in Section 6.1.

Simultaneity of events is closely related to concurrency. Unconstrained concurrency can give rise to simultaneity, to race conditions and resource conflicts that destroy deterministic behavior. The arbitration of simultaneous access to resources requires time and energy, which can be saved if these concurrent resource conflicts are avoided *by design* in the first place.

A good example for an architecture that supports determinism is the time-triggered architecture (TTA) [15], which is based on a sparse time-base. In the TTA resource conflicts are not resolved by the competition

of processes (which can lead to non-determinism), but by the progression of a global notion of sparse time.

One common method to deal with (low-probability) non-determinism is to neglect this non-determinism in the higher-level models and to call any behavior that deviates from the deterministic model of a system a *failure* of the system or a *miracle*. (We should rather call it a failure of the *model* than a failure of the *system*). If such failures are caused by probabilistic phenomena and can be considered to occur rarely and independently in replicated artifacts, then we can mask the effect of these kinds of failures by redundancy, e.g. by triple modular redundancy (TMR). In a TMR system three results are computed by three synchronized replicas and are then compared by a *voter* that outvotes an erroneous result. A necessary prerequisite for the application of TMR is the *replica-determinism* [16], i.e., all correct replicated units must always produce identical results (see Section 6.3).

5. On Failures

From the complexity management point of view we must ensure that the abstractions and models that are developed for understanding the system are ***stable*** *even in case of failures (error containment principle).* This implies that the system must be partitioned into independent fault containment units (FCUs). A fault can then be contained within its FCU and errors must be detected at the boundaries of the FCUs in order to avoid error propagation to other units that have not been affected directly by the fault occurrence. If we neglect the *error containment principle*, then fault-diagnosis cannot be done at the level of abstractions developed for understanding a fault-free system, but must be performed at lower levels of abstraction, diminishing significantly the utility of the developed high-level models.

The *error containment principle* constrains the physical structure, the *partitioning* (see Section 3.3), of the evolving computing system. John van Neumann has formulated a related principle already in 1956 [17]: *If you look at automata which have been built by men or which exist in nature, you very frequently notice that their structure is controlled to a much larger extent by the manner in which they might fail and by the (more or less effective) precautionary measures which have been taken against their failure.*

The uncontrolled sharing of resources by independent processes without proper error containment mechanisms violates the *error containment principle* and makes the analysis of errors, and thus the maintainability of systems, more complex.

6. Examples of *Basic-level Concepts*

In this Section we describe some *basic-level concepts* (see Section 2) that we have refined over the years to reason about the behavior of component-based distributed embedded systems at the *system level[18]*. The *basic-level concepts* that we are considering here are *component, message, sparse time*, *cycle* and *state*.

At the *system level*, we distinguish between the *distributed computer system under consideration* (we call it a *cluster*) and its *environment*. A *cluster* consists of a set of components, connected by a cluster-internal communication system that transports messages among the components. A *component* is a hardware/software unit that accepts input messages, provides a useful service and produces after some *elapsed physical time* output messages containing the results. The syntax and semantics of the component service must be precisely specified in a component-interface model [19], preferably without reference to the concrete component implementation. The interface model should specify the algorithms implemented by the component in an executable form, such that the algorithms can be automatically translated into different implementation technologies (e.g., program on a CPU or an FPGA). Furthermore, the interface model must include the temporal parameters of the intended component behavior. In Section 6.4 we discuss some refinements of the component concept.

A *message* is an atomic data structure that is formed for the purpose of *transmitting data* and *control signals* from a sending component at a given instant to one or more receiving components that receive the message at a later instant. The message concept does not make any assumption about (abstracts form) a specific transport mechanism or about the meaning of the bit-vector contained in the data field of the message. However, the time it takes to transport a message from the sender to the receiver is part of the control aspect of the message concept.

6.1 Sparse time

In order to be able to establish a system wide consistent notion of simultaneity of *distributed events* we consider a *global sparse time base* as a key concept in spatially distributed computer systems.

In the *sparse time model* the continuum of real-time is partitioned into a sequence of alternating intervals of *activity* of duration ε, and *silence* of duration Δ. The duration of these intervals depends on the *precision* of the clock synchronization[20]. Within the system, the occurrence of events of significance (e.g., the *sending of a message)* is restricted to the *activity intervals.* We

call the events that occur within an activity interval *sparse events*. We number the activity intervals by the positive integers and call the integer number assigned to an activity interval, the *global timestamp* (or *timestamp* for short) of events happening in that interval. We consider all sparse events that happen within the same *activity interval* ε as having occurred *simultaneously*. A *system-wide consistent temporal order* of all *sparse events* occurring in a distributed system can now be established on the basis of their global timestamps. We assume that all events that are in the *sphere of control* of the system (e.g., the sending of messages) happen within the activity intervals ε of the sparse time base and are therefore *sparse events*. Events that are outside the sphere of control of the system and occur on a dense time base must be transformed to *sparse events* by the execution of an agreement protocol[20].

The granularity of the sparse time-base must be chosen in agreement with the achievable precision of the clock-synchronization at the selected abstraction level. For example, at the hardware level a high precision of the global time can be established by hardware means, e.g. by the IEEE 1588 standard[21]. However, at the level of the application software, the jitter introduced by the operating system and by the middleware services impacts the precision of the time and thus the granularity of the sparse time-base must be adjusted accordingly.

6.2 Cycle

A cycle is a key concept in any periodic control system and in any time-triggered system [22]. A *cycle* is a period of physical time between the repetitions of regular events. A cycle is specified by the duration of its period and the position of its start, the *cycle start phase* relative to some given time reference. In order to avoid the exponential explosion in the combination of cycles and thus an enormous increase in complexity, we establish the rule that all cycles must be in a *harmonic relationship*, i.e., the duration of any cycle must be a power of two of the duration of the shortest allowed cycle. If we use a binary representation of physical time, such as the IEEE 1588 time standard [21], then any cycle duration can be defined by specifying a particular bit in this binary time-representation. The start of a cycle, i.e., the *cycle start phase*, can then be identified by specifying the offset of the start instant of the cycle from the start instant of the respective period in the global time representation.

6.3 State

State is introduced in order to separate the *past behavior* from the *future behavior* of a component. We follow the definition of Mesarovic[23], p.45 :

The state enables the determination of a future output solely on the basis of the future input and the state the system is in. In other word, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of a system. Knowing the state "supplants" knowledge of the past. . . . Apparently, for this role to be meaningful, the notion of past and future must be relevant for the system considered.

The sparse time model introduced above makes it possible to establish the consistent system-wide separation of the past from the future that is necessary to define a *consistent system state* between selected (periodic) *activity intervals* of the sparse time base. Without a proper model of time, the notion of *state* of a distributed system is an ill-defined concept.

The notion of *state* is an essential element of the definition of *determinism* (see Section 4), since the specification of the *initial* state is the starting point of any deterministic behavior. This is one reason why we consider the interlinked concepts of *sparse time*, *cycle* and *state* as key concepts in the realm of dependable distributed real-time systems.

In order to facilitate the recovery of a component in case its state has been corrupted by a fault, it is necessary to introduce *periodic instants* where the state of a component is small. We call such a small state where no process is active and all communication channels are empty a *ground state* and the corresponding cycle a *ground cycle*. In a *state-aware design*, the precise specification of the ground state at the start of the ground cycle (the planned recovery points) is a key requirement.

6.4 Component Refinements

In this Section we refine the component concept and distinguish between computational-components (*c-components*) and interface components (*i-components*) [24].

A *c-component (computational component)* can be modeled by a state-machine, i.e. it can contain internal state. We distinguish between two types of c-components, *time-triggered (TT) c-components* and *even-triggered (ET) c-components*. A *cycle*, specified by its duration and phase, is associated with every TT-c-component. Whenever the global time reaches the start-instant of this cycle the TT-c component starts its processing activity[25]. The TT-c component finishes its computation after an a *priori* known time-interval

after the start-instant (the Worst-case execution time or WCET).

An ET-c-component starts its processing activity whenever a new input message arrives. It will terminate when the processing activity is finished.

At the chosen *system-level of abstraction* we are not interested in the detailed internal algorithms of a component, or in the exact *hardware/software boundary* within a component. This hardware/software boundary may change as a component is developed. In the Platform-Independent-Model (PIM) of a component an algorithm might be expressed in a high-level specification language (e.g., System C), augmented by the required temporal parameters, while the Platform-Specific-Model (PSM) might consist of software implemented on a CPU, an FPGA or a direct hardware implementation.

An *i-component* (interface component) provides a connection from the *cluster* to the *environment of the cluster*. It thus acts as a gateway, transforming the *idiosyncratic* representation of the information in the environment to the *standardized cluster-internal* form. It musts be ensured that no error outside the sphere of control of a component can interfere with the control logic of the i-component (e.g., spurious interrupts generated by a fault in the environment).

6.5 Message Refinements

We find it useful to introduce the following refinements of the message concept, an *Event-triggered (ET) message, a Time-Triggered (TT) message,* and a *TT-state message.*

An ET message conforms to the widespread notion of a message. An ET message is processed as soon as possible on a *best effort basis.* It is appended to a sender queue by the producing process at the sender and consumed by the communication system from the sender queue as soon as the transmission starts (as soon as the channel becomes free). At the receiver side, the ET message is appended to the receiver queue and remains in the queue until it is consumed by its final receiving process. If the sending queue or the receiving queue overflows, an error-condition is raised. In order to avoid a queue overflow, the message-production rate of the sender must be—in the long run—the same as the message-consumption rate of the receiver. In the short run deviations of the sender-production rate and the receiver-consumption rate are buffered by the queues. It is difficult to give temporal guarantees for ET messages.

The processing of a TT message is triggered by the progression of a global notion of time. With every TT message a *cycle* with a given *duration* and *phase* is associated. Whenever the global time reaches the start instant of this cycle, the clock generates a control signal to trigger the start of transmission of a pending TT message. The TT communication system guarantees to deliver the TT message within an *a priori* known interval.

A *TT-state message* is fetched from a send buffer (non-consuming read) and is transmitted in every cycle. On arrival, a *TT-state message* overwrites the contents of the receive buffer. There are no queues associated with a TT-*state message*, and therefore no queue overflow can occur. The writing of a TT state message by the sender, the transmission of the TT state message by the communication system, and the reading of a TT state message by the receiver can proceed at differing rates. Thus the coupling between a sender and a receiver is less tight when TT-state messages are used than when ET messages are used.

It must be assured that there is no unintended temporal interference among the transport of messages of independent subsystems. Such unintended interferences lead to hidden dependencies and open new error-propagation paths in the temporal domain that are difficult to detect and analyze.

A well-defined message concept that includes the temporal aspect of message transmission, can unify the diversity of inter-process communication constructs (some of them ill-defined with respect to time)that have been proposed in the literature, such as *semaphores*, *test-and-set*, *remote procedure call*, *rendevouz*, *event counters*, *common memory*, *path-expression*, *signal-wait, request-reply, mailbox*, etc.

7. Design Patterns

A *design pattern* is a standard solution schema for an often-occurring problem. Design patterns codify engineering practices and fundamental insights and help the designer to apply proven solutions in his development work. Design patterns have their origin in architecture, based on the seminal work of Christopher Alexander [26].

7.1 Introduce a Sparse Time Base

In a distributed real-time system, a sparse global time-base of appropriate precision forms the foundation for the specification and design of temporal properties, e.g.:

- Identification of a *system-wide consistent state* at selected instants (e.g. ground state).
- Specification of the temporal properties of component interfaces
- Specification and validation of the temporal validity of real-time data

- Synchronization of control actions
- Error detection in the temporal domain

It is much easier (and energy efficient) to generate such a global time at start-up and to maintain the established synchronization during system operation than to establish temporal order out of an unsynchronized environment every time anew when temporal order is needed—the approach taken by some of the *atomic broadcast protocols.*

It is much easier to maintain order, once it has been established, than to generate order out of chaos every time order is needed.

7.2 Introduce Few Orthogonal Concepts

Every new concept (abstraction) must be understood and learned with some effort by a user, just as a new word has to be learned when studying a new language. We should try to introduce only as few orthogonal *basic level concepts* as absolutely necessary and, if needed, refine these concepts at a lower level, or introduce more abstract concepts at a higher level, without changing the meaning of the *basic-level concept* (see Section 2). A new *basic-level concept* should be stable and useable in many different scenarios without any qualifications, restrictions or modifications. The above example of a *message* (see Section 6.5)) is a good example of a *basic-level concept* that covers all kinds of data-transfer and signaling issues of local and distant inter-process communication, is stable and has a strong identity.

7.3 Simplify by Partitioning

Wherever and whenever possible, separate unlike functions and implement them in nearly autonomous components with well-defined interfaces. *In particular separate communication from computation, since these are two different technologies, and provide a standard observable message interface between the two.*

If components are nearly autonomous and maintain their abstraction even in the case of failures, then the reuse of components in another application is facilitated. *Composability of components* and proper *simplification by partitioning* are thus related.

7.4 Simplify by Segmentation

It is straightforward to divide a sequential process, into segments that can be analyzed one after another and thus to reduce our *cognitive load* at any instant. Our mind is not well adapted to deal with concurrency, where such a segmentation is much more difficult to achieve. Concurrency should therefore be introduced with utmost care, since it is often the source of unmanaged complexity. Concurrency of highly intertwined processes that access a common data pool is of particular concern.

7.5 Deterministic Models

The identification of deterministic models in then natural sciences has been a driving force for the technological progress. Logical reasoning, which is based on deterministic models, is the fundamental scheme of rational analysis. Whenever possible we should therefore try to build artifacts the behavior of which can be described by deterministic models. Determinism is essential if we want to mask errors by TMR.

We are aware of the fact that non-determinism is *unavoidable* in an open environment, where an unknown number of actors interact in an ad hoc opportunistic manner (e.g., in an ambient intelligence scenario). However, the unjustified introduction of non-determinism must be avoided in order to improve the understandability of systems.

7.6 Network of Model Hierarchies

As pointed out in Section 2, *the recursive application of the principles of abstraction and refinement leads to a hierarchy of models that Hayakawa[4] calls the abstraction ladder.*

For each aspect of investigation of an artifact (e.g., behavior, energy efficiency, reliability, etc.) we can build a distinct hierarchy of models, with more abstract models at the top and more concrete models at the bottom. Each one of these models should be small enough to be understandable and maintainable. Some of the lower-level models of a model hierarchy can be generated automatically from higher-level models—this is the idea of *model-based design*.

It is important to encapsulate different concerns in different models. For example, the specification of an algorithm should be independent of the concrete implementation of the algorithm (*principle of algorithm separation*) in order that different implementation options that may be orders-of-magnitude different from the point of view of non-functional constraints can be followed. This *principle of algorithm separation* helps to solve the *technology obsolescence* problem as well. An algorithm that is specified in a high-level specification language can be recompiled to a new hardware base without a major redesign effort.

The model hierarchy should contain redundancy, such that the effects of failures—particularly the

corruption of state—can be contained and tolerated without a major impact on the service to the system's environment.

The different models and model hierarchies should not be integrated into a colossal *integrated model*, since this would violate the *principle of partitioning*. Instead, the models should be kept separate and designed with well-specified and observable input/output interfaces that support inter-operability such that the outputs of one model can become the inputs for another model. (Alternatively, the models can be designed to feed to a common data-base.) In this way we generate a *network of model hierarchies*.

8. Conclusion

In order to be able to reduce the cognitive complexity of embedded computer systems, we must take account of the limited cognitive capabilities of humans and develop artifacts that can be modeled at different levels of detail by models that are simple enough for the human mind to understand. Referring to results obtained in the cognitive sciences, we have analyzed what makes a system representation simple or complex from the point of view of cognition. Based on these insights we gave a set of example of useful basic-level concepts to model embedded systems at the system level. Finally we tried do summarize our finding in design patterns that lead to a simplification of a design.

Acknowledgements

The work contained in this paper has in part been supported by the European IP DECOS and by a grant from the Austrian FIT IT program (TT-SoC project). Many discussions within the DECOS project team and the TTSoC project team, particularly with Roman Obermaisser, Christian ElSalloum, Bernhard Huber from the TU Vienna and Neeraj Suri from TU Darmstadt, are warmly acknowledged.

References

1. Jackson, D., M. Thomas, and L.I. Millet *Software for Dependable Systems: Sufficient Evidence*. 2007: National Academic Press. 120.
2. Broy, M., *The 'Grand Challenge' in Informatics: Engineering Software Intensive Systems*. IEEE Computer, 2006. **39**(10): p. 72-80.
3. Popper, K.R., *The Logic of Scientific Discovery*. 1968, London: Hutchinson.
4. Hayakawa, S.I., *Language in Thought and Action*. 1990: Harvest Original, San Diego.
5. Gershenson, C. and *Complex Philosophy*. in *Philosophical, Methodological and Epistemological Implications of Complexity Theory*. 2002. La Habana, Cuba.
6. Bar-Yam, Y., *Dynamics of Complex Systems*. 1997: Westview Press, .
7. Avizienis, A. *The Four-Universe Information System Model for the Study of Fault Tolerance.* in *Proceedings of the 12th FTCS Symposium*. 1982. Los Angeles: IEEE Press.
8. Edmonds, B., *Complexity and Scientific Modelling*, in *Foundations of Science*. 2000. p. 379-390.
9. Vigotsky, L.S., *Thought and Language*. 1962, Boston, Mass.: MIT Press.
10. Halford, G.S., W.H. Wilson, and S. Phillips, *Abstraction, Nature, Costs, and Benefits*. 1996, Department of Psychology , University of Queensland, 4072 Australia.
11. Halford, G.S., et al., *How Many Variables Can Humans Process?* Psychological Science, 2005. **16**(1): p. 70-76.
12. Hoefer, C., *Causality and Determinism: Tension, or Outright Conflict.* Revista de Filosofia, 2004. **29**(2): p. 99-225.
13. Barborak, M., M. Malek, and A. Dahbura, *The Consensus Problem in Fault-Tolerant Computing.* ACM Computing Surveys, 1993. **25**(2): p. 171-220.
14. Kopetz, H. *Sparse Time versus Dense Time in Distributed Real-Time Systems*. in *Proc. 14th Int. Conf. on Distributed Computing Systems*. 1992. Yokohama, Japan: IEEE Press.
15. Kopetz, H. and G. Bauer, *The Time-Triggered Architecture*. Proceedings of the IEEE, 2003. **91**(January 2003): p. 112-126.
16. Poledna, S., et al., *Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems.* IEEE Transactions on Computing, 2000. **49**(2): p. 100-111.
17. Neumann, J., *Theory of Self-Reproducing Automata*. 1956, Urbana, Ill: University of Illinois Press.
18. Jones, C., et al., *DSOS Conceptual Model*. 2003: University of Newcastle upon Tyne, Techn. Report CS-TR-782, TU Vienna, Technical Report 54/2002, Qinetic Technical Report TR030434, LAAS Technical Report. p. 1-122.
19. Kopetz, H. and N. Suri. *Compositional Design of Real-Time System: A Conceptual Basis for the Specification of Linking Interfaces*. in *ISORC 2003--The 6th International Symposium on Object Oriented Real-Time Computing*. 2003. Hakodate, Japan: IEEE Press.
20. Kopetz, H., *Real-Time Systems, Design Principles for Distributed Embedded Applications; ISBN: 0-7923-9894-7, Seventh printing 2003*. 1997, Boston: Kluwer Academic Publishers.
21. IEEE, *1588 Standard for a Precision Clock Synchronization Protocol for Network Measurement and Control Systems*. 2002.
22. Kopetz, H., *Pulsed Data Streams*. 2006, Institut für Technische Informatik, TU Wien, Austria. p. 7.
23. Mesarovic, M.D. and Y. Takahara, *Abstract Systems Theory*. Lecture Note in Control and Information Science. Vol. 116. 1989: Springer Verlag.
24. Kopetz, H. *The Time-Triggered (TT) Model of Computation*. in *Real-Time System Symposium 1998*. 1998. Madrid, Spain: IEEE Computer Society Press.
25. Kopetz, H., et al. *Periodic Finite-State Machines*. in *ISORC*. 2007. Santorini Island, Greece: IEEE Press.
26. Alexander, C., S. Ishikawa, and M. Silverstein, *A Pattern Language*. 1977, New York: Oxford University Press.