Considerations on the Complexity of Embedded Real-Time System Design Tasks

Bernhard Rumpler and Wilfried Elmenreich
Vienna University of Technology
Institute of Computer Engineering, Real-Time Systems Group
Treitlstraße 3/3, 1040 Vienna, Austria
{rumpler,wil}@vmars.tuwien.ac.at

Research Report 70/2006

Abstract – *In this paper, relational complexity theory is used to discuss various aspects of the complexity of computer system design tasks, with a special focus on embedded real-time systems for high dependability environments. Designing such systems is often especially complex as various timing and dependability constraints must be met. An approach is presented that allows for a strict conceptual separation of components by minimizing the relational properties of components. An example component structure that adheres to the presented concepts is shortly described.*

1 Introduction

1.1 Complexity in Embedded Real-Time System Design

Embedded real-time systems are becoming increasingly important in many areas of our life. Especially in industrial, automotive, and avionics control applications there are high dependability requirements as a failure can have fatal consequences, including loss of life. Design faults often have their roots in lacking understanding or overlooking of some system aspects. To be able to avoid design faults, the complexity of the design process must be limited. Complexity management is receiving increasing attention in recent years but there are no universal theories or techniques describing how to deal with complex design tasks. This paper tries to focus attention on some basic issues in the domain of embedded real-time systems where solutions for dealing with some aspects of complex systems have been found.

1.2 Cognitive Complexity

In this paper the terms *complexity* and *cognitive complexity* are used interchangeably as complexity is considered only from a cognitive perspective. The cognitive complexity of a given task describes the amount of cognitive resources that are required to perform the task. If a task has high resource requirements this becomes manifest in the increased time required for the task and in the number of errors that occur.

1.3 Objectives

To be able to make statements on the complexity of design tasks of embedded real-time systems, a conceptual framework that has its roots in cognitive psychology is needed. The characteristics of human cognition are the reason for what we perceive as either simple or complex.

This work uses relational complexity theory [1] as a theoretical framework to discuss various aspects of computer system design. It is explained which factors affect component complexity and which properties of component interfaces help to reduce the complexity of design tasks. This paper describes a constructive design approach for embedded real-time systems in high dependability domains where latent design faults caused by unmanaged complexity cannot be tolerated. It is argued that design for simplicity is an important principle for the creation of these systems which must be understandable and certifiable to the highest criticality levels.

1.4 Related Work

There exist various approaches to measure complexity of software systems, most of which focus on pro-

gram code complexity [2]. There also exists some work on measuring the architectural complexity of software. These approaches focus on the high-level structure rather than on the implementation details of any specific source module [3, 4, 5].

Complexity metrics can be used to assess the overall design of a system according to some properties and then use this assessment to perform corrective actions early in the development process [5] or use it for re-engineering purposes [3]. Thus, while complexity metrics give an overview over some specific characteristics of system properties (those measured by the metric), metrics in general do not consider the complexity of individual tasks. For instance, a metric that measures the overall architectural regularity of a system may return a high value, which means that the system is highly regular and should thus be easy to understand. Nonetheless, tasks performed with this system, even "general understanding" tasks, may be very hard as the measure does not consider the complexity of the tasks to be performed. But it is the tasks that the developers are performing, which must receive attention when making statements on design complexity. This paper focuses on the task-based theory of relational complexity.

Visual models that depict the relationships between the components of a system support comprehension [6]. These techniques are orthogonal to the concepts presented in this paper.

2 Basic Concepts

2.1 Relational Complexity

Relational complexity is a theory from cognitive psychology that is supported by a wide variety of empirical data [1, 7, 8]. According to the theory, the processing load of a cognitive task is determined by the complexity of the relations that must be processed in a given step. The *arity* of a relation describes its dimension, i.e., the number of independent elements that must be considered simultaneously. For example, BIG(dog) is a binding between the unary relation BIG and one argument. The relation can be interpreted as expressing a state or an attribute. Class membership, e.g., DOG(fido) can also be expressed as a unary relation. A binary relation such as BIGGER(dog,mouse) relates two components. Univariate functions and unary operators can be represented at this level. Ternary relations are needed to represent bivariate functions and concepts such as transitivity or class inclusion. Formal similarity mappings, independent of content, can also be made at this level. Greater abstraction is thus related to higher dimensionality [9]. Quaternary relations are the most complex we can handle [7]. At this level four-way comparisons are possible. Examples for quaternary relations are matching tasks of objects according to four independent attributes, such as color, shape, filling pattern, and orientation.

Relational complexity theory proposes that the cognitive demand can be reduced through conceptual chunking and segmentation [1]. *Conceptual chunking* means recoding of a relation to lower dimensionality. For instance, velocity can be considered as a function of distance and time ($v = s/t$) which is a ternary relation. It can, however, also be considered as a unary relation if it is conceptualized as SPEED(50km/h). The chunked variables become inaccessible, i.e., they cannot be considered. *Segmentation* means to reduce problems of high dimensionality into a number of tasks of lower dimensionality that can be solved serially. However, not all relations can be decomposed into simpler relations and then recomposed into the original relation. Even when relations are decomposable, people may not have the necessary strategies. This explains the dependence on expertise of higher cognitive processes [9].

2.2 Computer System Architectures

A *computer system architecture* describes the overall design of a class of computer systems that share a set of common characteristics. This definition is not constrained to pure hardware but also includes software aspects. Architecting is a consequence of system complexity [10]. Architectures reduce the effort of the design process as they guide the designers by constraining the design. Examples for computer system architectures according to this definition are, for instance, the DECOS architecture [11] which is mentioned in more detail in section 5.2, and AUTOSAR [12], which is an architecture for automotive control systems. Also the well-known personal computers we all use in our homes and offices, together with the operating system and device drivers represent computer system architectures according to the definition used in this paper.

2.3 Task-Based Complexity

To be able to develop a theory on design complexity, the basics of what makes a task complex have to be considered. From this viewpoint it does not make sense to consider the overall complexity of the whole

system, as the complexity always depends on the task that must be performed. The central concern of this paper is thus task-based complexity.

There are two main factors that affect the performance of a task. First, the knowledge (or expertise) in the problem domain and second, the inherent complexity of the task itself [9].

Unfortunately, there exist no cognitive process models for design tasks of computer systems. As such models must be developed for the various computer system architectures and are not yet available, this paper focuses on the task of general system understanding which is—without doubt—involved in many tasks including, e.g., constructive design and maintenance.

2.4 Components

A *component* is a self-contained subsystem that can be used as a building block for a larger system. This definition of component reflects its most general meaning in technical systems without being restricted to pure software components on the one hand and software-hardware components on the other hand. Besides the constructive system design approach that uses components to build up a larger system, components can also be seen as the result of a top-down design process in which a large system is decomposed into a number of smaller components, e.g., for complexity management.

Low coupling between modules and high cohesion inside each module are commonly accepted features of good software design [13, 14]. From a complexity management viewpoint it is important to define these terms in more detail and to create a theoretical framework that describes which properties of components either simplify or complicate specific design tasks.

2.5 Interfaces

An *interface* is a boundary between subsystems. The purpose of an interface is information exchange between subsystems. A component interacts with its environment via linking interfaces (LIFs) [15]. The environment may either be other components or sensors and actuators that are connected to the component.

For a constructive system integration, the properties of the interfaces determine the complexity of the integration process [16]. The integration process is a part of the system design. Usually, large systems are constructively built from a number of subsystems. The subsystem design can either be part of the system design or the designers can use existing components.

2.6 Design Tasks

To be able to reason about the complexity of a design task it is important to have a process model of the way the task is performed. Usually, a design task, such as "develop a brake-by-wire" system consists of a number of subtasks. For each of the subtasks, the complexity analysis must be performed. The subtasks can be seen as some natural segmentation of the larger design tasks.

It is important to identify the most complex subtask as it is not the number of tasks but the cognitive resources required by a single task, that influences the design task complexity.

A design task is *well-defined* if it is clear how exactly the task can be performed, i.e., if there is a process model on how to accomplish the task. Well-defined tasks are typical for architectural design approaches, especially if there exist design tools that guide the system developers.

A design task is *ill-defined* if the designer has no clear conceptualization how the task can be done.

It is clear that just well-defined tasks can be analyzed so to make judgments about the task complexity. The goal of any computer system architecture must be to make all tasks well-defined.

A design task must support segmentation and chunking to such a level of complexity that can be handled by the designer. To develop segmentation strategies, it is usually necessary to understand the whole structure.

Architectures can support segmentation and chunking strategies by structuring the system appropriately and guiding the developers by providing strategies, algorithms, and tools.

Unfortunately, design tasks are usually not analyzed according to their cognitive process models. One reason for this might be that, in general, systems are designed according to an ad-hoc fashion or according to design patterns, but rarely according to well-defined architectures. As there is a trend towards architectural system development, design task analysis will become an essential prerequisite for assessments of the cognitive complexity of the design process.

3 Complexity in Computer System Design

Large computer systems, e.g., integrated architectures for automotive and aerospace control applications [11], have a high level of complexity. It is impossible to consider and understand the whole system at once. Thus, the system design process must be modular so that the system can be built from components. Each design step must have a level of complexity that can be handled. According to relational complexity theory the effective complexity must not exceed relations including more than four dimensions.

3.1 Attributive vs. Relational Properties

We can categorize component properties as either attributive or relational. An *attributive property* is self-contained in the sense that it has a meaning if it is considered in the context of the component. A *relational property* only has meaning with respect to some other component, i.e., a relation must be established so that the attribute can be understood.

Ideally, a component should have only minimal relational properties as this minimizes the relational complexity of tasks involving these attributes. Depending on the number of entities that are uniquely related to a relational attribute the arity and thus the relational complexity of the tasks increases.

An example for an attributive property is that a device supports a standard communication interface like USB or IEEE 1394. However, if a data sheet of a device states a serial RS232 interface, the user needs to adjust several relational properties, e.g., baud rate, parity mode, number of stop bits, and type of flow control between the two communication partners, which increases the complexity for setting up a working system.

3.2 Message-Based Communication

When components communicate via messages, there are two possible classes of data semantics [17]: *State information* describes the state of a real-time entity at a particular instant, e.g., the temperature of a vessel. *Event information* describes an event. It contains the difference between the state before the event and the state after the event, e.g., that the position of a valve has changed by 3 degrees.

Messages that contain event information are called *event messages*, state information is transmitted via *state messages*.

As the data used in automotive and avionics control systems usually are simple sensor and effector data values that are used and transformed by the components of the computer system [18], a message based communication seems to be appropriate.

A precise specification of a message-based interface in the time and value domain allows to consider the component interface in isolation. It thus supports a clean conceptual separation at the interface: For the component designer or component integrator it is sufficient to know the interface specification. It is not necessary to think "beyond" the interface as it fully describes the information flow into and out of the component. This is an important characteristic of message-based interfaces regarding complexity management.

3.3 State Messages

As described above, a state message is self-contained. This means that a state message can describe the state of some entity, without any dependence to other messages.

In terms of relational complexity, the task of understanding the state of an entity as described by a single state message represents a unary relation, as the message is self-contained. It requires no knowledge about the history of state changes to understand the state.

3.4 Event Messages

An event message describes an event and does not contain the full state of the entity it describes. It is thus not self-contained and usually needs some integration with the history of past events.

The task of understanding the state of an entity after receiving a single event message represents a binary relation as the message is not self-contained. It requires the integration of the previous state and the state change described by the event message.

Thus, while the transmission of a data stream containing value updates using event messages can save considerable bandwidth, from the perspective of relational complexity this makes understanding tasks more complex.

3.5 Function or Object-Based Interfaces

A subsystem can have a function-based or object-based interface. Such an interface does not communi-

cate via simple messages, but the subsystem provides functions to other components. However, such a component is not self-contained. Each component needs knowledge about other components and the functions supported by the other components. This view of components corresponds to objects in object-oriented programming languages.

For object-based interfaces, the objects usually present their interfaces to a number of potential users. This is some kind of client-server relationship. The server often does not need to know about the clients, but the clients need knowledge about the server to be able to use its interface. Thus, an explicit relation must be established between client and server. The client is not conceptually independent of the server.

Moreover, the direct invocation of methods of other objects that is not coordinated globally may lead to processing and network overloads that must be considered during system design. This is an issue that has not yet been solved satisfactorily for object-oriented approaches. Such systems are, in general, not composable [19].

Despite these drawbacks, the advantage of the object-oriented approach is that data being passed around can be chunked without losing the chunked dimensions as those dimensions are still available by querying the object that represents the chunk. With a message-based approach chunking can either be "real" chunking, i.e., losing the chunked dimensions, or the dimensions that are not needed can be considered opaque.

4 Segmentation and Chunking

As mentioned in section 2.1, segmentation and conceptual chunking are the primary mechanisms that allow to perform complex tasks. In the following subsections, segmentation and chunking are discussed in the context of computer system design.

4.1 Conceptual Chunking

Chunking means to reduce the dimensionality of the problem space by hiding information that is not needed for the task at hand [1]. Interfaces support chunking by hiding component-internal details and are thus the primary technique used for conceptual chunking, often implicitly.

Ideally, an interface hides as much information as possible for the given task, thus reduces the dimensionality of the task to the lowest possible level. To achieve this reduction, the interface must be designed with the task in mind. If very different tasks must be supported it is thus likely that the component must provide separate interfaces for those tasks so to support all tasks optimally.

4.2 Segmentation

Segmentation means to split a task into a number of subtasks that can be performed serially [1]. For complex problems it is often not obvious how to find appropriate segmentation strategies. Regarding the general understanding task, segmentation on the system level means to decompose the large system into smaller components each of which has some degree of independence from other components so that the segmentation is of any use.

Segmentation is also possible at the interface level of each component. There, it can help by providing appropriately structured information and functionality. If the interface is badly structured or does not exhibit any obvious structure at all, the user first has to find an appropriate segmentation strategy, i.e., split the interface into parts and identify the ones that are needed for the task at hand. Such a search for a segmentation strategy of course takes some time and cognitive effort.

A primary source for concepts that are hard to understand is if the relational complexity is high, i.e., if a component consists of many different aspects that are hard to integrate into a coherent concept.

5 Structuring Systems

To structure a system, the designers usually apply a combination of conceptual chunking and segmentation.

Large systems with a regular substructure are simple to create and maintain, whereas even relatively small systems with no obvious regularities are perceived as being far more complicated. This can be explained—at least in part—by a re-use of chunking and segmentation techniques that can be applied to a large number of subsystems, thereby limiting the total amount of chunking and segmentation techniques required. This is a very strong argument to limit the types of components and their interaction. A computer system architecture that minimizes the types of components is the DECOS integrated architecture [11, 16].

5.1 The System-Level Perspective

The system comprises a number of components that interact with each other to implement some service. The tasks that must be performed at this level, e.g., component integration, usually require an understanding of what the system components do and how they interact to deliver the emerging services. This is especially important if there are multiple developers working at different parts of the system, or if pre-fabricated components are used. Thus, a general overview over the system—the "big picture"—is needed. Also for maintenance, obtaining a general understanding of the system is usually needed if the maintenance is done long after system development and the developers are either no longer familiar with the system or if the maintenance tasks are performed by different people.

For these system-level tasks the need for chunking and segmentation is obvious. In the optimal case, the conceptual structures that result from the segmentation and chunking process are also reflected by the system structure. This means that the conceptual borders between the subsystems coincide with the interfaces between the subsystems. The reason for this is that such a system structure allows for "built-in" chunking and segmentation. The developer trying to get an understanding of the system does not have to develop her own chunking and segmentation strategies but can simply take the system components as chunks. If the components and conceptual chunks that are required to understand the system do not match, this causes increased cognitive effort as both structures must be considered simultaneously and the required underlying concepts must first be detected. The structure of the system is not easily detectable or relies on some abstract or hidden mechanisms which considerably complicates the creation of appropriate chunking and segmentation strategies. Moreover, complexity depends on the number of relational properties of a component, i.e., the degree of concept independence. The next subsection discusses some properties of components that allow for this conceptual separation.

Moreover, each component should represent a meaningful concept. If the system is chunked and segmented into ill-defined concepts, understanding suffers significantly.

5.2 Case Study: A Self-Contained Component

A component interface provides the component service to its users. The characteristics of the component interface determine whether a clean conceptual separation between the components is possible or not. This section discusses the properties of components called *jobs* that are used to compose a distributed application subsystem in the DECOS integrated architecture [11, 20].

The DECOS architecture provides a distributed execution platform for dependable embedded computer systems. DECOS is based on the Time-Triggered Architecture (TTA) [21] and provides strong encapsulation of communication and task execution. The architecture supports the integration of software modules with different criticality levels from different sources (vendors) by providing effective partitioning—not only at architecture level, but even at the level of a single hardware unit (ECU). Thus, a malfunctioning subsystem cannot interfere with other subsystems by monopolizing memory, processing time or communication bandwidth.

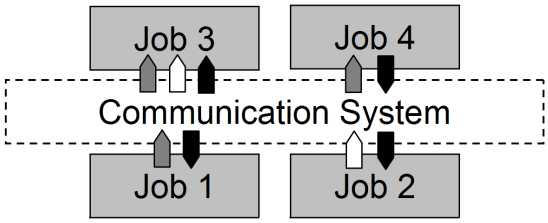

Figure 1: Four jobs communicating via ports

The jobs of a distributed application subsystem implement a specific functionality, such as a steer-by-wire system. A job has a number of ports to send messages to and receive messages from other jobs. For safety-critical applications the messages are time-triggered state messages. A time-triggered communication system is responsible for the reliable message transport [17]. The interface between a job and the communication system is a strict data-sharing interface based on state messages.

Figure 1 shows a simple distributed application subsystem consisting of four jobs sending and receiving various state messages. Job 1 sends a message depicted by the gray port symbol which is received by jobs 3 and 4. Similarly, the ports shaded in other values depict other messages. The communication system decouples the jobs from each other. There is no information or control flow besides the messages.

The self-containedness property of the messages and the autonomous message transport by the communication system allow for a strong conceptual isolation of the component which is a prerequisite for composability [22]. Except for the message content, the communicating jobs do not need any information about each other. For the communication system the messages are just opaque data entities. The communication schedule is derived from the timing requirements of the jobs which are part of the job interface specification.

No complex structural relations are established by this approach. The system developers are limited to simple message-based interactions. This concept has proven to successfully deal even with a large number of components and messages.

Other approaches such as object-oriented systems do not exhibit this strong conceptual isolation of components, but on the other hand support more diverse system structures. Objects may depend on various other objects or can be passed around. In general, they have a far more diverse interface structure, without strict conceptual borders between the objects. Object-oriented systems tend to introduce more relations than the message-based approach described above due to their inherently relational interfaces. Thus, such systems usually have a higher relational complexity for the general understanding task. For a similar effect as the conceptual separation described above, lots of restrictions would have to be imposed.

6 Conclusion and Outlook

For embedded systems, where there are usually severe resource, timing and dependability constraints, an important issue is to manage the complexity that arises by the interactions of the components. Relational complexity theory has been used to describe what is needed to support a general understanding of a computer system. It is important to limit the effective complexity of design tasks to four independent dimensions. For more complex tasks, conceptual chunking and segmentation strategies must be provided. This is possible with an architectural approach that guides the developers accordingly.

Understanding a large system is significantly easier if the conceptual chunks required for understanding match those of the system components. Moreover, it has been shown that the choice of communication mechanism severely influences the complexity of design and maintenance tasks. An approach with state-message based interfaces allows for simple system structuring to manage complexity by minimizing relational component properties and thus limiting the scope of consideration to a single component.

To be able to make statements about the complexity of more specific design and maintenance tasks than for the general understanding task described in this paper, cognitive process models must be developed. These models will heavily depend on the chosen architecture. Moreover, empirical evaluation of the predictions made in this paper will have to be done.

Acknowledgments

This work has been supported by the FIT-IT program of the Austrian Federal Ministry of Transport, Innovation, and Technology, project number 809442 and by the European IST project DECOS, project number IST-511764.

References

[1] Graeme S. Halford, William H. Wilson, and Steven Phillips. Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral and Brain Sciences*, 21:803–831, 1998.

[2] Brian Henderson-Sellers. *Object-oriented metrics : measures of complexity*. Prentice-Hall, Upper Saddle River, New Jersey, 1996.

[3] Rick Kazman and Marcus Burth. Assessing architectural complexity. In *2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 104–112, Florence, Italy, March 1998. IEEE Computer Society.

[4] Jianjun Zhao. On assessing the complexity of software architectures. In *Third International Workshop in Software Architecture*, pages 163–166, Orlando, Florida, 1998. ACM Press.

[5] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1998.

[6] Object Management Group (OMG). OMG Unified Modeling Language Specification. Technical report, OMG, 2003.

[7] Graeme S. Halford, Rosemary Baker, Julie E. McCredden, and John D. Bain. How many variables can humans process? *Psychological Science*, 16(1):70–76, January 2005.

[8] Damian Patrick Birney. *The Measurement of Task Complexity and Cognitive Ability: Relational Complexity in Adult Reasoning*. PhD thesis, School of Psychology, University of Queensland, St. Lucia, Queensland, Australia, March 2002.

[9] Graeme S. Halford, William H. Wilson, and Steven Phillips. Abstraction: Nature, costs, and benefits. *International Journal of Educational Research*, pages 21–35, 1997.

[10] Eberhardt Rechtin. *Systems Architecting: creating and building complex systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[11] Hermann Kopetz, Roman Obermaisser, Phillip Peti, and Neeraj Suri. From a federated to an integrated architecture for dependable real-time embedded systems. Technical report, Vienna University of Technology, Austria; Darmstadt University of Technology, Germany, August 2004.

[12] H. Heinecke et al. Automotive open system architecture—an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures. In *Convergence International Congress*, Detroit, MI, USA, October 2004. Convergence Transportation Electronics Association.

[13] Edward Yourdon and Larry L. Constantine. *Structured Design:Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979.

[14] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[15] Hermann Kopetz and Neeraj Suri. Compositional design of rt systems: A conceptual basis for specificatiom of linking interfaces. *6th IEEE International Symposium on Object-Oriented Real-Time Computing (ISORC'03)*, May 2003.

[16] Bernhard Rumpler. Complexity management for composable real-time systems. In *Proceedings of the 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing*, Gyeongju, Korea, April 2006. IEEE.

[17] Herman Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, volume 395 of *Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, 1997.

[18] Doug Lea. Design patterns for avionics control systems. Technical report, SUNY Oswego & NY CASE Center, DSSA Adage Project ADAGE-OSW-94-01, 1994.

[19] Thomas Losert. *Extending CORBA for Hard Real-Time Systems*. PhD thesis, Vienna University of Technology, May 2005.

[20] Phillip Peti, Roman Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 2–13, Seattle, Washington, USA, May 2005.

[21] Hermann Kopetz and Günther Bauer. The time-triggered architecture. In *Proceedings of the IEEE*, volume 91, pages 112–126, January 2003.

[22] Hermann Kopetz and Roman Obermaisser. Temporal composability. *Computing & Control Engineering Journal*, pages 156–162, August 2002.