

## Chapter 2

# Designing Reconfigurable Systems: Methodology and Guidelines

Zoltan Papp, Raul del Toro Matamoros, Coen van Leeuwen,  
Julio de Oliveira Filho, Andrei Pruteanu and Přemysl Šůcha

**Abstract** One of the major challenges when designing software for complex systems relates to a lack of a specific and comprehensive set of rules and methodologies. Even more so, adaptation to field conditions is difficult to model and implement on systems composed of a larger number of devices/components, such as distributed systems or systems of systems. On state-of-the-art technology such as wireless sensor/actuator networks and cyber-physical systems, addressing the lack of a compressive set of rules for their design and realization offers considerable benefits. If successfully realized, it can accelerate and simplify their design and implementation. The main contribution of this chapter is a clear set of rules that are specific for the design of adaptive networked embedded systems. To be more specific, we discuss design-time vs. runtime trade-offs, introduce design patterns for reconfigurable real-time monitoring and control, propose techniques for runtime design space exploration (managing runtime reconfiguration) and a systems engineering process for runtime reconfigurable systems. We provide guidelines for all stages of the architectural process and help system and software designers in choosing wisely specific algorithms and techniques. In conclusion, this chapter introduces a set of rules (methodologies) that are specific for designing adaptive networked embedded systems.

---

Z. Papp · C. van Leeuwen · J. de Oliveira Filho (✉)  
TNO, Oude Waalsdorperweg 63, The Hague, The Netherlands  
e-mail: julio.deoliveirafilho@tno.nl

C. van Leeuwen  
e-mail: coen.vanleeuwen@tno.nl

R. del Toro Matamoros  
UPM, Madrid, Spain  
e-mail: raul.deltoro@car.upm-csic.es

A. Pruteanu  
TU Delft, Postbus 5, Delft, The Netherlands  
e-mail: a.s.pruteanu@tudelft.nl

P. Šůcha  
Czech Technical University in Prague, Technická 2, 166 27 Prague 6, Czech Republic  
e-mail: suchap@fel.cvut.cz

## 2.1 Introduction: Why Design for Runtime Reconfiguration?

System reconfiguration adapts the system to changes, and as such is carried out for satisfying a predefined goal. Finding the suitable system configuration, while maintaining all design and execution constraints is a demanding and knowledge intensive process. In essence, the runtime reconfiguration is carrying out system design activities during the operation of the system. Consequently all steps of the reconfiguration cycle assumes domain insight and generic engineering knowledge.

The reconfiguration schemes in DEMANES takes into consideration the particular challenges of the DEMANES application domains. These special challenges are:

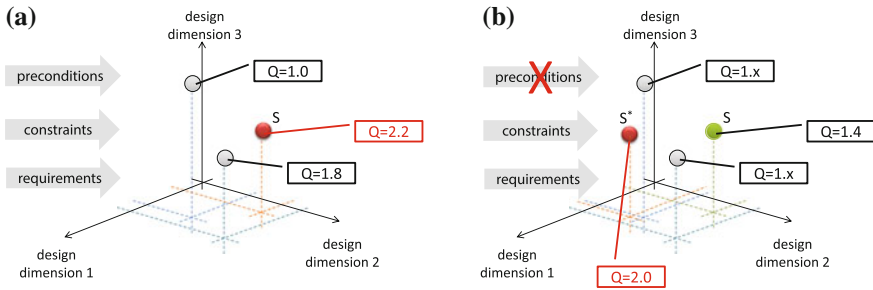
- The primary functionality of the system should be kept in operation. For that, we advocate a separation-of-concerns approach where the primary datapath of the application and the reconfiguration functionality are clearly separated.
- The reconfiguration should be carried out under time constraints. It does not have to be necessarily fast, but it have to respect the response times that are expected to the system application.
- The typical underlying hardware is resource limited: computing capabilities, memory, communication capabilities, energy supply are severely constrained. Accordingly, related architectural and algorithmic challenges should be addressed.

### 2.1.1 *Reasons for Reconfiguration*

During the design process, the designer faces a number of questions about design alternatives and parameterization of the design. Implicitly or explicitly, every question is answered via evaluating alternatives, assigning a measure to them which reflects the “quality” of the alternative, and then selecting the “best” solution.

Except trivial cases the “question–answer” process is very complex: the number of design variables is typically large, there are many possible answers and there are interactions and dependencies among design decisions. The design variables define a design space, in which every design alternative is represented by a point, as depicted in Fig. 2.1. While exploring the design space, possibly complex and large number of requirements design constraints should be satisfied—and these obviously constrain the design space.

Furthermore when answering design questions, the designer makes assumptions, whose validity should be considered as preconditions for the validity and quality of the design. Figure 2.1a represents an (overly simplified) abstract design situation: the designed system is meant to monitor a connected process. There are three design dimensions and the designer have to select a particular combination of these three, that maximize the quality (Q) of the resulting system. All that, while satisfying all requirements and design constraints. Along such design, the designer makes certain



**Fig. 2.1** Design space exploration

assumptions, such as the connected system is governed by linear dynamics; and/or there are some available resources to carry out certain calculations.

As an example, imagine that under the mentioned conditions, the design annotated with  $S$  is the best choice, and thus this combination of the free design parameters should be selected. If, for whatever reasons, the requirements or constraints change, or any of the design assumptions becomes invalid (Fig. 2.1b), then the design  $S$  may become sub-optimal (or even not feasible at all), and consequently the design should be adjusted. It means that the design space exploration should be continued to find a new solution via different selection of the design parameters ( $S^*$ ).

When changes in requirements, constraints, or preconditions may happen during the nominal operation of the system then the design may be done such as to be prepared for the expected changes. For example, designing suboptimal system states that remain valid under the expected changes; or adding an “overlay mechanism” which isolates the system from these changes are perfectly valid options. The problem is that frequently such design time solutions for coping with changes result in complex, over-dimensioned designs—and the designed system remains “fragile” anyhow as it can only handle changes envisioned during the design.

An alternative, and in a number of aspects, more robust solution is implementing reconfiguration capabilities for runtime. In this case the design space exploration and finding the most suitable design is a runtime activity. For that, the explicit representation of the design and some kind of “reasoning mechanism” should be part of the running implementation of the system. The design space exploration, design evaluation, and runtime decision making are resource demanding activities. In order to maintain the resource usage on an acceptable level and still assure temporal guaranties, the design space for the runtime design should be constrained. Thorough analysis (in design time) can be carried out to determine the scope of the runtime reconfiguration—carefully balancing between the resulting robustness and the required resources.

Runtime reconfiguration capabilities are typically used to address the following situations:

**Hardware failure/degradation:** This is typically results in losing resources and assigned functionalities. The challenge is to find a different resource allocation, which can keep (at least) the critical functionalities running.

**Communication failure/degradation:** In distributed embedded systems the quality of the interconnection directly influences the correctness of the operation. Consider for example response deadlines which can be missed due to communication degradation. Changes in communication performance, or loosing the communication completely, may trigger reallocation of functionalities and/or switching to a different processing scheme, which has lower communication demand.

**Energy considerations:** In energy constrained implementations—for example, where certain subsystems run on battery power—assuring the required life-time of the system is of primary importance. If for reasons not accounted at design time, the energy level in some subsystems falls faster than expected, counter-measures can be introduced by, for example, restructuring and/or reallocating functionalities.

**Changing user requirement:** The life-cycle of deeply integrated monitoring and control systems is typically very long and meanwhile changes in user requirements are to be expected. On the other hand, these deeply integrated systems are continuously in use and new capabilities or modification of existing ones should be carried out on-the-fly, typically in a gradual fashion.

**Changing operating conditions:** As mentioned earlier, the system design builds on assumptions about the behavior of the coupled external processes. During operation these preconditions may become invalid and thus the system design should be adjusted to accommodate the situation.

**Mobility, configuration changes:** In certain application, the interconnection topology tends to change due to component mobility. As the interconnection topology has critical impact on the operation of functionalities implemented on the distributed platform, changes in topology should be managed. One alternative is to add functionalities and redundancy to maintain static topology, but often this leads to an overly expensive and even unfeasible solution. Instead, runtime reconfiguration capabilities may change processing architecture to “follow” the topological changes and assure functional robustness.

This list is not meant to be exhaustive—instead it just indicates a few typical uses. It always should be kept in mind that managing reconfiguration is a resource demanding activity in itself. Building in runtime reconfiguration capabilities should always be justified. Consequently thorough analysis should be carried out to determine the scope of the reconfiguration—and sometimes solutions derived at the design phase may be the best approach. In the following, we discuss a bit more the trade-off between taking decisions at design time or at runtime.

## 2.2 The Design Time Versus Runtime Optimization Trade-Off

The main advantage of reconfiguration is ability to react to new situations at runtime. It provides flexibility to the system and it prepares it to situations that were not considered at design time. However, the reconfiguration itself has to often solve

computationally complex problems, i.e. problems that cannot be optimally solved in polynomial time. Take for example the runtime re-mapping of tasks to the physical units described in Sect. 1.4.2. In practical applications, it is solved using heuristic methods in order to be able to react reasonably fast. By using heuristics, there is no guarantee how far from the optimal state the system will be after a reconfiguration is applied. That shows that the flexibility gained through reconfiguration often comes at cost of using a sub-optimal state.

Another aspect contributes to this sub-optimality. The essence of distributed systems is decentralization: it means that parameters and information about particular nodes is often spread out in the entire network and there is no a centralized authority having all the information necessary to perform the reconfiguration. Without a global view on the system state, it is often impossible to make decisions that lead the system to a global optimal state.

Despite the fact that the above described problems cannot be solved at the runtime efficiently, there are cases when design time optimization techniques can make the reconfiguration more efficient. There are no doubts that if something can be decided at the design time (with at least the same quality/efficiency as at the runtime) it should be decided there and it should not increase complexity of the reconfiguration. Moreover, at the design time there is more time to pre-compute, for example, several design patterns that can be applied at the runtime. The design time optimization cannot be generalized since it is a problem specific operation. However, we can categorize it into three classes:

**Initial system state design:** In applications such as wireless sensor networks, the reconfiguration cannot be powerful enough to guarantee optimal or near optimal system behaviour. The aim of the reconfiguration is often to keep the system alive. However, due to decentralized nature of the system or lack of time to perform some more sophisticated optimization, the key performance indicators can slowly get worse and worse. Even though this issue is very difficult to solve, the situation can be improved by a better initial solution. This solution can be found at the design time where a more complicated algorithm may be used to solve the problem. Consequently, the system at runtime starts with much better key performance indicators which may lead to better system behavior and also to its higher reliability.

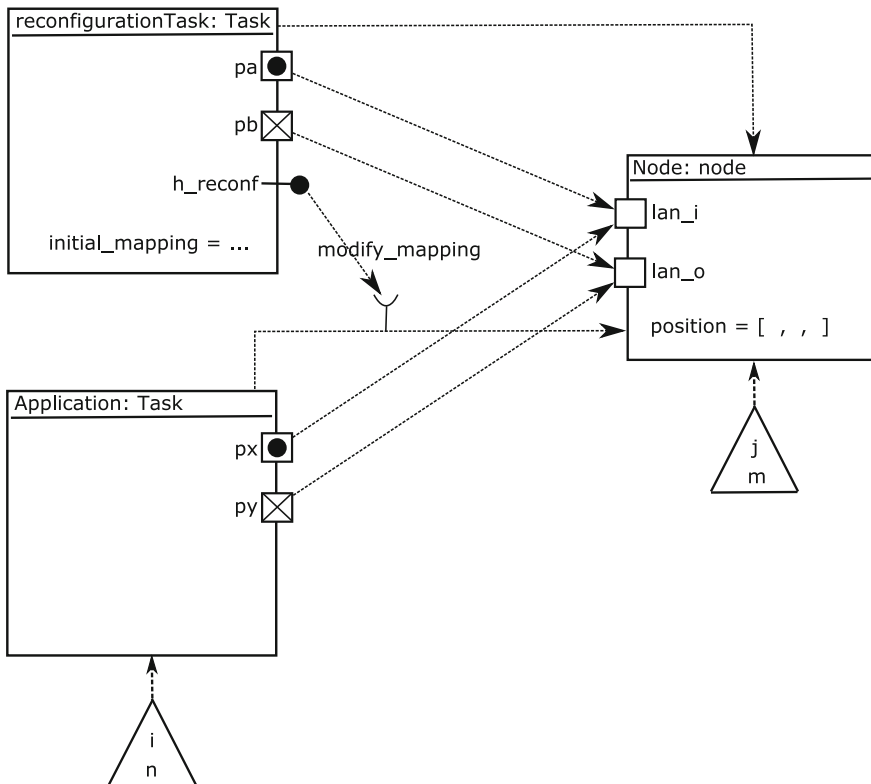
**System design optimization aimed at the reconfiguration:** In many cases there is no possibility to carry over complexity of the reconfiguration from the runtime to the design time. However, it may be possible to optimize properties of the system in order to make the reconfiguration more efficient, for example by determining the optimal number of nodes that guarantee sufficient reliability of the system.

**Decomposition based design time optimization:** Some applications allow decomposing the reconfiguration into decision that must be made at the runtime and the decisions that can be made at the design time. Typical examples are systems operating in a finite number of modes, for example, high/low battery capacity. Then the reconfiguration decides which mode will be selected at the given moment. In such a case the system designer can pre-compute patterns, for

example a schedule of activities, to be applied in the system according the active mode. By doing so a part of the decision making is moved into the design time and can simplify the whole reconfiguration mechanism.

A good example of the first category is the runtime re-mapping of tasks to the nodes described in Sect. 1.4.2. The corresponding model is illustrated in Fig. 2.2. Task called ‘reconfigurationTask’ can stop a task (‘Application’) in one node (‘Node’) and start it in another one. This task mapping is controlled via ‘h\_reconf’ signal at the runtime. However, the initial mapping is computed at the design time and it is stored in ‘reconfigurationTask’ in property ‘initial\_mapping’.

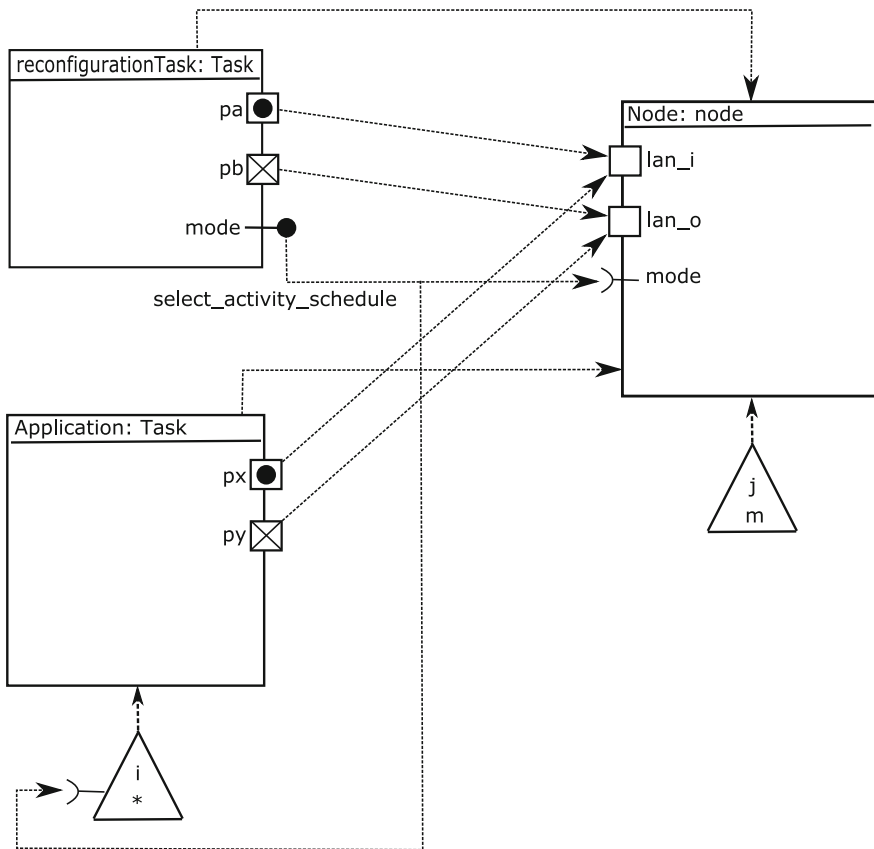
In the first case, the objective of the design time optimization is the same as the objective of the reconfiguration. In the second category, the function at the design time optimization is different. Considering the same example of runtime re-mapping of tasks illustrated in Fig. 2.2 the system designer may be interested in such a physical network design that minimizes the number of needed nodes ( $m$ ). Another aim of the design time optimization would be to determinate position of nodes (property



**Fig. 2.2** Design time optimization

‘position’) such that each node has enough neighbors that can take over his tasks if needed. In this way we can guarantee certain reliability of the reconfiguration.

To illustrate the last category lets consider a slightly different scenario. The model in Fig. 2.3 considers tasks to be executed on nodes. In this case, the mapping between nodes and tasks is fixed. However, we consider two modes of the network, e.g. high battery and low battery state. If the network operates in high battery mode the network performs all the tasks while in the second one some tasks are deactivated. This is controlled by ‘reconfigurationTask’ via the replicator of ‘Application’ Task. For different modes there are different schedules of tasks stored in property ‘schedule’ in each Node. Since the task mapping is fixed the problem can be decomposed into two phases. The decision made at the runtime is responsible for selection of the network mode and the design time optimization assigns to each mode and each node a fixed schedule of activities.



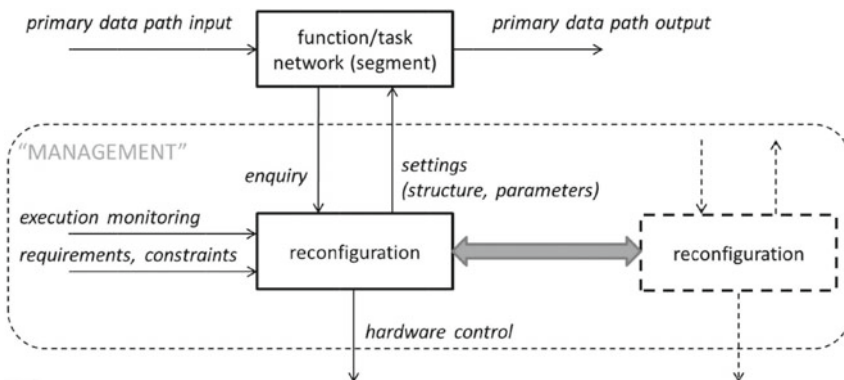
**Fig. 2.3** Design time optimization based on a decomposition

### 2.3 Design Patterns for Reconfigurable Real-Time Monitoring and Control

The main purpose a system is to implement a certain functionality, which transform the input data stream (set) into an output data stream (set) according to the requirement specification (covering both functional and non-functional requirements). This functionality is called primary functionality (PF, for short). The reconfiguration—carried out as a response (counter-measure) to the influences listed above—makes changes in the PF and/or in the hardware configuration and/or in the assignment of the PF components to resources (Fig. 2.4). The separation of PF and the RF is the key for resource aware, time constrained runtime reconfiguration.

Following this separation-of-concerns approach, one of the most used reconfiguration “meta-model” is the MAPE-K, which is shown in Fig. 2.5. According to this approach the reconfiguration is carried out via a monitor–analyze–plan–execute cycle. The monitor activity derives the symptoms from the raw observations. The analysis step builds situational awareness and initiates the generation of the new system configuration. The planning step determines the sequence of actions, by which the current configuration is transformed into the new one. The last step is the timely execution of the plan. It should be emphasized that in this scheme the “Sensors” element in the figure collects information about the system target to reconfiguration (and typically they have nothing to do with the sensors applied in data acquisition for the main function of the system). The same applies to the “Effectors”: the effectors are the means of making changes in the system configuration.

The reconfiguration process (i.e. the MAPE-K cycle) results in a sequence of reconfiguration actions. Such reconfiguration actions can be categorized as follows:



**Fig. 2.4** The separation of PF and RF



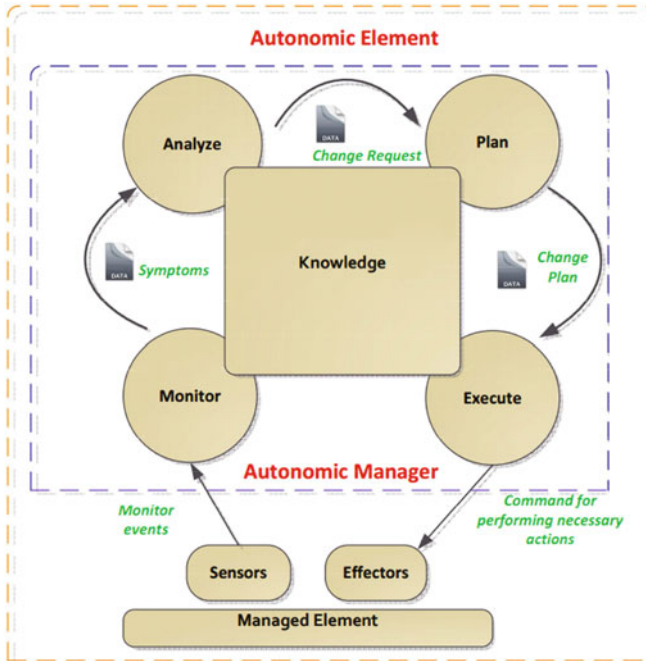


Fig. 2.5 The MAPE-K concept

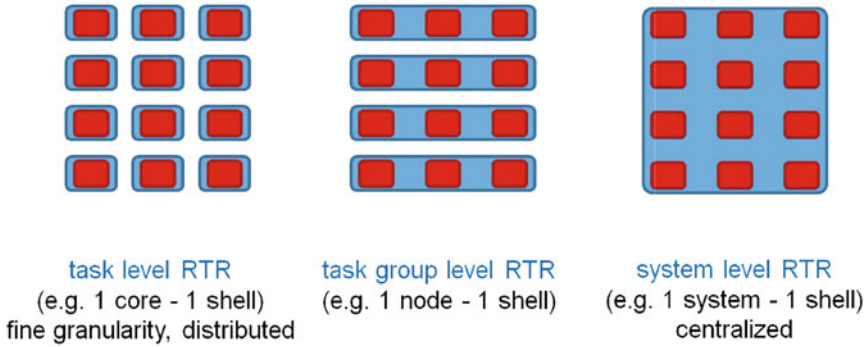
**Parameterization:** During parameterization actions, the processing scheme remains the same except that only certain parameters of certain elements of the scheme are changed. For example, filter or protocol parameters. Conceptually this is the simplest form of reconfiguration.

**Selecting algorithmic alternatives:** Also here, the processing scheme remains the same. But certain elements are replaced by different realizations. The different realization has different non-functional properties, such as computation demand, information need, accuracy, robustness, etc.

**Modifying the schedule of execution (of tasks):** The processing scheme remains the same; indirectly the schedule influences the computing and communication demands, responsiveness, accuracy, priorities, etc., i.e. giving wide range of tuning capabilities.

**Modifying the task allocation (task  $\Rightarrow$  processor assignment):** Resource balancing, dynamic redundancy can be achieved via dynamic task assignment; typically the processing scheme remains the same, but re-parameterization is necessary in most of the cases.

**Modifying the processing structure (task graph):** This is the “deepest” reconfiguration activity; the result is a new processing scheme, which typically requires new task mapping and schedule.



**Fig. 2.6** The granularity of reconfiguration management

Reconfiguration can be carried out on different “granularity levels”. At one extreme, the system binds a RF element to every PF elements. The RF element is responsible for “taking care” of the bound PF element, i.e. making sure that the operational conditions for the PF element is correct and the PF element properly contributes to the system level goals (Fig. 2.6a). This reconfiguration variant typically requires information exchange between RF elements, otherwise system level considerations cannot be handled.

The other extreme is having one single RF, which handles all PF elements. This is a centralized scheme: system level considerations can be handled, but information about the execution state of the PF elements—and their hosts in distributed cases)—should be collected and the actuation commands should be distributed—which may result in high communication demand (Fig. 2.6c).

An intermediate solution is when the RF elements manages a subset of PF elements (Fig. 2.6b). In distributed configurations a typical assignment is the node level reconfiguration management, in which each node hosts one RF element that controls all PF elements assigned to that node. Extra inter-node communication is necessary only to coordinate the actions among the RF elements.

In distributed implementations the “granularity alternatives” should be mapped into the physical configuration, and as such the interaction scheme between the RF is of primary concern. Besides that, the communication topology, the interaction scheme, and the allocation of RF components have direct impact on the performance of runtime reconfiguration. The factors listed also influence the performance of the primary data processing path as PF and RF functionalities compete for the same resources (processing/calculation capabilities, communication, energy, etc.).

The reconfiguration related activities include monitoring of the system state, building situational awareness, determining the new configuration, and carrying out the reconfiguration plan (actuation). At one extreme, all these activities rely on locally available information and actuation capabilities. Under a local scenario, the reconfiguration does not introduce communication inter-node overhead, and only local resource usage has to be considered. Typically, the extra processing needs of the

reconfiguration related tasks. The local RF has a limited insight to the state of the whole distributed system, and thus the scope of the situational awareness and the reasoning about the optimal configuration is limited to the node itself. Only “selfish” decisions can be made about the reconfiguration, i.e. system level objectives typically cannot be achieved.

At the other end of the spectrum, the distributed management of the reconfiguration relies on shared situational awareness and cooperative “reasoning” about the updated configuration. In this case—at least theoretically—the optimal solution can be found<sup>1</sup> In summary, when designing a distributed runtime reconfiguration schemes, the following concerns should be explicitly addressed:

- The costs of building shared situational awareness: complexity of algorithms and communication overhead due to the distributed configuration.
- The costs of building coordination for concerted actions: communication overhead and the quality of the solution. For example, loosing the guarantee for finding the global optimum, relying on greedy or heuristic approaches.
- The costs of execution of RF: communication overhead, complexity of distributed planning, and actuation.

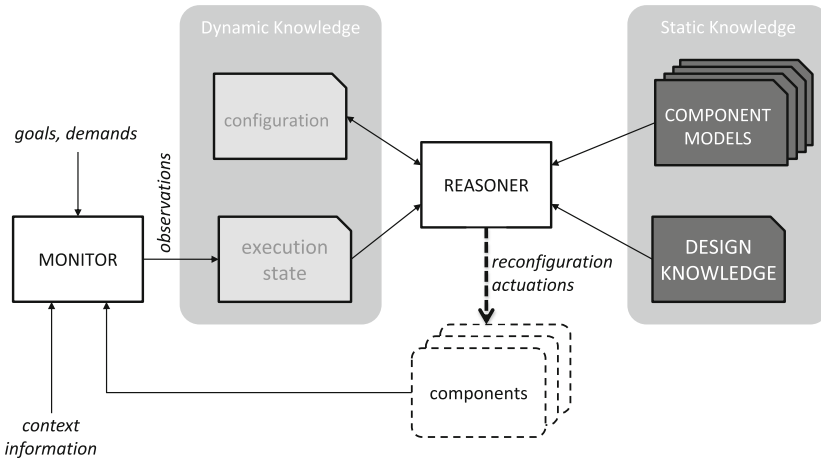
### 2.3.1 Formalizing the Reconfiguration Functionality

The runtime reconfiguration process can be considered as a continuation of the system design process that is performed at runtime. As such it remains a domain knowledge intensive process. Design decisions—such as algorithm selection, parameterization, interconnection topology, task allocation—should be made on various levels based on the actual circumstances, and guided by design expertise, general systems engineering knowledge, etc. Also when dealing with this complexity aspect, the architectural separation of the primary and the reconfiguration functionalities (with clear “actuation interfaces” between them, see Fig. 2.4) results in designs with cleanly assigned responsibilities and manageable non-functional properties.

The implementation of the reconfiguration functionality can take various forms. A frequent implementation pattern is to develop a custom algorithm addressing the particular case in hand. In this case, the “design knowledge” is hard-coded into the implementation. Such approach is frequently chosen for its runtime performance, but—due to the eventual complexity of the reconfiguration challenge—the solution can be error prone. In addition, extending this built in “design knowledge” can be an overly demanding work because it may require thorough rewrite and extension of the existing custom code. An more efficient and economic alternative is to follow

---

<sup>1</sup>Decomposing the design space exploration and the associated search/optimization into distributed configuration, that is a “divide and conquer” approach. Such sub-problems are significantly less complex than the whole, but interaction constraints should also be concerned and they are a challenge by themselves. See more details in the next section.



**Fig. 2.7** A case independent reconfiguration scheme

the “knowledge based approach” pattern, in which the “knowledge” and the “use of the knowledge” are clearly separated entities.

Figure 2.7 shows a reconfiguration solution inspired by “knowledge based” pattern. As the name suggests the “knowledge based” pattern explicitly represents and uses knowledge. By knowledge, we mean any formal representation information, which is relevant in the context of making decisions about the using system resources to achieve the pre-set operational goals. In most of the approaches, the knowledge is captured into models that describe certain aspects of the operation of the system. Knowledge elements also describe the current state of the system relevant to reconfiguration: this part is called dynamic knowledge. Two main constituents of the dynamic knowledge are:

**Configuration:** This is the representation of the actual architecture the systems describing the components involved, their parameters, interconnections, etc. Differently stated, the configuration is a “machine friendly” representation of the design documents. It should be emphasized that

- the configuration does not necessarily describes the full design (which can be overwhelmingly complex) but only those parts and aspects, which are target of reconfiguration (see also the “runtime—design-time tradeoff” considerations);
- the representation used should be “mutable”, as the configuration description should be updated in order to reflect the changes introduced by the reconfiguration process

**Execution state:** The execution state reflects the conditions of the operation, which may influence the satisfaction of the functional and non-functional requirements set against the system. It is assumed that these conditions can change during the lifetime of the systems, consequently they can trigger reconfiguration actions.

The generic design knowledge is “coded” into the static knowledge part of the knowledge base. The static objective refers to the fact that this part does not change during the operation of the system.<sup>2</sup> Using a metaphor, the static knowledge is a set of textbooks a designer can use to acquire design knowledge.<sup>3</sup>

The main component of the reconfiguration scheme of Fig. 2.7 is the reasoner process, which “combines” these knowledge sources in order to satisfy the goals and demands under the given (current) circumstances. From the reasoner point of view the static knowledge is given, the execution state is derived from external sources—thus its scope of influence is making changes in the configuration. When the reasoner finds a new configuration, which satisfies all requirements, this configuration can be implemented, i.e. the changes are introduced in the physical configuration of the system by the reconfiguration actions. The reconfiguration loop is closed then via the monitoring of the system and updating the execution state information.

In this section, we use the “knowledge representation” and “reasoning” in a very wide sense: wide spectrum of representation and associated precessing mechanisms (reasoning) are available. There is no “ultimate solution”. The selection of those should be carried out by thoroughly analyzing of the properties and needs of the problem in hand.

A few alternatives:

- Constraint satisfaction
- Optimization
- Search (explicit design space exploration)
- Feedback control (control theoretical formalization)
- Pattern matching
- Mathematical logic (theorem proving)

### 2.3.2 Task Models for Runtime Reconfiguration

Task models is a design pattern for reconfiguration extensively used in the DEMANES project. The idea behind task models is that the behavior of a (distributed) embedded system should be formalized as a set of interacting tasks. Consequently, the processing scheme implementing the runtime reconfiguration should be mapped into a task model. Each reconfiguration scheme define roles for tasks and fixes where they

---

<sup>2</sup>In this chapter, we do not consider learning systems. Learning systems are a subset of adaptive systems, which are capable of improving their knowledge during operation based on experience or user feedback. For example they can introduce new design rules based on frequently occurring cases or modify design rules if their application creates unwanted behavior. In the design scheme introduced the learning would be manifested by changes in the “static knowledge” part, i.e. strictly speaking it could not be considered static anymore. Still, the mechanism, which would introduce changes is outside of the reconfiguration mechanism considered here, i.e. from this viewpoint the knowledge is read-only and conceptually static.

<sup>3</sup>Due to the non-learning scenario this knowledge remains constant, i.e. the extra knowledge a human designer may acquire via experience is not stored.

should be located. Here we consider only the most commonly used configurations, namely:

- Local monitoring, local reasoning, local actuation (LLL type reconfiguration)
- Full scale monitoring, local reasoning, full scale actuation (FLF type reconfiguration)
- Constrained monitoring, local reasoning, local actuation (CLL type reconfiguration)
- Constrained monitoring, constrained reasoning, local actuation (CCL type reconfiguration)

The interpretation of the terms is as follows:

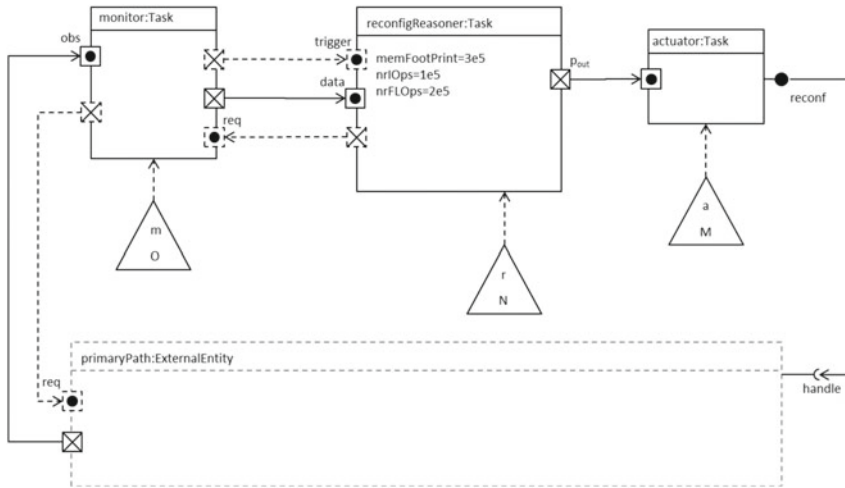
- Local: the scope of the activity is restricted to the node hosting the PF targeted by the reconfiguration (the PF mentioned here may only be a part of the complete PF)
- Constrained: the scope of the activity is restricted to a subset of nodes hosting the PF
- Full: the scope of the activity covers the total system (i.e. all nodes are involved)

The LLL type of reconfiguration is the simplest scheme: every node monitors its own execution state, reasons locally about local goals and the reconfiguration actions are restricted to the node itself. The FLF type of reconfiguration corresponds to the centralized implementation of the reconfiguration: one assigned node collects all execution state information from all nodes comprising the system, carries out the reasoning (locally) and actuates components on all nodes. The CLL differs from LLL in that the local reasoning uses information about the execution state of a subset of nodes, typically neighboring nodes. The CCL scheme relies on cooperative reasoning mechanisms: besides sharing execution states the reasoners cooperate during the reasoning process to establish consensus, i.e. they attempt to achieve system-wide optimality instead of selfish local optimality.

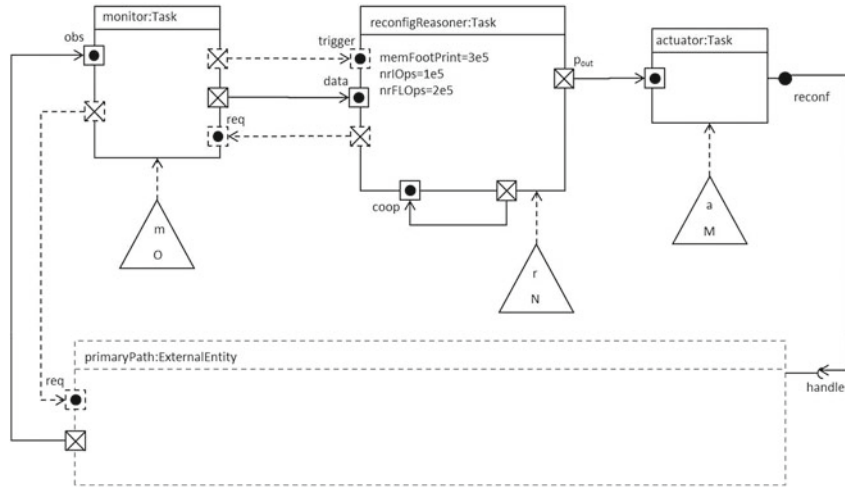
The task models are very similar visually for all reconfiguration schemes, be it LLL, FLF, CLL, or CCL. The difference occurs in three points: the cardinality of the components, the connectivity between the reasoner tasks, and the mapping of the tasks to physical components. Expanding the task model and applying the mapping yields very different operation schemes.

Reconfiguration schemes using non-cooperative reasoning are common when using a local reasoning strategy (e.g. LLL, FLF, and CLL). Their task graph is shown in Fig. 2.8. In these cases, it is usual to have only one reasoner (indicated as `reconfigReasoner`) even when monitors and actuators are distributed in different nodes. The use of many reasoner tasks sharing monitors and actuators may happen. However, different reasoners do not exchange information and thus they do not cooperate. This can be noted in the task model diagram by the absence of communication links between reasoner tasks.

Reconfiguration schemes using cooperative reasoning are mostly common when using a distributed reasoning strategy (e.g. CCL). Their task graph is shown in Fig. 2.9. In these cases, there exist always more than one reasoner in the system



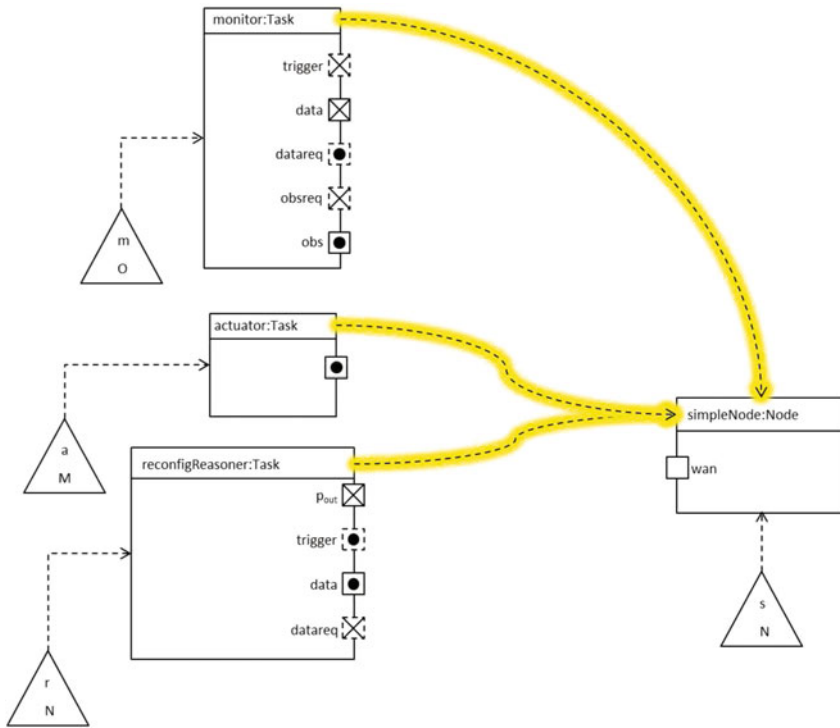
**Fig. 2.8** Non-cooperative reasoning for reconfiguration



**Fig. 2.9** Cooperative reasoning for reconfiguration

and they do communicate with each other. They may or may not share monitors and actuators, but decisions are taken with consideration of the direct information exchanged between them. In the task model diagram, this direct communication is denoted by the presence of communication links (self-loop in the reconfigReasoner block) between reasoner tasks.

Cooperative and non-cooperative reasoning can be used in all reconfiguration schemes discussed before. The differentiation between reconfiguration schemes is determined by the way the task to node mappings happen.



**Fig. 2.10** Task mapping for the LLL reconfiguration scheme

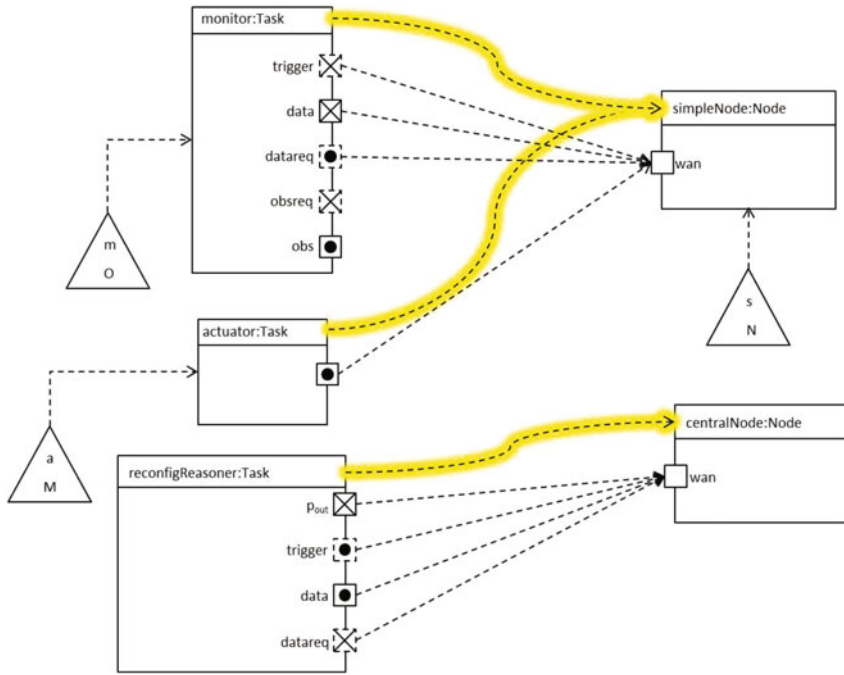
### 2.3.2.1 The LLL Reconfiguration Scheme

When the LLL reconfiguration scheme is applied, reasoners that are mapped to one given node use only monitors and actuators that are mapped to this same node. That implies reasoning is made based on local information, and the scope of the reconfiguration action is restricted to the node itself. Most usually, only one reasoner is used in this scheme, but this restriction is not mandatory. When many reasoners are used in a cooperative way, one extra condition must hold to characterize an LLL scheme: reasoners mapped to one given node can communicate only with other reasoners mapped to this same node. That maintains the reasoning information local within the node scope. The task-node mapping is shown in Fig. 2.10. The filtering notation in the map guarantees the necessary conditions to create an LLL scheme.

### 2.3.2.2 The FLF (Centralized) Reconfiguration Scheme

The FLF reconfiguration scheme implies that information is collected by monitors spread all over the system and transferred to one single point, where the reasoning





**Fig. 2.11** Task mapping for the FLF reconfiguration scheme

takes place. Then, decisions taken by the reasoner(s) may trigger actuators placed in any part of the system. Therefore this scheme is usually interpreted as a centralized reasoning with global view and global actuation. When this scheme is applied, all the reasoners (normally one) are mapped to one single node, whereas monitors and actuators can be mapped to any other node. That implies reasoning is made based on global information (collected wherever necessary), and the scope of the reconfiguration action is also global (any node of the system). Most usually, only one reasoner is used in this scheme, but this restriction is not mandatory. That maintains the reasoning information local within the node scope. The task-node mapping for centralized schemes is shown in Fig. 2.11. The filtering notation in the map guarantees the necessary conditions to create an FLF scheme.

### 2.3.2.3 The CLL Reconfiguration Scheme

When the CLL reconfiguration scheme is applied, a reasoner that is mapped to one given node must only use actuators that are mapped to the same node—that implies their actuation scope remains local. However, such reasoner will use (at least one) monitors that are not mapped to the same node—that is, they will collect non-local information. This is typically the case when the reasoner has visibility to non-local

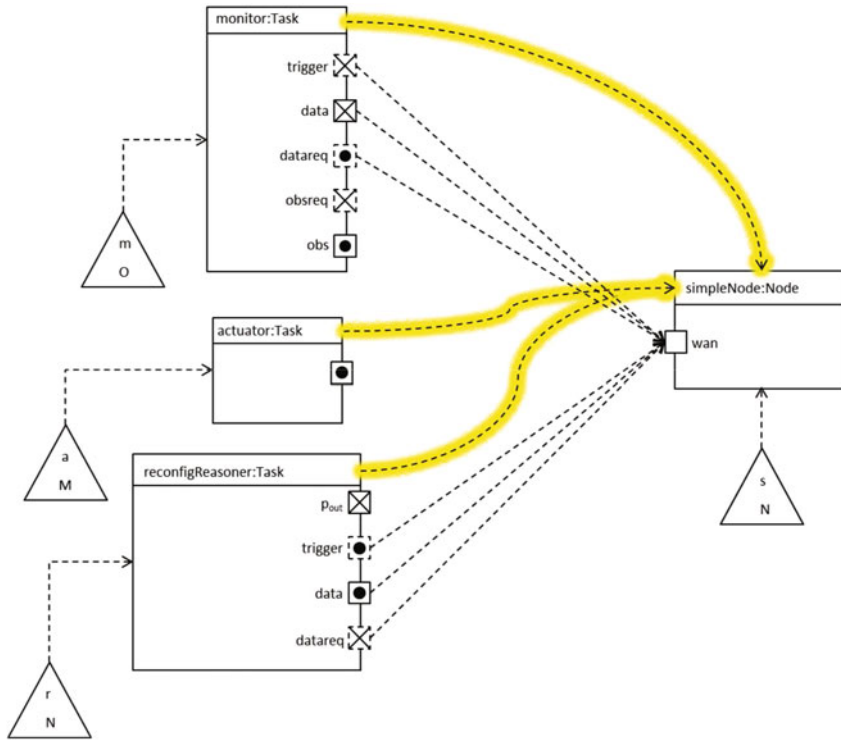
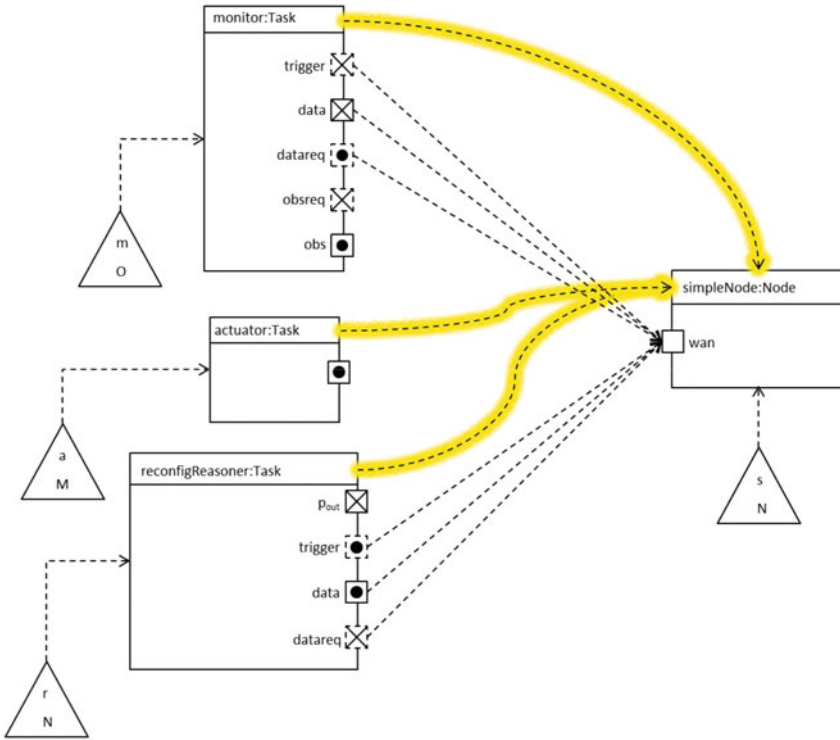


Fig. 2.12 Task mapping for the CCL reconfiguration scheme

information, but can actuate locally only. In other words, reasoning is made based on partial system information, but the reconfiguration action is restricted to a local scope. Most usually, only one reasoner is used in this scheme, but this restriction is not mandatory. When many reasoners are used in a cooperative way, one extra condition must hold to characterize the CLL scheme: reasoners mapped to one given node can communicate only with other reasoners mapped to this same node. That maintains the reasoning information local within the node scope. The task-node mapping is shown in Fig. 2.12. The filtering notation in the map guarantees the necessary conditions to create a CLL scheme.

### 2.3.2.4 The CCL Reconfiguration Scheme

When the CCL reconfiguration scheme is applied, a reasoner that is mapped to one given node must only use actuators that are mapped to this same node—that implies their actuation scope remains local. However, such reasoner will use (at least one) monitors that are not mapped to the same node—that is, they will collect non-local information. Also, they will exchange information with other (at least one) reasoners.



**Fig. 2.13** Task mapping for the CCL reconfiguration scheme

This is typically the case when the reasoner has visibility to non-local information, cooperate with other non-local reasoners when making decisions, but can actuate locally only. In other words, reasoning is made based on partial system information and partial reasoning information, but the reconfiguration action is restricted to a local scope. For the CCL scheme, there are always more than one reasoner task. The task-node mapping is shown in Fig. 2.13. The filtering notation in the map guarantees the necessary conditions to create a CLL scheme.

## 2.4 Design Space Exploration for Runtime Reconfiguration

Design space exploration (DSE) is the process of searching through different system design alternatives. The aim of this process is usually to find some design which outperforms the other alternatives. Ideally one would want to find the optimal system design, but this is not always feasible. The power to operate on the space of potential design candidates renders DSE useful for many engineering tasks, including rapid

prototyping, optimization, and system integration. The main challenge in DSE arises from the sheer size of the design space that must be explored.

The most challenging problems in design space exploration are managing the solution space size, and using a cost function which accurately describes what the desired performance is of the system.

### ***2.4.1 A Quick Survey on Design Space Exploration and Design Decision Making***

There are several techniques for performing design space exploration, be it at design time or at runtime. In the following, we summarize some of these techniques with example cases where they are applied for the design of embedded systems.

#### **2.4.1.1 Genetic Algorithms**

Palesi and Givargis [1] propose a method for exploring design space for a system-on-a-chip (SoC) architecture using a Genetic Algorithm (GA). They assume that the architecture is dependent on a set of parameters which have a large effect on the performance of the application run on the chip, but is difficult to tune manually. In order to run a GA it is necessary to define three things: A representation of the configuration (gene), an objective function or goodness measure and thirdly a convergence criterion which defines when a solution is found. In this example, and for any reconfiguration task, the represented gene is a vector of the parameters of the system. The function is in this case the power consumption and the time required to perform the task to be implemented. Finally the convergence criterion is a maximum number of iterations or a Pareto-optimal solution.

What a genetic algorithm then will do is iteratively try multiple variations of the configuration, and determine what the utilities (or fitness) of the proposed configurations are [2]. A selection is made from the top scoring configurations, and alterations are made upon these configurations for the next iteration. This method has some very clear links with the Darwinist theory of evolution. The exact method of altering existing configurations may vary from mutations where any parameter changes by a random offset, or crossover where complete sections are copied from other viable solutions. Some variants even allow for invaders which are completely random new competitors added to the pool at each iteration to guarantee a larger coverage in the search space area.

Using genetic algorithms has some advantages, namely that they are generic; little to no knowledge about the problem is required except for the three abovementioned things. Another advantage is that GA algorithms satisfy the anytime algorithm criterion: it will always come up with a solution, but the refinement improves as the algorithm runs for a longer period of time.

Downsides of the GA approach are the fact that it is not using any intelligence to optimize the existing solutions. It in fact quite haphazardly generates and evaluates random variations of existing solutions. Due to this principle it may take a long while before it converges to an optimal solution. It can also be quite computationally expensive to generate and evaluate thousands of possible configurations.

A GA could also be used to just optimize an existing solution provided from a different strategy. If it is true that parameters of the system are not critically dependent, and it is viable to do some runtime exploration of the configuration space, it is possible to run a GA continuously. It could change the parameters of the existing system, and monitor the performance. Noted that this approach would work very slowly, it could provide very nicely adaptive systems.

A similar approach is employed by Liu [3] in which a framework is proposed for evolving distributed algorithms. An algorithm is proposed in which agents at each node can choose between a set of basic transition functions, and the employed method is chosen initially randomly, after which the different agents will self-evaluate and change their function if it no longer meets the requirements. It can change via random mutation or by selecting and implementing the function of the best performing neighbor.

#### 2.4.1.2 Graph Based Methods

Georgas [4] already proposed a method for describing reconfiguration as graph based models. This method called ARCM (Architecture Runtime Configuration Management) is specifically aimed to improve the visibility and understandability of runtime adaptive processes while allowing for human input in the adaptation-control loop.

At the core of their method (or linchpin as the authors put it) is the architectural configuration graph. Each node in this graph is a configuration, and the edges are transitions with a specific set of conditions. Whenever something is changed in the configuration of the system, this is stored in a graphical representation, and the amount of times the configuration is chosen, as well as the amount of time spent in that configuration is stored. Afterwards this graph can be inspected by the user in order to see how the system (re)acted.

Finite state machines have been used to describe reconfiguration by Teich and Köster [5]. They introduce a concept of self-reconfigurable finite state machines where in iterative steps a finite state machine can reconfigure itself by changing at most one transitions/output pair. In order to change a complete chain, some intermediate temporary configurations are required. If each intermediate configuration is considered a city, then the problem of reconfiguration in this case becomes comparable to the traveling salesman problem, and no algorithm can find an optimal way to reconfigure in polynomial time. Therefore the authors use a Genetic Algorithm approach to find a way to solve the reconfiguration problem. It should be noted that in this case the actual application that is reconfigured is a finite state machine, whereas the reconfiguration problem is not a finite state machine.

An example of a reconfiguration problem solved using graph matching methods has been shown by Kuo and Fuchs [6]. In their article the authors use reconfiguration using graph matching for designing large electronic circuits. Specifically the faced challenge is to allocate spare units in a way to optimally replace faulty units. Especially when multiple spare units are candidates to replace a faulty unit, the amount of possible reconfigurations can be very large. In the proposed solution the authors can solve the problem for allocating a spare unit to every faulty node in polynomial time  $O(V^3)$  using the Hungarian Method [7]. This algorithm is still the most efficient for finding an optimal solution for a weighted bipartite graph matching algorithm. For unweighted bipartite graph matching problems there exist more efficient algorithms, most notably the Hopcroft-Karp algorithm which has a worst case complexity of  $O(V^{2.5})$ , but for random graphs it runs in near-linear time.

#### 2.4.1.3 Constraint Based Methods

Kogekar et al. [8] presented an approach for constraint-guided software reconfiguration in sensor networks. They have implemented a system which relies on monitoring the system requirements expressed as formal constraints. Those constraints drive the reconfiguration process that takes place in a base station that can communicate to all the sensor nodes. They subsequently demonstrate their approach using simulation results from a simple one-dimensional tracking problem.

In a related article from Eames [9], a method called DesertFD (Design Space Exploration Tool using Finite Domain constraints) is demonstrated for design space exploration based on constraint satisfaction problems. The authors specifically mention that the tool was initially produced for design-time configuration, but was later embedded in a runtime reconfiguration framework in order to do on-the-fly optimization. They combine design-time derived metadata from profiles, benchmarks and expert knowledge, with runtime information from monitoring instruments. The process is then periodically called, and evaluates and prunes the set of system configurations with the goal of selecting the proper configuration to deploy.

An example of a reconfiguration model based on constraint satisfaction programming is given by Syrjänen [10] who examined of software configuration management. A declarative rule-based formal language is proposed for representing configuration knowledge. A case study is shown for managing Debian GNU/Linux packages. The author states that finding a stable in any program in normal logic is NP-hard. In order to find a solution for the configuration problem a method is used to find stable models semantics for logic programs developed by Simons [11]. This method called *smodels* is an algorithm for solving a constrained design problem, and as the authors denote, it does have some overhead and even possibly more than comparable methods. However this overhead provides the designer with a more powerful language. Consequently, problems that can be more compactly represented by logic programs can be more quickly solved with *smodels* than with a satisfiability checker.

#### 2.4.1.4 Logic Based Methods

A logic based reconfiguration method will rely on some type of logical reasoner to find a configuration that matches requirements in runtime. One example a first-order logic interpreter is Prolog [12]. Prolog uses backtracking methods and Selective Linear Definite clause resolution to find solutions [13, 14]. This way of solving logic statements is proven to be both sound and complete [15]. Backtracking is the process of finding a solution by searching for partial solutions. For each candidate of a partial solution the validity is checked, if at one point no solutions exist for the last partial solution, a different one is searched for the previous one. This is analogous for finding solutions in a search tree, and looking depth-first for a solution that satisfies all sub statements.

Selective Linear Definite clause resolution is a method for checking statements by denoting them as disjunctive literals, and refuting all negated literals via the before mentioned backtracking method. SLD-resolution is very efficient both in terms of time and space. However, similar sub goals may be derived multiple times if the query contains recursive calls. Moreover, SLD-resolution is not guaranteed to always terminate [16].

In the SOSE project [17] a framework is described for reconfiguration at the heart of which lies a Prolog based reasoner. Using a relatively simple set of rules the authors can describe the reconfiguration strategies. The method is demonstrated for a greenhouse use case in which an optimal configuration needs to be employed for estimating the temperature distribution. Whereas in this case the goal of the system was purely state estimation, the framework was set up in a generic manner, so different applications are possible as well.

Isermann [18] already used a fuzzy logic controller for automatic supervision and fault diagnosis. He concluded that fuzzy logic provides a systematic framework to process vague variables and/or vague knowledge. As an example of this type of framework, Aubrun et al. [19] used fuzzy logic for monitoring and reconfiguring a power plant. They concluded that fuzzy logic provides better results for failure detection in terms of the robustness of the models comparing to classical methods. A freely available fuzzy logics reasoner called FuzzyDL [20] uses algorithms that combine a tableaux algorithm and that of a MILP-solver. It extends normal description logic with concepts of real values, integers, strings and definitions of fuzzy membership functions. It also allows for concepts in the knowledge base to be constrained using syntax which is comparable to constraint description. In a related article, the same authors propose a method for finding solutions using fuzzy logics from models described in OWL [21].

#### 2.4.1.5 Fuzzy Logic-Based Decision Making

Fuzzy Logic is a powerful tool or framework for decision making in situations of essential uncertainty in models, information, objectives, restrictions and control actions, by emulating the making decision process of humans.

In the specific field of control systems, it is often the case in processes that control strategy designed according to other patterns (classical, adaptive, etc.), do not provide the desired results or simply fail. This is highly complex processes and yet, in the vast majority of cases, skilled and experienced human operators achieve satisfactory results. In 1965, Zadeh gives birth to the so-called theory of fuzzy sets, which lays the foundation of the so-called linguistic synthesis, showing how can be used vague logical statements to derive inferences (also vague) from imprecise data [22].

### Linguistic Variables

One of the most important concepts in Fuzzy Logic is the linguistic variable [23]. In the classical viewpoint we only deal with conventional variables, and it is necessary for us to understand how variables have to be transformed before they can be handled by fuzzy systems.

The linguistic variable is characterized by a quintuple  $(\chi, T_e(\chi), D, G, M)$  in which  $\chi$  represents the name of the variable,  $T_e(\chi)$  is set of linguistic values (attributes, adjectives) of  $\chi$ ,  $D$  the universe of discourse,  $G$  the syntactic rule to generate  $\chi$  names, and  $M$ : the semantic rule to associate each value with its meaning. Therefore the variable  $\chi$  can be transformed (mapped) into a “linguistic variable  $\chi$ ” and vice-versa.

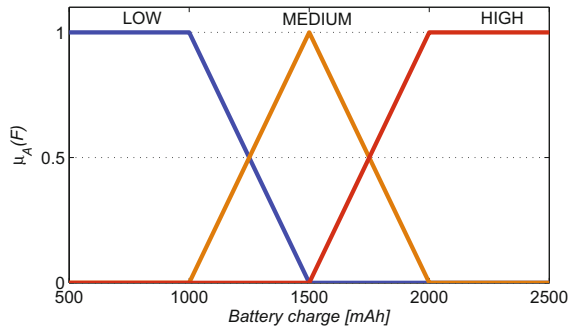
For example, if *battery charge state* is interpreted as a linguistic variable (see Fig. 2.14), then  $T_e$  is the set of linguistic attributes defined for the *battery charge*. The fuzzy partition in the universe of discourse will be performed according to those attributes (*low, medium, high, very high, etc.*). For example:

$$\begin{aligned} T_e(\text{battery charge state}) &= \{\text{low, medium, high, ...}\} \\ D(\text{battery charge state}) &= [500, 2500] \end{aligned}$$

In this case the use of the syntactic rule is not necessary because the attributes follow a natural order. However, in other cases attribute location needs to be arranged, e.g.:

$$T_e(\text{battery charge state}) = \{\text{fairly high, excessively high, very high, ...}\}$$

**Fig. 2.14** Linguistic variable battery charge state





Then the syntactic rule could be:  $G: \dots$ , “very high” at the upper end of  $D$ , and after that “excessively high” and “fairly high”, respectively. The semantic rule  $M$ , for instance, allows us to interpret in our example that the battery charge is classified completely (100%) as “high” above about 2000 mAh, the value at which membership in this set declines linearly down to zero at 1500 mAh.

$$M_{HIGH} = \begin{cases} \mu_{HIGH} = 1, & E \geq 2000 \text{ mAh} \\ \mu_{HIGH} = \frac{E}{500} - 3, & 1500 \text{ mAh} \leq E \leq 2000 \text{ mAh} \\ \mu_{HIGH} = 0, & E \leq 1500 \text{ mAh} \end{cases} \quad (2.1)$$

$$M_{MEDIUM} = \begin{cases} \mu_{MEDIUM} = \frac{E}{500} - 2, & 1000 \text{ mAh} \leq E \leq 1500 \text{ mAh} \\ \mu_{MEDIUM} = 4 - \frac{E}{500}, & 1500 \text{ mAh} \leq E \leq 2000 \text{ mAh} \\ \mu_{MEDIUM} = 0, & E \leq 1500 \text{ mAh}; E \geq 2000 \text{ mAh} \end{cases} \quad (2.2)$$

$$M_{LOW} = \begin{cases} \mu_{LOW} = 1, & E \leq 1000 \text{ mAh} \\ \mu_{LOW} = 3 - \frac{E}{500}, & 1000 \text{ mAh} \leq E \leq 1500 \text{ mAh} \\ \mu_{LOW} = 0, & E \geq 1500 \text{ mAh} \end{cases} \quad (2.3)$$

### Fuzzy Logic Device (FLD)

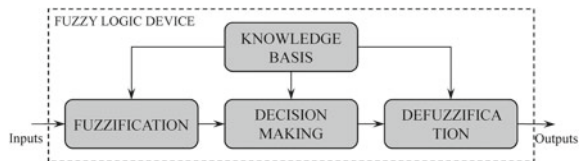
Zadehs conclusions suggested using a fuzzy rule-based (human reasoning-based) approach to the analysis of complex systems and provided a decision-making procedure together with a mathematical tool [24]. In general fuzzy systems are knowledge-based systems that can be built up from expert operator criteria and can be considered universal approximators where input/output mapping is deterministic, time invariant and nonlinear [25, 26].

The fuzzy logic device (FLD) is a general concept in which a deterministic output (crisp values) is the result of the mapping of deterministic inputs, starting from a set of rules relating linguistic variables to one another using fuzzy logic. For the mapping to be performed, deterministic values are converted into fuzzy values, and vice-versa. A FLD is made up of four functional blocks: fuzzification, the knowledge base, decision-making and defuzzification.

The classic configuration universally accepted as representing the four functional blocks described above is shown in Fig. 2.15. It is important to understand that these blocks have a functional meaning, so it is not necessary to separate them algorithmically or physically, as might be concluded from the diagram.

Regarding its input-output mapping, a Fuzzy Logic Device (FLD) has severely nonlinear input/output characteristics. The static characteristic of a typical two-input ( $x_1, x_2$ ), one-output ( $y$ ) FLD is shown in Fig. 2.16. The shape of the surface (in general

**Fig. 2.15** Block diagram of a FLD



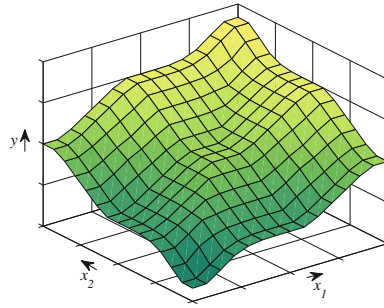


Fig. 2.16 Static characteristic of a two-input/one-output FLD

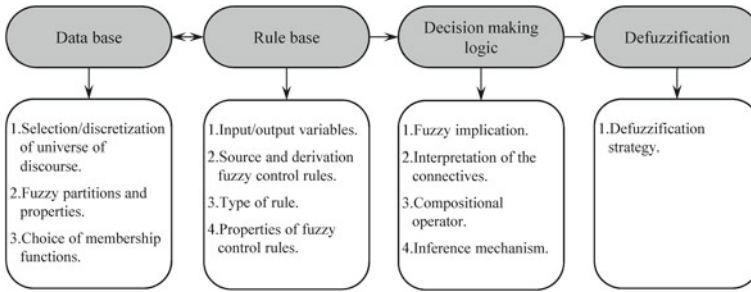


Fig. 2.17 Main FLS design parameters

a hypersurface) depends on the knowledge base. There are many parameters involved in designing an FLD, and they determine the shape of this nonlinear hypersurface.

The need to define linguistic variables by means of the FLD’s input/output variables causes there to be a high number of parameters (design parameters) with usable information about the fuzzy partitions. The definition of operations in the decision-making procedure provides additional design data. So, there is enormous flexibility in the design stages, but the drawback is that these parameters have to be properly tuned. The main FLD design parameters are illustrated in Fig. 2.17.

One of the most significant steps is the selection of input/output, membership functions and the fuzzy partition of universes of discourse. The fuzzy partition of input/output spaces is what determines the “granularity” of the FLD. A heuristic procedure is usually applied to find the optimal fuzzy partition, although some algorithms based on neural networks and genetic algorithms are currently available for this purpose as well [27, 28].

However, the main step in FLD design is undoubtedly the derivation of fuzzy control rules. Tong [29] classified the ways for creating rules as *verbalization*, *fuzzification* and *identification*. *Verbalization* technique, assumes that the rule basis is formulated from verbal process descriptions obtained from skilled operators and technologists [30, 31]. *Fuzzification* (not to be confused with the fuzzification stage of the FLD) provides fuzzy decisional models drawn from ordinary (classical)

mathematical expressions and using Zadehs Extension Principle. *Identification* involves the generation of relational descriptions obtained from numerical data gathered during manual process operation performed by expert operators. This method provides the objectivity of actual measurements, after processing large data files containing the information gathered during manual operation. Several computational procedures can be applied to implement *identification*; two good examples are *Fuzzy Clustering* and *Artificial Neural Networks* [32].

#### 2.4.1.6 Rule-Based Ontology Reasoning

There are several mechanisms that can be used to define rules and manage them: to employ a semantic reasoner, a rules engine or a reasoning algorithm.

**Semantic Reasoners** They are also known as reasoning engines, because they use an inference engine to infer or deduce logical consequences from a set of facts or axioms. They are called semantic because they use a semantic language for reasoning or inference (OWL). OWL axioms infer new knowledge through language itself. Apart from infer new knowledge, the reasoners are also used to validate an ontology, i.e., see if it can create the instances that the developer needs to implement.

**Rules Engine** These systems are based on initial information and a set of rules, detects which of these rules should be applied at a given time and what results from its application. Describe the policies, standards, operations, definitions and constraints of a particular environment. These systems are based on rules to infer new knowledge. There are semantic rules engines that used semantic languages as OWL, and rules engines that use no semantic languages.

**Reasoning algorithm** The algorithms are able to predict or infer the behaviour of a user or system, based on taxonomy trees that store past experiences of the individual or system. No used for this purpose any semantic and power mechanism, it is based on the power of the implemented algorithm.

#### *Semantic Reasoner*

A semantic reasoner, reasoning engine, rules engine, or simply a reasoner, is a piece of software able to infer logical consequences from a set of asserted facts or axioms. The notion of a semantic reasoner generalizes that of an inference engine, by providing a richer set of mechanisms to work with. The inference rules are commonly specified by means of an ontology language, and often a description language. Many reasoners use first-order predicate logic to perform reasoning; inference commonly proceeds by forward chaining and backward chaining.

They are also known as reasoning engines, because they use an inference engine to infer or deduce logical consequences from a set of facts or axioms. They are called semantic because they use a semantic language for reasoning or inference (OWL). OWL axioms infer new knowledge through language itself. Apart from infer new

knowledge, the reasoners are also used to validate ontology, i.e., see if it can create the instances that the developer needs to implement.

We can distinguish two types of reasoning:

**Standard Inference** It makes use of languages such as RDFS and OWL. Inference rules embedded in RDFS and OWL are fixed so that reasoners are able to implement inference even without the use of a rules engine, using only their own algorithms.

RDFS and dialects like OWL Lite and OWL DL are based on first-order logic. These languages allow through its expressiveness infer new assertions. To understand an example of reasoning that takes place, it possible to think of the class “Teacher”, which is subclass of “Person”. If the knowledge base contains an instance of Professor  $\pi$ , applying a reasoner against knowledge base and ontology, there would be a new assertion indicating that the instance of Professor  $\pi$  is a Teacher and a Person.

**Rule-based Inference** This type of reasoning requires an inference language for representing rules and a rules engine.

When the expressiveness provided by the ontology specification languages is not sufficient for semantic reasoner, then it can define rules using rule languages as SWRL. In these cases, not only reasoners employ an inference engine but also a set of semantic rules to generate knowledge and make inferences. The responsibilities of each of these elements are as follows:

- The inference engines interpret and evaluate the facts in the knowledge base to provide an answer.
- The semantic rules allow expressing the behaviour of an individual within a domain.

Reasoners allow defining semantic rules using different languages with varying degrees of expressiveness, and the inference engine is responsible of executing those rules in order to obtain new knowledge. It is essential that the rules are well defined in order to generate new knowledge successfully.

### *Ontology Reasoners*

Reasoners can be classified into two groups: semantic reasoners and logic programming reasoners. Description Logic Reasoners perform standard inference through languages like RDFS and OWL. The description logics can represent a knowledge base that describes a particular domain. This domain is represented by classes (concepts), individuals and roles (properties).

Knowledge representation is made through two elements, TBox, which describes intentional knowledge as concepts and definitions of roles and Abox, which uses the ontology terms to declare assertions about individuals.

The descriptive logic reasoners (DL) must be able to provide the following inference services:

- Validation of the consistency of an ontology. The DL reasoner must be able to ensure that an ontology doesn't contain contradictory facts.
- Validation of compliance of the ontology concepts. The DL reasoner must determine whether a class may have instances. If a concept that does not satisfy this principle, the ontology is inconsistent.
- Ontology classification. The DL reasoner computed from the axioms declared in the ontology T-, relationships of subclass between all concepts explicitly declared to build the classes hierarchy.
- Details in the hierarchy concepts. The DL reasoner can infer what the classes which directly belongs.

### *Review of algorithms for runtime rescheduling/mapping*

Dynamic migration of functionality from one device to another device is an important ability of self-adaptive reconfigurable networked embedded systems. There are two reasons why dynamic functionality migration is considered in networked embedded systems. The first one is reliability of these systems, e.g. when a device is added to the system or a device fails all functionalities are preserved. The second one is efficiency of the system expressed via an objective function(s) aimed at, e.g. energy consumption minimization or reliability maximization. Functionality in the systems is described via a set of tasks and functionality migration is realized by dynamic mapping and scheduling of these tasks with respect to actual state of the system. Mapping in this context means assignment of tasks to devices while scheduling means assignment of time slots when the tasks are executed.

The problem of task mapping is mainly addressed in parallel and grid computing area, e.g. [33]. Much less attention is paid to this problem in networked embedded systems. According to the quality indicator of task mapping expressed by the objective function(s) there are three possible approaches to task mapping and scheduling [34]:

**Global approach** There is only one decision maker (task mapper and scheduler) having single objective function.

**Cooperative approach** There are several decision makers (e.g. devices) that cooperate in making the decisions.

**Non-cooperative approach** There are several decision makers but each decision maker optimizes its own objective.

Majority of existing works deal with off-line mapping and scheduling of tasks considering global approach. Authors in [35] deal with task mapping and scheduling problem on networked embedded systems. Moreover, they also involved a control synthesis into the design process. The problem is solved off-line by a genetic algorithm. An ILP (Integer Linear Programming) problem formulation is proposed in [36]. This algorithm considers non-preemptive tasks and pipelining. Moreover, in order to exploit parallelism as much as possible replication of tasks is allowed.

Even less works are dealing with distributed algorithms for tasks scheduling and mapping. A self-organizing sensor network is described in [37]. The authors show a case study illustrating Kalman filtering on a distributed network of embedded systems. A distributed algorithm for on-line task mapping on reconfigurable networked embedded systems is described in [38]. The algorithm is based on diffusion algorithm, first introduced by Cybenko [39]. The disadvantage of the algorithm [38] is that it could generate huge communication traffic in the network. A diffusion algorithm is also used in [40]. Their algorithm is applied to a homogeneous system, i.e. system where capabilities of all devices are equal. Unlike the algorithm in [38] they consider integer granularity of tasks.

## 2.5 A Systems Engineering Process for Runtime Reconfigurable NESs

Developing large-scale real-time systems pose significant challenges and has been a target of intensive research both in academia and industry. These efforts resulted in a number of standards and recommended practice some of them relatively generic [41], others are dedicated to particular application areas [42]. The networked embedded systems domain (with the extra flexibilities, constraints and uncertainties brought in by the networked topology) further increases the complexity of the design. In order to cope with these uncertainties and improve the dependability of the networked embedded systems when deployed for large-scale, difficult to maintain and/or critical scenarios in a resource and cost effective way, the runtime reconfiguration promises a great advantages. Unfortunately the runtime reconfigurability brings in a new dimension for design complexity and well-established tools and methodologies prove inadequate to cope with these challenges [43, 44].

In the previous sections it was shown that model-based engineering (MDE) techniques for system design are becoming even more important for this class of applications—practically the thorough use of MDE techniques is the only reasonable hope to answer these challenges. MDE techniques assure that the designer can make informed decision during the design process and thus can minimize the effort spent on design iterations.

This section focuses on this decision process, i.e. it proposes a “work-flow” to get to a “proper” design in an efficient way. The main concern is addressing the added complexities brought in by the runtime reconfiguration capability. It is assumed that the requirements are well-defined and understood, thus the (far from obvious) requirement analysis and specification stages are not covered here.<sup>4</sup>

---

<sup>4</sup>Due to its complexity and far fetched consequences the requirement analysis and specification are thoroughly studied processes by themselves and long list of underlying processes, methodologies and tools (both commercial and research) have been proposed. Even a overview of the approaches would go beyond the scope of this book. Interested readers are referred to [45] for overview of frequently used approaches.

The programming/implementation stage of the system development only marginally influenced by the runtime reconfiguration: it is mainly the MDE approach, which makes a difference there. The impact of MDE approaches on system implementation is covered elsewhere (e.g. [46]). On the other hand the validation and verification of runtime reconfigurable demand for novel approaches, methods and tools. Because of its significance for the widespread introduction of runtime reconfiguration in practice, validation and verification are targeted in details in a separate chapters (see Part 2).

### 2.5.1 Related Work

In order to increase the efficiency of and reduce the uncertainties in the system development process development of methodologies and supporting tools is a target of intensive research. One of the first extensively published and studied system development family methods relied on the waterfall model. Though the underlying process is clear and logical, these methods fail seriously in larger scale problems. It can handle the requirement and design uncertainties especially inefficiently: as a result the design feedback loops (i.e. making corrections in the design, choosing different alternative, etc.) are unacceptably long, effecting a number of development stages (Fig. 2.18). Still, the waterfall model is a “reference point” and later developments attempted to eliminate some of its shortcomings.

One of the other “mother of development methodologies” is the so called V-model. As far as the distinguished development stages are concerned the V-model uses the same concepts as the waterfall model, actually it can be considered as a “folded waterfall model”. The significant difference is the emphasis on and the structuring of the different levels of validation and verification (Fig. 2.19) The V-model was definitely a step to the right direction, still the system level validation and verification can only be carried out late in the development process. Consequently—though

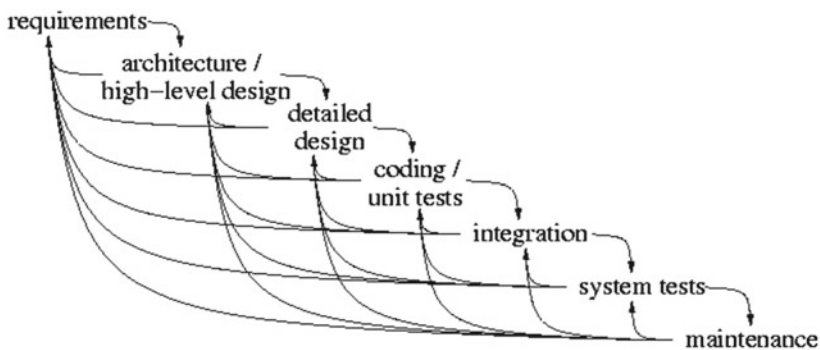


Fig. 2.18 Waterfal development process inreality

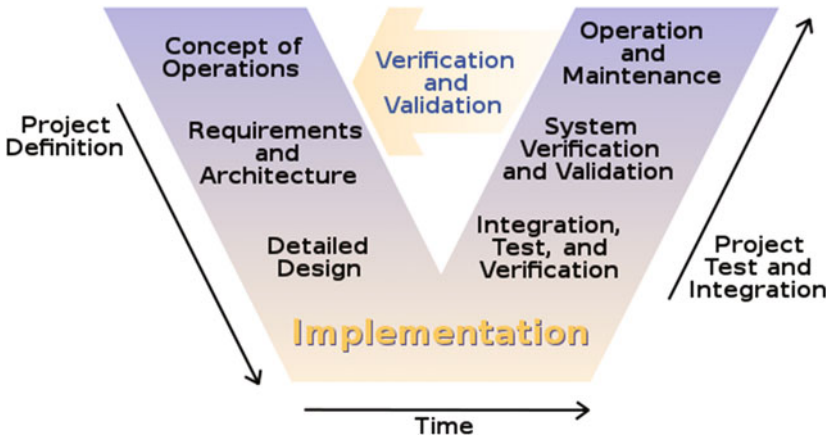


Fig. 2.19 The V-model of the system development process

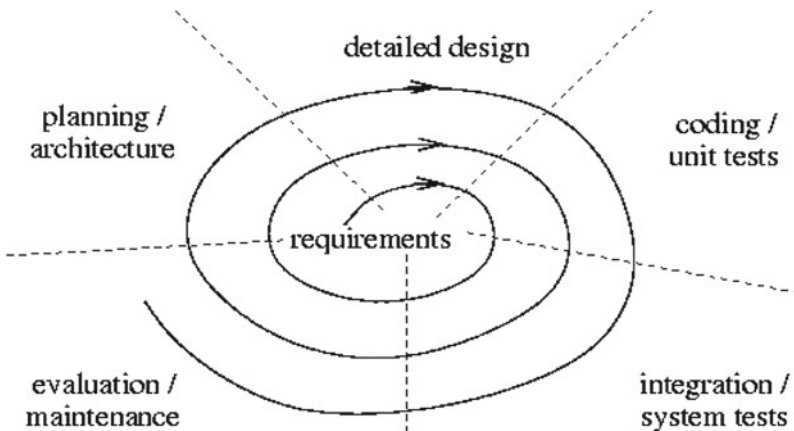


Fig. 2.20 System development in iterations

subsystem specifications considered correct, subsystem validation and verification completed—the complete system may not solve the real problem, i.e. there can be discrepancies with respect to the system level and end-user requirements. Similar issues of course may surface on lower levels of decompositions, too.

More recently developed methodologies attempts to shorten the feedback path, i.e. during the development more guidance is provided to the system designers and developers. Various iterative development process proposals were developed. A common development process structure is shared, as shown in Fig. 2.20. The underlying principle is that during the full development process a number of “complete solutions” are delivered and tested. The “completeness” is defined in such a way that the products can be shown to and experienced by the end-users, i.e. they can provide



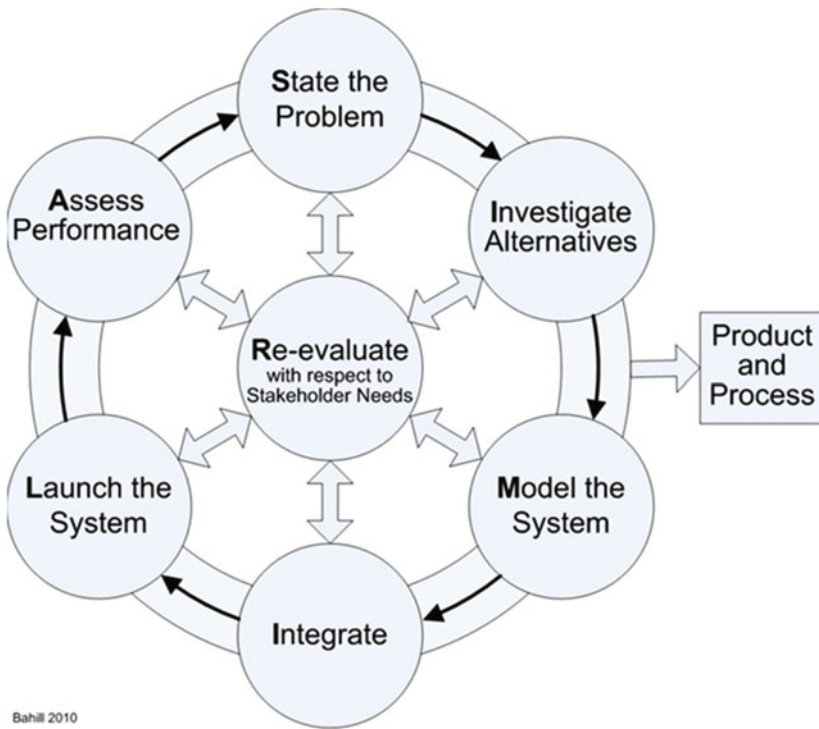
feedback about the product. The products delivered by the iterative development are typically extended variants of that of previous cycles, e.g. extended functionalities, improved user experience, tighter integration to legacy systems, etc.

The agile development methodologies bring the iterative approach to the extreme, requiring short and tightly scheduled iterations, particularly organized development teams and underlying business organizations [46].

Though iterative development processes (incl. agile methods) showed significant success in user centric and/or business software development in the (networked) embedded domain they have to be considered with caution. The tight integration of the developed system with the embedding physical processes, the new class of non-functional requirements (e.g. fault tolerance, temporal properties as condition of correctness, safety criticality, etc.) makes the specification, design implementation and testing/validation more demanding: the role of engineering disciplines is more pronounced, the validation and verifications are especially difficult/expensive/time consuming due to tight links to the embedding processes—rendering the rapid product deployment cycle infeasible. The model-based approaches come naturally as a rescue. The extensive and thorough use of models throughout the whole development process can provide frequent feedback, means of communicating with end-users, and (formal) verification of design decisions. A “pre-MDE” age development methodology emphasizing the role of modeling and design feedback is the SIMILAR systems engineering process [47]. The process is developed by comparing the similarities of several different systems engineering processes and extracting the similarities of the processes into SIMILAR process. SIMILAR is a dynamic, hierarchical, recursive and iterative process that evolves with time where several actions are done in parallel. SIMILAR process uses continuous re-evaluation functions and feedback loops as shown in Fig. 2.21. SIMILAR explicitly states the role of using models and the management of design alternatives. Also, the importance of continuous evaluation, validation and verification is emphasized by placing these activities to the center of the process and indicating the necessary interactions between the design stages. Though conceptually clean, the SIMILAR proposal cannot answer the challenge of speeding up the design process by shortening the design iteration loops and aiming for “first time right” designs.

### ***2.5.2 The Customized Design Process***

The runtime reconfiguration in networked embedded context makes the development process even more complex. The runtime reconfiguration renders the more traditional systems engineering processes inadequate and customization of the processes is needed to cope with the specialties of the runtime reconfiguration. The customized process



**Fig. 2.21** The SIMILAR systems engineering process

- should implement an iterative design scheme to assure “continuous” feedback during the design;
- should facilitate the shortening of the feedback path, i.e. the design iterations should be quick: the “quick” adjective reflects both speeding up the feedback by quick completion of the individual design stages in the iteration and shortening the feedback loop (i.e. avoiding jumping back several stages in the design process) by facilitating informed design decisions;
- should support creating “architectural design patterns”, i.e. enabling the use of proven design frameworks—especially for the integration of runtime reconfiguration mechanisms.

These requirements immediately indicates why the one of the “classic” processes won’t work: the V-model falls short considering the “length” of the design feedback loop—which translates directly to time and costs. The iterative (incl. agile) processes are very difficult to realize due to the typically overly demanding non-functional requirements and tight integration with the embedding environment. The nominal incremental functional extension and testing cycle is not feasible to carry out in this context because the the design is typically not composable due to strong interactions among design aspects (see Sect. 1.3 for details).

In the customized design process the iterative scheme is implemented along a different dimension: instead of iterating by function extension, we iterate by adding details to the design. The main goal is to evaluate, test and verify the full system in every iteration. Every design iteration delivers the relevant emerging system characteristics (key performance indicators, KPIs), which can directly be compared to the system requirements. If some of the KPIs are not compatible with the requirements, the design should be adjusted (design feedback loop). An iteration cycle's fundamental goal is to detail the design and derive more accurate KPIs. The challenge is to increase the detailing and accuracy of these steps in every iteration while minimizing the implementation work in order to assure speedy iterations.

Building the design process strictly around model-driven design approach is the most promising approach to achieve these goals. Ideally the design and implementation stages are tightly integrated: the actual implementation is the result of model refinements and automatic code generation. The frequent (almost “continuous”) validation and verification is carried out in the model domain. The process can be represented as a modified iterative design process (Fig. 2.22). In each iteration the following steps are distinguished:

- Planning
- Design
- Modeling
- Integration
- Evaluation

Each iteration delivers a “complete system”, ready for evaluation in the context of the system requirements. It can be said that essentially the designer traverses the left-hand-side of the V-model (i.e. moving from higher level design to implementation), but still the frequent check against specifications is assured (Fig. 2.23). Every design

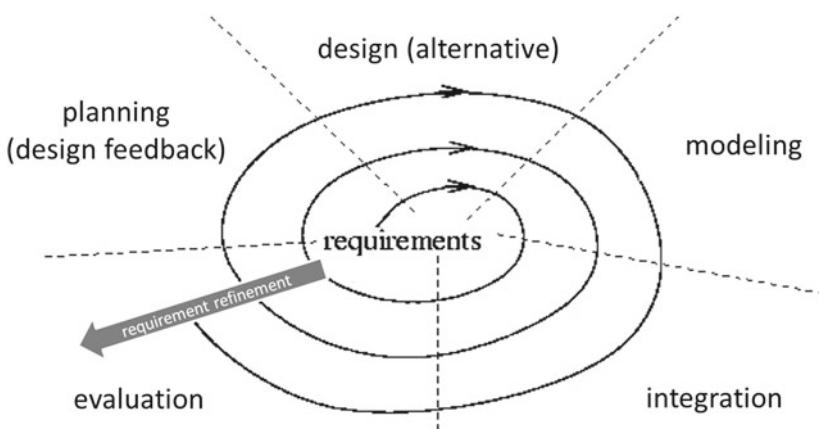
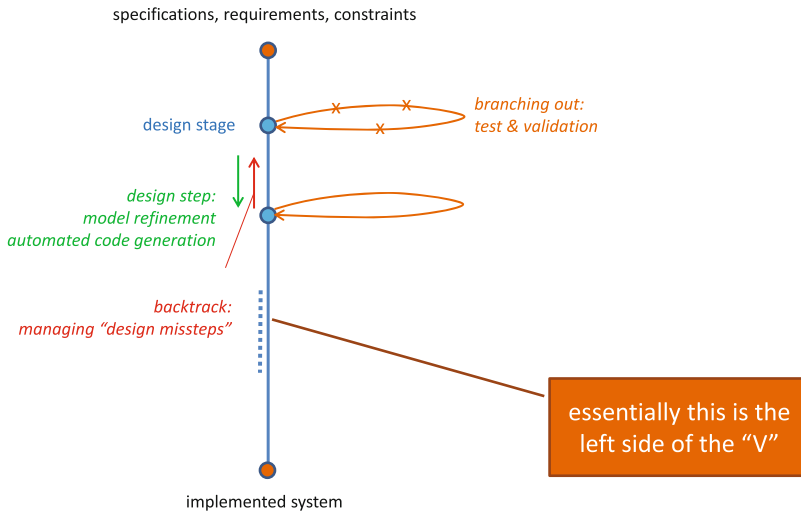


Fig. 2.22 Model driven iterative development process



**Fig. 2.23** From requirements to implementation

stage in the figure represents one full circle in the iterative scheme. As the design evolves, the next design stage is reached via completing design refinement of via automatic code generation.<sup>5</sup> In the model-driven context the refinement step covers<sup>6</sup>:

- **Adding more details:** This corresponds to the typical design decomposition activity. The designer “zooms in” on a (sub)system component and provides additional information about its composition. A typical examples are drawing a dataflow graph about a transformation element, which appears as a single “box” in the previous design stage of the system or providing additional attributes to a model element in order to facilitate higher fidelity evaluation.
- **Adding new aspect(s):** This refinement extends the scope of the modeling. Typically new aspect is introduced if a new class of requirements is to be managed. For example when the climate tolerance related requirements should be verified it will be necessary to include modeling aspects to cover heat generation (power) and heat transfer, heat radiation behavior.

At each design stage test and validation activities are carried out - see the integration and evaluation steps of the design iteration. It is important to emphasize that in parallel with the design refinements the requirements should also be detailed in order to allow more comprehensive testing.<sup>7</sup>

<sup>5</sup>In this section we focus on design refinement, i.e. when the designer further details the model to capture design and implementation decisions. Automatic code generation related concerns are covered elsewhere (see Sect. 3.2).

<sup>6</sup>For more details about model building methodology readers are referred to section ref-sec:1.6:methodology.

<sup>7</sup>In practice this is a very natural process: as the design is progresses so does the requirement specification detailing. The model driven development facilitates this aspect, too: when completing

### 2.5.3 *Managing Runtime Reconfiguration*

The needs for runtime reconfiguration emerge naturally when dealing with large-scale systems deployed in evolving environments: runtime reconfiguration can assure robust operation while keeping the system “lean and mean” and the design cost conscious. On the other hand implementing runtime reconfiguration capabilities makes the system design even more challenging.

As it was already overviewed in Sect. 1.4 runtime reconfiguration can be achieved via making two “extra functionalities” part of the deployed system:

- Building situational awareness, i.e. deriving the understanding about the execution conditions in the embedding environment and in the system itself (e.g. degradation of sensors, availability of resources, etc.);
- Managing of the situation, i.e. deriving action plan about achieving the goals as specified by the end-user and under the given situation.

The execution of these functionalities assumes the availability of sensing and actuation capabilities. The situational awareness relies monitoring of system states, which may not play role in the processing stages of the primary data path, i.e. dedicated sensing infrastructure may be needed.<sup>8</sup> The actuators (similarly to sensing) can be physical or conceptual) define the means for making changes in the configuration of the system: this can be parametric (e.g. changing parameters of algorithms (such as filter parameters)) or structural (e.g. changing communication topology, reallocating tasks, modify scheduling, etc.).

Finding the proper actuation (i.e. finding the new configuration) is the most challenging problem of implementing runtime reconfiguration. Note that the set of actuation points (and their value domain) corresponds to the free variables of the design thus it defines a design space, which should be explored during the operation of the system. Differently stated, certain design decisions are postponed to runtime and we need an “intelligent entity” to complete the design, i.e. find a reasonable (if not optimal) configuration for the given situation. Consequently in runtime

- the design space should be represented,
- algorithms should be executed to explore the design space,
- algorithms should be executed to assess the quality of the various design alternatives (generated during the design space exploration), and

---

(Footnote 7 continued)

a design iterations the results of the validation and verification can be communicated with the end-user, who can react with providing more detailed requirements or adjusting the existing ones. This is why working on the “complete system level” is important: this is the safest way to involving the end-users in the design cycles.

<sup>8</sup>This sensing infrastructure may (and typically does) consists of dedicated physical sensors (e.g. temperature sensor for ambient temperature, current measuring probe for power consumption estimation, etc.) and “virtual sensors”, which are interfaces to internal system properties, which are available but are not considered on the primary data path (e.g. input signal level of the wireless receiver, CPU utilization, etc.).

- algorithms should be executed to guide the design space exploration, i.e. to find an optimal/reasonable solution.

Extreme caution is needed when selecting the actuation points because contradicting requirements:

- The number of actuation points and their value set defines the design space for the search for solution—which may easily become unacceptably large (exponential complexity). Consequently there is a need for minimizing the free parameters of the design.
- On the other hand the size of the design space determines the adaptation capabilities of the system: larger the space (i.e. more actuation possibilities) wider the operational envelop the system. This is definitely a incentive for introducing more actuation points.

These contradicting requirements should be balanced taking into account the availability of resources for executing the reconfiguration related activities and temporal constraints for reconfiguration. This is where the model driven engineering comes as a “rescue”. Runtime reconfiguration related functionalities should be modeled as set of interacting tasks. Depending on

- the selected design space representation,
- the associated design space exploration algorithm,
- the dimensions and the size of the design space, and
- the selected search/optimization algorithm

the resource requirements of the reconfiguration can be estimated. Based on these estimations the reconfiguration related tasks in the task model can be properly parametrized and after determining the task mapping (Sect. 1.2) the system model is completed and ready for evaluation under end-user defined scenarios. It should be emphasized that besides taking the resource demand of the reconfiguration into account, the effectiveness of the reconfiguration should also be estimated and these “effectiveness figures” should be evaluated in the context of the system level requirements (e.g. required quantified system performance characteristics under user-defined set of failure modes).

## 2.6 Conclusions

One of the major challenges when designing software for complex systems is caused by a lack of a specific set of rules (methodologies). Adaptation (reconfiguration) to field conditions is difficult to model and implement on systems composed of a larger number of devices/components (distributed systems or systems of systems).

For state-of-the-art technology such as wireless sensor and actuator networks (cyber-physical systems), addressing the lack of a compressive set of rules for their

design and realization offers considerable benefits. This leads to can accelerate and simplify their design and implementation.

With the extensions presented in this chapter, a model driven design approach becomes applicable to runtime reconfigurable cases. It should be noted that designing runtime reconfigurable system still remains a complex process. It starts with the extra difficulties requirement specification, continuous with the more demanding modeling and evaluation iterations and ends with challenging validation and verification. This latter has far reaching consequences for the whole design and will be considered in detail in Part 2. Model driven engineering fundamentally integrated into the development process is the key for the successful system design. We have to emphasize the importance of the careful balancing of the design-time–runtime trade-off and the importance of the high fidelity modeling of the situation assessment and management algorithms for implementing runtime reconfiguration. There are no “ultimate solution” and “design pattern” for runtime reconfiguration is general. Generic approaches can easily become unfeasibly complex thus taking into account the specialties of the case in hand and using proven heuristics are necessary. This emphasizes the importance of experimenting and thorough case-based evaluation. The proposed model driven development process can greatly speed up design iterations and reduce costs. Due to the design evaluation and testing challenges of networked embedded systems the seemingly extra effort needed for setting up and following strict model based design approach will be greatly compensated for during the later stages of the implementation, testing, deployment and maintenance. Modern systems engineering processes can easily be customized to accommodate for the specialties of runtime reconfigurable systems.

According to our experience the model centered methodology for building runtime reconfigurable systems is the main enabler for the widespread application of this technology.

## References

1. M. Palesi, T. Givargis, in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign, 2002. CODES 2002* (IEEE, 2002), pp. 67–72
2. D.E. Goldberg, (Addison-Wesley Professional, 1989)
3. Q. Liu, S. Dulman, M. Warnier, Area: an automatic runtime evolutionary adaptation mechanism for creating self-adaptation algorithms in wireless networks (2013). Under submission
4. J.C. Georgas, A. van der Hoek, R.N. Taylor, *Computer* **42**(10), 52 (2009)
5. J. Teich, M. Köster, in *Proceedings of the Conference on Design, Automation and Test in Europe* (IEEE Computer Society, 2002), p. 559
6. S.Y. Kuo, W.K. Fuchs, in *Proceedings of the 25th ACM/IEEE Design Automation Conference* (IEEE Computer Society Press, 1988), pp. 609–612
7. H.W. Kuhn, *Nav. Res. Logist. Q.* **2**(1–2), 83 (1955)
8. S. Kogekar, S. Neema, B. Eames, X. Koutsoukos, A. Ledeczi, M. Maroti, in *Proceedings of the 3rd international symposium on Information processing in sensor networks* (ACM, 2004), pp. 379–387
9. B. Eames, in *2006 IEEE Mountain Workshop on Adaptive and Learning Systems* (IEEE, 2006), pp. 127–132

10. T. Syrjänen, *A rule-based formal model for software configuration*, Technical Report (Helsinki University of Technology, 1999)
11. P. Simons, I. Niemelä, T. Soinen, *Artif. Intell.* **138**(1), 181 (2002)
12. OWL, <http://www.w3.org/TR/owl-time/>
13. K.R. Apt, R.N. Bol, *J. Log. Program.* **19**, 9 (1994)
14. M. Ben-Ari, *First-Order Logic: Logic Programming* (Springer, London, 2012), pp. 205–222
15. R.F. Stärk, *A Direct Proof for the Completeness of SLD-resolution* (Springer, Berlin, 1990), pp. 382–383
16. B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, L. De Raedt, *Theory Pract. Log. Program.* **11**(4–5), 663 (2011)
17. C. van Leeuwen, Z. Papp, J. Sijs, in *16th International Conference on Information Fusion* (2013)
18. R. Isermann, *IEEE Trans. Syst. Man Cybern. Part A Syst. Hum.* **28**(2), 221 (1998)
19. C. Aubrun, D. Sauter, H. Noura, M. Robert, *Int. J. Syst. Sci.* **24**(10), 1945 (1993)
20. F. Bobillo, U. Straccia, in *IEEE International Conference on Fuzzy Systems, 2008. FUZZ-IEEE 2008* (IEEE World Congress on Computational Intelligence) (IEEE, 2008), pp. 923–930
21. F. Bobillo, M. Delgado, J. Gómez-Romero, *Expert Syst. Appl.* **39**(1), 258 (2012)
22. L.A. Zadeh, *Inf. Control* **8**(3), 338 (1965)
23. L. Zadeh, *IEEE Comput.* **21**(4), 83 (1988)
24. H.J. Zimmermann, *Fuzzy Sets theory and its Applications*, 3rd edn. (Kluwer, Boston, 1996)
25. W.A. Kwong, K.M. Passino, E.G. Laukonen, S. Yurkovch, *Proc. IEEE* **83**(3), 466 (1995)
26. P.U. Lima, G.N. Saridis, *IEEE Trans. Syst. Man Cybern. Part B: Cybern.* **29**(2), 151 (1999)
27. J.M. Bernardo, A.F. Smith, *Bayesian Theory*, vol. 405 (Wiley, 2009)
28. C. Jung, K. Kwon, *Control Cybern.* **27**, 545 (1998)
29. R. Tong, *The Construction and Evaluation of Fuzzy Models* (North-Holland, Amsterdam, 1979)
30. S. Kim, E. Kim, M. Park, *Fuzzy Sets Syst.* **81**(2), 205 (1996)
31. F. Matia, A. Jimenez, *Int. J. Intell. Control Syst.* **1**(3), 407 (1996)
32. T. Tobi, T. Hanafusa, *Int. J. Approx. Reason.* **5**(3), 331 (1991)
33. Y.K. Kwok, I. Ahmad, *J. Parallel Distrib. Comput.* **59**(3), 381 (1999)
34. D. Grosu, A.T. Chronopoulos, M.Y. Leung, in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM* (IEEE, 2002), pp. 52–61
35. A. Aminifar, S. Samii, P. Eles, Z. Peng, in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, vol. 1 (IEEE, 2011), pp. 133–142
36. Y. Yi, W. Han, X. Zhao, A.T. Erdogan, T. Arslan, in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09* (IEEE, 2009), pp. 33–38
37. J. Sijs, Z. Papp, in *2012 15th International Conference on Information Fusion (FUSION)* (IEEE, 2012), pp. 1012–1019
38. T. Streichert, D. Koch, C. Haubelt, J. Teich, *EURASIP J. Embed. Syst.* **2006**(1), 9 (2006)
39. G. Cybenko, *J. Parallel Distrib. Comput.* **7**(2), 279 (1989)
40. P. Neelakantan, *Int. J. Comput. Appl.* **39**(4), 7 (2012)
41. [https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\\_bestpractices\\_TP026B.pdf](https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf)
42. ISO/IEC 107463 (1996)
43. G. Karsai, J. Sztipanovits, *Intelligent Systems and their Applications*, *IEEE* **14**(3), 46 (1999). doi:[10.1109/5254.769884](https://doi.org/10.1109/5254.769884)
44. T. Saxena, A. Dubey, D. Balasubramanian, G. Karsai, in *2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASE)* (2010), pp. 137–144. doi:[10.1109/EASE.2010.22](https://doi.org/10.1109/EASE.2010.22)
45. B. Berenbach, D.J. Paulish, J. Kazmeier, A. Rudorfer, *Software and Systems Requirements Engineering: In Practice Education* (McGraw-Hill, 2009)
46. J. Whittle, J. Hutchinson, M. Rouncefield, *Software*, *IEEE PP*(99), 1 (2013). doi:[10.1109/MS.2013.65](https://doi.org/10.1109/MS.2013.65)
47. A. Bahill, B. Gissing, *IEEE Trans. Syst. Man Cybern. Part C: Appl. Rev.* **28**(4), 516 (1998). doi:[10.1109/5326.725338](https://doi.org/10.1109/5326.725338)



Runtime Reconfiguration in Networked Embedded  
Systems

Design and Testing Practices

Papp, Z.; Exarchakos, G. (Eds.)

2016, XXII, 171 p. 85 illus., 62 illus. in color., Hardcover

ISBN: 978-981-10-0714-9