


FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

# Exploring Dynamic Reconfiguration in Software Architectures

**Seminar Thesis**

presented by

**Engels, Christoph**

Matriculation Number: 3216817

**1st Supervisor: Prof. Dr. B. Kraft**

**2nd Supervisor: David Schmalzing, M.Sc.**

**Eidesstattliche Erklärung**

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Exploring Dynamic Reconfiguration

in Software Architectures

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Engels, Christoph

Aachen, den 15. 12. 2020

Unterschrift der Studentin / des Studenten

## Abstract

Dynamic reconfiguration means changing the configuration of an application while it is executing. The ability to do so significantly increases the flexibility of an application. Hence, dynamic reconfiguration is a helpful tool when striving for this goal. Many architectures, however, are static, even if their application requires dynamism. Because of this, there are many approaches to modeling dynamic reconfiguration. We will compare some of them.

Dynamic systems are more flexible than static ones. This benefit can be used to expand the range of functionality the system offers. It can also allow a system to evolve into arbitrary structures, as designers can continue developing the system and update single components without halting the running application. Furthermore, increased flexibility is beneficial in systems that are running in ever-changing environments. Here dynamic reconfiguration can help to develop a system that can correctly respond to changing requirements.

In this work, we present multiple different approaches to achieving dynamic reconfigurability for various use cases. We describe their individual differences and advantages as well as the general advantages of utilizing dynamic reconfigurability.

We focus on employing dynamic reconfiguration in software architectures in general and in distributed systems as a special variant of that. Additionally, we also look at the dynamic reconfiguration of hardware architectures, which is quite different from the other two versions.

# Contents

# 1 Introduction

Dynamic Reconfiguration means modifying the structure of a program at its execution time. The concrete modifications, however, differ depending on their use case. In this work, we will explore different types of dynamic reconfiguration in various kinds of software architectures.

The ability to perform dynamic reconfiguration generally improves the flexibility of an architecture, because it allows an application to adapt to requirements that might occur during execution time. While approaches that limit the number of possible configurations at design time cannot achieve a maximum of flexibility, they are more predictable. Therefore, they are more suitable for detailed static analyses, which helps to ensure the correctness of an architecture.

Dynamic reconfiguration processes may be executed by external programs or by the architecture itself. They might also be triggered externally and then performed without external help. Another important point is the timing of the process, which can differ among the approaches, too.

In this work, we will, at first, address dynamic reconfiguration of hardware in chapter 2. We will describe how dynamic reconfiguration can be used in the context of hardware architectures, as well as listing up advantages and disadvantages for doing so in section 2.1.

In software architectures, however, dynamic reconfiguration looks quite different because software architectures can be manipulated more easily. We will focus on that in chapter 3. It begins with a short description of component and connector architectures in section 3.1. Section 3.2 introduces an example use case of reconfiguration and addresses the issue of timing reconfiguration processes. In contrast to that, the following section 3.3 depicts how dynamic reconfiguration can be avoided in a static architecture and which downsides that has. After that, we give an overview of different kinds of reconfiguration mechanisms and how parts of the dynamic structure can be reused.

We furthermore describe how dynamic reconfiguration mechanisms shine in the area of distributed systems in chapter 4. There we give an overview of how dynamic reconfiguration helps to make a system less vulnerable to failure of components in section 4.1. And we show how components can be cleverly replaced during runtime in section 4.2.

# 2 Dynamically Reconfiguring Hardware

In the context of hardware, dynamic reconfiguration means replacing the logic of certain circuits while they are executing. This allows designers to specify multiple behavior models for one circuit that can be switched during runtime, which greatly increases the flexibility of the hardware. The reconfiguration, however, takes time and energy, and utilizing it may be complicated.

A field-programmable gate array (FPGA) is a kind of integrated circuit which means that it contains an amount of physically implemented logic gates. Those gates are hard-wired and therefore quicker in executing their instructions than a CPU would be. The logic gates are grouped into frames [BML03]. Some FPGAs allow replacing frames with other predefined ones. Together with corresponding additional logic and memory, this allows changing the configuration of the circuit at execution time. This is the basis of dynamic reconfiguration of hardware architectures. These FPGAs are SRAM-based FPGAs [MVV+15]. Since this type of FPGA is the only one that allows reconfiguration at runtime it is the only type relevant to this work. We will therefore always refer to SRAM-based FPGAs when using the term FPGA in the following.

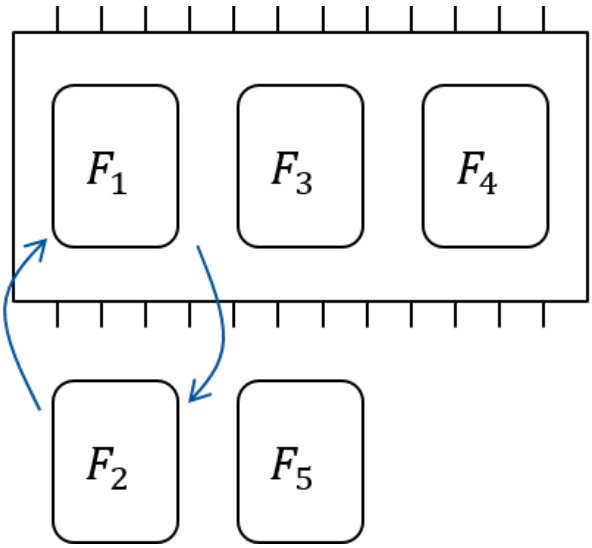


Figure 2.1: If fitted with external memory, a dynamically reconfigurable FPGA may implement more functionality than its working memory allows

Not all sections of the FPGAs have to be dynamically interchangeable. In the case of partial reconfiguration, a user may define static areas that do not change during execution time. These parts usually control the repartitioning of the circuit's dynamic area [MVV+15].

## 2.1 Traits of Dynamic Hardware

Using Dynamic Reconfiguration in hardware offers many benefits. Three of them are described by [MVV+15]. Firstly, the startup of circuits can be accelerated, because only required parts of the circuits have to be loaded when booting. Other components can be loaded later when they are needed. That way, there is less effort during startup which can save some time.

Furthermore, the use of dynamic reconfiguration implies a more modular and therefore quicker development approach. Programming a static FPGA means designing the whole circuit as one unit. As a result of that, designers may have to rework the complete application and adapt changes even if they apply only a small adjustment to an almost finished product. The sections in a setup that aim at dynamic reconfigurability are by nature less dependent on each other since they might have to function independently at runtime when one part is active and one is not. This means they are coupled more loosely. Hence the impact of a modification in this kind of system will be more limited locally and therefore easier to manage. This facilitates the development of FPGAs.

A third advantage is the fact that dynamic architectures need fewer resources compared to static ones. Usually, not every functionality of a circuit is required at the same time. Therefore, the use of dynamic reconfiguration allows deactivating sections that are not needed at that moment or replacing them with ones that have to be on service. The deactivated sections do neither consume energy nor occupy active memory. This lowers the overall power demand and allows designers to choose a smaller FPGA for the same task, as shown in fig. 2.1.

Dynamic reconfiguration may also lead to a general speedup of a system, as brought up by [LCD+00]. An architecture in which a CPU is paired with a dynamically reconfiguring FPGA may run quicker than alternative ones. That is because, in this setup, a compiler can distribute the instructions generated from code among both elements. The CPU executes some parts of the code whereas expensive instructions that have to be used frequently can be processed by the FPGA. The later one is quicker because the instructions are programmed in physically. Dynamic Reconfiguration enables the FPGA to fulfill more functionality than its memory allows, as described above.

On the other hand, the usage of dynamic reconfiguration in hardware also bears challenges. Replacing partitions during execution costs time. This problem is addressed by [LCD+00] who introduce a compiler that can evaluate whether the speed gain through hardware execution compensates this overhead.

Dynamic reconfigurability also adds complexity to a circuit. This makes the development of dynamically reconfiguring circuits more complicated and therefore its introduction may bear some initial costs. [MVV+15] aim at solving this by presenting a concept project that describes some examples and best practices as well as providing a cheap testing environment for programming FPGAs.

Although clever use of dynamic reconfiguration can lower the energy requirements of an FPGA, the reconfiguration process itself needs additional power. Quickly changing the complete logic of a circuit is very especially expensive, even more so on large circuits. This is brought up by [TCJW97]. They slightly reduced energy demand by lowering the voltage and memory required for reconfiguration. Storing the configuration data locally close to where it is used also helps saving resources.

## 2.2 Application

An exemplary utilization of dynamic reconfiguration is described by [TCJW97], who make use of the fact that dynamic reconfiguration allows describing more logic with less hardware, as described in section 2.1. They describe an FPGA that switches between nine possible configurations periodically. In their case reconfiguration always means changing the state of the circuit completely, not partly. While the FPGA is in one configuration, the eight other configurations are saved on inactive memory which is scattered around the chip. This allows every inactive piece of the architecture to be stored very close to the location it has to be at when it is active. This accelerates the reconfiguration process as the pieces have to travel less distance to switch their memory location during the process.

They present three reconfiguration mechanisms for their FPGA. The first one is the static mode, which makes the FPGA behave like a static circuit. In time share mode dynamic reconfiguration is used to simulate nine virtual FPGAs that communicate with each other. With each active configuration, the FPGA represents another virtual circuit. Therefore, the FPGA loads previously calculated values at first, runs some computation cycles, saves its results, so future configurations can access them, and changes its configuration. When to reconfigure and which configuration should be applied next may be decided by an external controller. In logic engine mode the FPGA simulates a bigger circuit, where each configuration of the FPGA represents another part of the bigger virtual FPGA. In contrast to the previous mode, the order of configuration defined in advance and fixed and timing is more sensible as the FPGA has to reconfigure as soon as it has finished its computation cycle. Otherwise, the single computations cannot work together to correctly emulate a bigger circuit. Hence the FPGA reconfigures more frequently in logic engine mode than in time share mode.

# 3 Dynamic Reconfiguration of Software

In the software area, dynamic reconfiguration can mean replacing, updating adding, and even deletion of parts of the architectural structure. It mainly increases the flexibility of a system. There are many approaches to utilizing reconfiguration for various use cases in different ways.

## 3.1 Component and Connector Architectures

Component and connector architectures are common types of software architectures. Due to their graphical background, they are very suitable for visualizing dynamic reconfiguration. Hence, many approaches, that describe dynamic reconfiguration of software, rely on component and connector architectures. Since there are many approaches to modeling architectures of this kind, there may be multiple terms for describing the same element. In this work, we will use the terminology and graphical representation of MontiArc [Wor16].

As the name implies, component and connector architectures consist of interconnected components. Components model units of computation. Their interface is defined by ports. Through incoming ports, a component receives data of a certain type from its environment, whereas outgoing ports indicate that a component provides a certain type of data. Ports of matching type can be linked by connectors, with respect to their directions. Connectors transfer the data between connected ports. The kind of moved data depends on the modeling language. MontiArc usually transfers data-values, whereas Darwin [MDEK95] communicates requests or confirmations for providing certain types of functionality. Depending on the used language the direction of a connector may be bidirectional, which is the case in Wright [ADG98] or unidirectional like in MontiArc. A port that outputs something may be connected to multiple incoming ports receiving the same thing. One input port, however, must not be linked to more than one output port, because in that case the data coming from the set of ports could be mixed unpredictably.

Components can either be atomic or composed. In the first case, their behavior is described independently from the architecture of the system. Hence they appear as black boxes in the architectural structure. Composed components on the other hand consist of other components, which, again, may be atomic or composed. Connecting the ports of those subcomponents allows the components to communicate with each other. Connecting them to ports of the encapsulating components forwards the connection to an upper layer. The substructure of a composed component, therefore, describes the behavior of the component.

Figure 3.1 shows an example of a simple robot. A sensor called "mainSensor" and a robot arm called "arm" are subcomponents of the robot. The sensor can measure a certain distance. Through the connector, it sends its measurements to the arm, which then uses the data to calculate its movements.

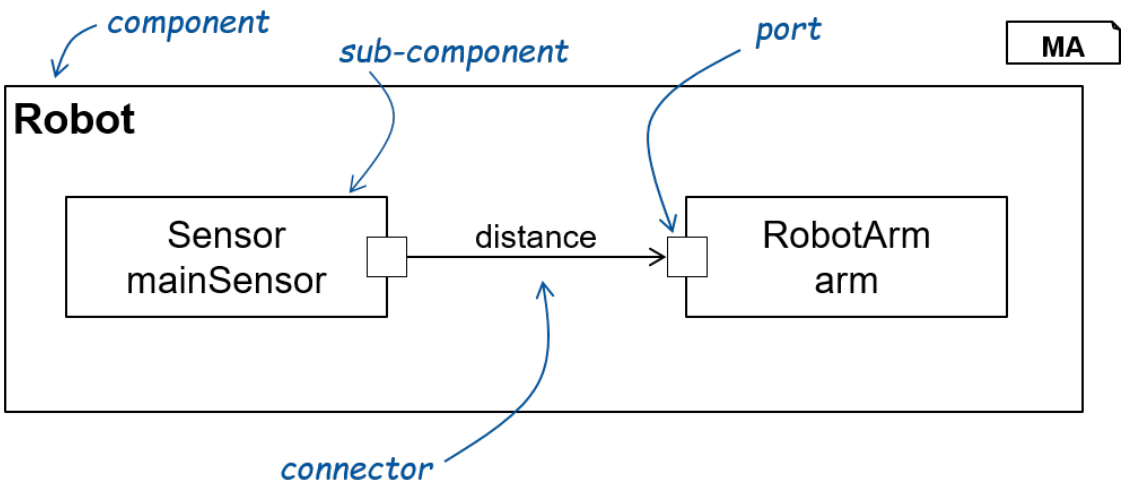


Figure 3.1: A simple composed component

## 3.2 Reconfiguration of Components

Some of the component and connector modeling languages support modeling dynamic reconfiguration of the architectures they describe. That usually means they support adding and deleting components during execution time as well as addition, removal, and redirection of connectors.

Let us assume that the sensor used in the example introduced previously is not completely reliable and may turn off unpredictably during runtime. Figure 3.2 shows how dynamic reconfigurability offers an easy solution to this problem. The robot now contains a second sensor which will replace the first one if that one goes out of service. In this case, the robot just has to switch the source of the connector from one sensor to the other one. Hence the robot may change between two configurations, which are both shown in the diagram. This is an elegant solution because the subcomponents of the robot do not have to change at all, the reconfiguration is hidden from them.

Since no software process can happen instantly, dynamic reconfiguration also takes some time. For this and perhaps other reasons correct timing of the reconfiguration is important. Usually, the reconfiguration events should wait for ongoing computations to ensure that there is no data lost during the process. Many approaches just prohibit reconfiguration at inconvenient times. For instance, in Wright [ADG98] or in the approach of [HP93] designers have to specify when configurations are allowed occur.

There are, however, approaches that are more specific about timing the reconfigurations. The one of [GH04] is an example for that. They fit all components are with a state chart, which ensures that every involved component is not active anymore before a reconfiguration can occur. This statechart consists of three main states. Active components are working normally. Passive components have finished all processes they initiated and will not request further ones, but may still process requests of other components. Finally, quiescent components are ready for reconfiguration, as no other components will request services from them, too. There are further intermediary states which ensure that all involved components confirmed each other's states. It is also possible to model components

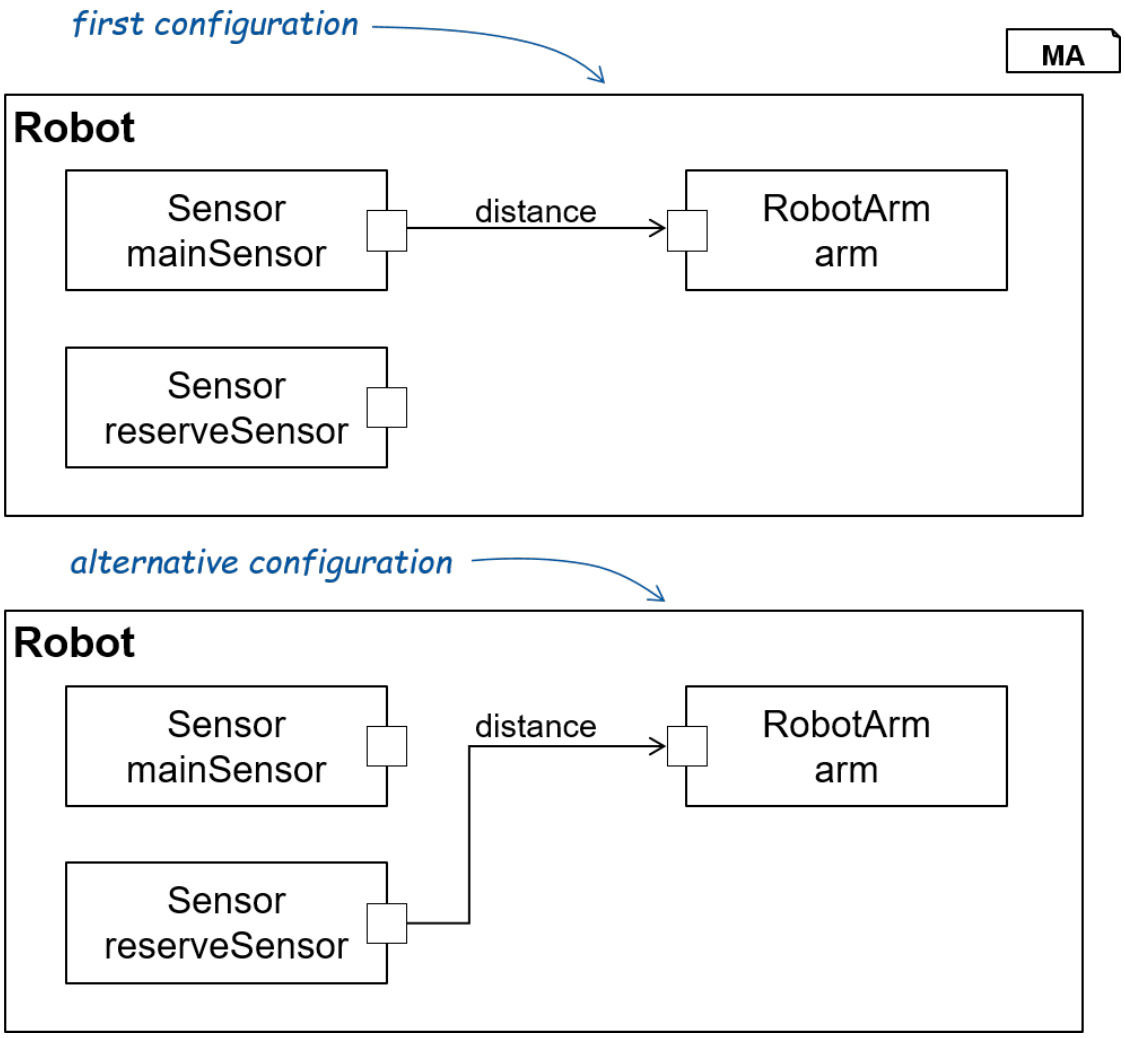


Figure 3.2: An advanced robot with two configurations. In the first configuration it uses its main sensor, in the second configuration, it uses the alternative sensor

that go only partially quiescent, which means that a component is quiescent concerning some ports but active at other ones. In their approach, a designer may also declare additional statecharts for more user-specific behavior.

The quiescence criteria, however, is quite strict, as driving components to this state, even partially, can seriously interrupt the program flow. It is indeed sufficient to delay a reconfiguration to a tranquil state, as [VEBD07] describe. Tranquility is a reduced version of quiescence. For both states, a component may not be processing services it requested and may not request new ones. But a quiescent component is not and will not be involved in any requests from other components. Tranquil components on the other hand may be involved in a request if they are not actively contributing to it currently and they either did not do so in the past or they will not do so in the future. The last condition ensures that if a component takes part in a process and is then replaced, the new component will not have to continue the communication with the state of the replaced component. An elegance of this approach is, that in practice most components will frequently turn tranquil on their own. So if the system has to reconfigure, it will just wait until the corresponding

components reach a tranquil instead of instructing them to do so, which would be an interruption of the program. Nevertheless, if a component does not turn tranquil quick enough it must be instructed to turn quiescent, as described above. The behavior of the components, however, cannot be hidden as a black box, since that would make it impossible to determine when a component participates actively and when it participates passively in a process, which is crucial for detecting tranquility.

Some mechanisms explicitly allow reconfiguration at any time, like the ones of [GMK02] or [VVE16]. Other approaches like Darwin [MDEK95] only model architectural structure but not computations, hence they do not need to address the problem of timing the reconfigurations.

Reconfiguration mechanisms can also demand specific requirements for the timing of events during one reconfiguration process. In Wright [ADG98] the connectors have to be unlinked at first, then components can be deleted, after that connectors can be reattached, and finally, new components can be created. This pattern is expanded by [GH04]. Here the involved components are ordered to go partly or complete quiescent at first and then some components are ordered to save their current state. Then the architectural structure changes, similar to before, but with a slightly different order, by detaching connectors, deleting components, creating new components and reattaching connectors. Afterward, one component is ordered to leave its quiescent state and new components that replace old ones adapt their state by using the state information saved before. Then all other components are reactivated.

## 3.3 A Static Alternative

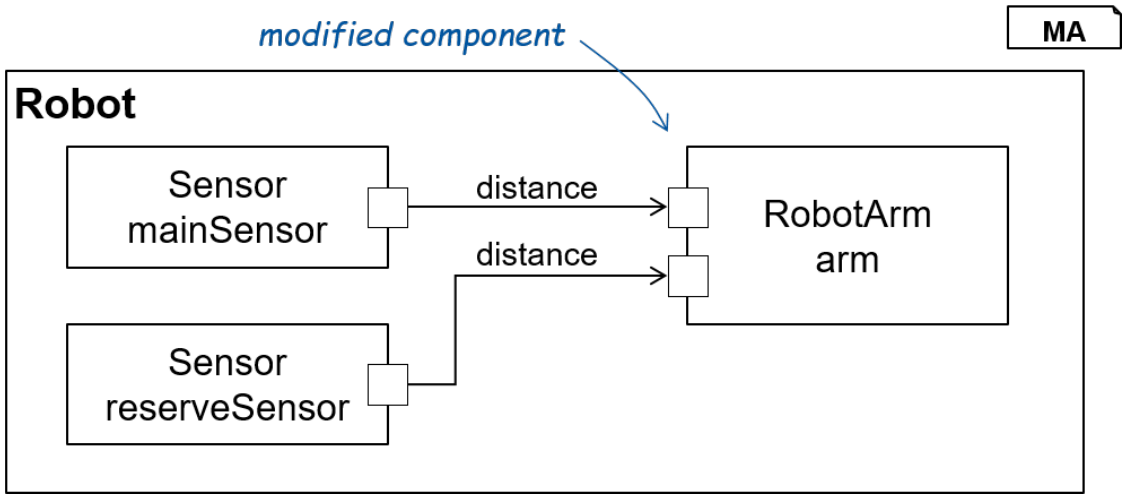


Figure 3.3: A static variant of the fail-safe robot that does not rely on dynamic reconfiguration

Dynamic reconfiguration changes the behavior of composed components by altering their structure. Since the behavior of those components is indirectly defined by their subcomponents, and each component finally consists of atomic components, one might also change the behavior of those to achieve the same effect. It might be easier to change the state of

atomic components to modify their behavior. However, in larger architectures, this might lead to having multiple components whose state has to be kept synchronous. So managing this state gets difficult as it will create dependencies between the components. Therefore, dynamic reconfiguration is more elegant [ADG98].

For example fig. 3.3 models another version of the robot presented in fig. 3.2. This implementation does not rely on dynamic reconfiguration as the arm is connected to both sensors permanently. Instead, the arm itself decides when to use which sensor. To achieve this, however, the arm component has to be adjusted as its modified interface contains two ports. Furthermore, the modified component has two tasks to fulfill, as it also has to select which sensor values to use, next to controlling the arm of the robot. This contradicts a clear distribution of duties where every element has exactly one purpose, which is generally favored in software architectures, as it leads to high cohesion.

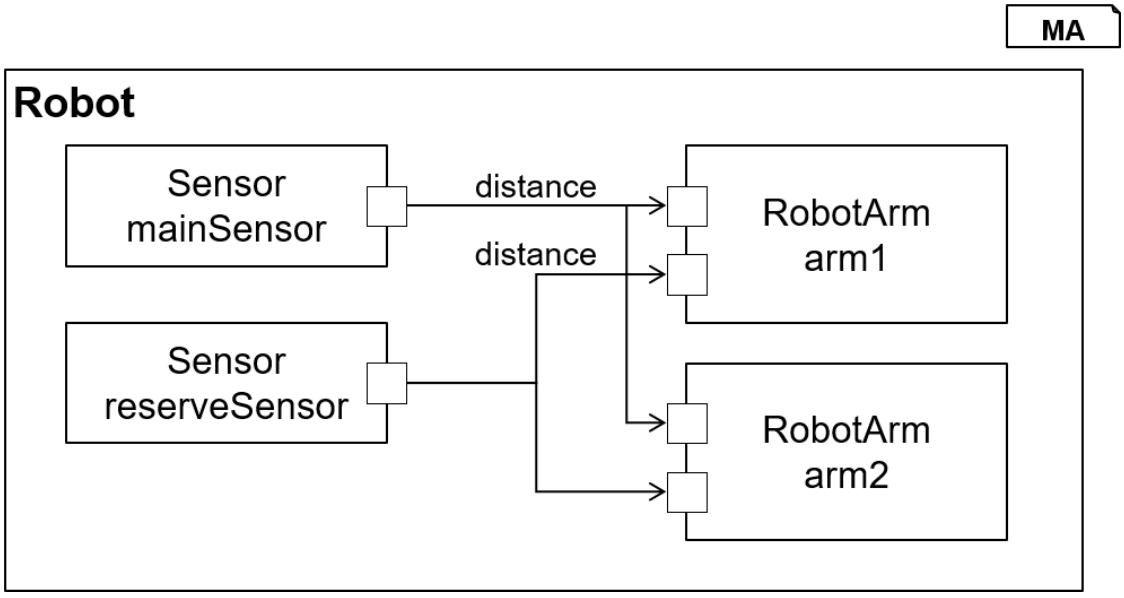


Figure 3.4: The non-reconfiguring variant of the fail safe robot does not scale up very well

Another downside of this approach is that it is less flexible for changes. Adding another arm to the robot as showcased in fig. 3.4 illustrates this problem. If the robot arms have to move in sync with each other, they must rely on the same distance values and therefore always fetch their data from the same sensor. Since both arms decide on their own which sensor they use, addressing this issue may be complicated. In a dynamic architecture, the decision of selecting the sensors is centralized in the robot, hence this problem does not occur.

## 3.4 Types of Dynamic Reconfiguration

As the idea of dynamic reconfiguration is not new, there are several approaches to utilizing it in software architectures. There are six main aspects in which the approaches can differ [BHK+17]. In this classification, a reconfiguration may be restricted or open, ad-hoc or programmable, self-directing or not, instructed or triggered, and imperative or declarative.

Finally, a reconfiguration mechanism may or may not allow dynamic instantiation and deletion of components.

The first point is whether the approach supports the creation or deletion of components at runtime. As it can be hard to integrate new components into or remove other ones from an already valid architecture, some approaches only allow changing connectors. For instance, [MDEK95] point out, that their notation does not allow creating components with outgoing ports usefully. They do not support the deletion of components either.

The next one is the count of possible configurations of the architecture. In a restricted approach there is a finite number of configurations at design time the architecture can select from at runtime. In an open approach, there is either an infinite number of possible configurations or they are not limited at all during design time. For example the dynamic robot of fig. 3.2 can only exist in two configurations. Therefore, it could be modeled in a restricted approach. Modeling another robot that can create and use infinitely many sensors during runtime means that there are infinitely many configurations. Yet another robot that can create, delete and connect any kind and number of components arbitrarily is not limited at all. For both cases, a designer has to choose an open approach.

In Ad-Hoc reconfiguration mechanisms changes to the architecture are triggered and performed by an external agent. In this case, the architecture does not have to be specifically modeled as reconfigurable, as the dynamic part is completely hidden from it. Another option is to model the reconfiguration as part of the architecture, which they call programmed.

Programmed approaches may be self-directing. This means that every component may only reconfigure itself or its internal structure, instead of modifying any part of the architecture. This helps in keeping the components independent from each other.

Another aspect is the difference between instructed and triggered reconfiguration. Instructed means that a component will only instantiate a reconfiguration if it receives a respective command. In a triggered approach, a component may have special conditions that need to be met in order for it to start a reconfiguration. For example, the robot of fig. 3.2 may switch to its second configuration as soon as the main sensor stops delivering values.

Open reconfigurations as well as triggered ones, ad-hoc reconfiguration, non-self-directing systems, and approaches that allow creation and deletion of components offer more flexibility than their alternatives at the cost of predictability. Hence the flexible options are less suitable for static analyses of the architecture. They are also usually slightly more complex but more powerful since triggered specifications can model instructed behavior, non-self-directing ones can emulate self-direction and open ones are by definition less restricted than restricted ones. And modifying connectors is also possible in approaches that allow modifying components.

The last difference they point out is the notation style, which has no direct effect on the flexibility of the approach. Imperative reconfiguration looks like object-oriented or imperative programming languages [WW90]. Here designers present and modify architectural elements like objects. Declarative approaches on the other hand rather specify the configurations an architecture can evolve to. In this case, designers do not have to handle every type of element separately and define bigger structures in one part.

Furthermore, the work of [BHK+17] includes a table with 23 architecture description

```
component Robot {

  component RobotArm arm;
  component Sensor mainSensor;
  component Sensor reserveSensor;

  mode Standard {
    use arm;
    use mainSensor;
    use reserveSensor;

    connect mainSensor.distance -> arm.distance;
  }

  mode Alternative {
    use arm;
    use mainSensor;
    use reserveSensor;

    connect reserveSensor.distance -> arm.distance;
  }

  modeautomaton {
    initial Standard;

    Standard -> Alternative [mainSensor.distance < 0];
    Alternative -> Standard [mainSensor.distance >= 0];
  }
}
```

Listing 3.1: MontiArcAutomaton notation of the robots reconfiguration.

languages that support dynamic reconfiguration. The table gives an overview about the traits each approach has.

For example listings 3.1 to 3.3 show various textual representations of the robot introduced in figure 3.2. The robot just redirects the connector whenever the main sensor produces faulty values. For simplicity's sake, we assume that the state of the main sensor can be assessed by evaluating its output. It is working correctly as long as it is emitting positive values, but it has to be replaced temporarily if the values are negative, because negative distances are not allowed in this context. Listing 3.1 uses the MontiArcAutomaton language [HKR+16] to describe the robot, whereas listing 3.2 uses LEDA [CCT99] and Listing 3.3 uses Wright [ADG98]. Since MontiArcAutomaton specifies reconfiguration declaratively, it describes the configurations as modes and also provides a statechart (ll. 23) which describes when to switch to which mode. On the other hand, LEDA, which is imperative, specifies the sensor-connection as an assignment statement (l. 11). Wright also uses assignment statements (ll. 15, 16, 23, 24) to reconfigure the connector, since it also is an imperative approach. But in contrast to the other two languages, which are triggered, Wright is instructed. That means that it does not allow analyzing the output of the sensor to determine when to change configurations. Instead it relies on additional control events (ll 13, 21) the sensor has to provide.

```
component Robot {

  interface none;

  composition
    arm:RobotArm;
    mainSensor:Sensor;
    reserveSensor:Sensor;

  attachments
    arm.distance(d) <> if (mainSensor.distance >= 0)
                          then mainSensor.distance(d)
                          else reserveSensor.distance(d)
}
```

Listing 3.2: LEDA notation of the robots reconfiguration.

## 3.5 Reconfiguration Patterns

The configurations an architecture can evolve into during its runtime can also be modeled as parts of a software product line [GH04]. This facilitates the reuse of certain structures. The authors especially provided four different reconfiguration patterns that can be used in addition to usual software patterns when designing a dynamically reconfigurable architecture.

The first one is the master-slave reconfiguration pattern, where one master component sends requests to multiple slaves. The slaves on the other hand are only communicating with the master. They only answer the request but are not allowed to send requests themselves. This accelerates the transition to a quiescent state (cf. section 3.2) of the components. Changing the master affects all slaves, whereas changing the slaves only affects the master and can therefore happen independently from other slaves.

The centralized control system reconfiguration pattern features a centralized controller component that does not reconfigure. Changes to components cannot happen before the central controller turns quiescent.

Opposed to that is the decentralized control system reconfiguration pattern. It uses multiple controller components, which usually turn only partially quiescent. Furthermore, they inform the other controllers of their intent to turn quiescent.

In the client/server reconfiguration pattern a client requests a service from a server. As soon as the service is finished, both components are allowed to change.

```
Configuror RobotConfiguror
  Style Dynamic-Robot
    new.arm:RobotArm
    → new.mainSensor:Sensor
    → new.reserveSensor:Sensor
    → new.link:RobotConnector
    → attach.mainSensor.distance.to.link.sensorValue
    → attach.arm.distance.to.link.transferredValue
    → Standard

  where
  Standard = (
    mainSensor.status.inactive
    → Style Dynamic-Robot
    detach.mainSensor.distance.from.link.sensorValue
    → attach.reserveSensor.distance.to.link.sensorValue
    → Alternative
  ) [] §

  Alternative = (
    mainSensor.status.active
    → Style Dynamic-Robot
    detach.reserveSensor.distance.from.link.sensorValue
    → attach.mainSensor.distance.to.link.sensorValue
    → Standard
  ) [] §
```

Listing 3.3: Wright notation of the robots reconfiguration.

# 4 Distributed Systems

Distributed systems are special kinds of software architectures, in which the different components execute on independent hardware. Components are able to communicate with each other using some kind of network infrastructure. Distributed systems bear additional challenges since components may fail unexpectedly at any time. Components of long-running applications may also require regular maintenance or functionality updates. Systems, therefore, need to be able to adapt flexibly to a changing environment. Dynamic reconfiguration mechanisms can help in defining and implementing these adaptations.

## 4.1 Achieving Robustness

In fact, most architecture description languages, as described in chapter 3 are intended to model distributed systems. But they usually rely on one component that controls the reconfiguration. For example, the robot of fig. 3.2 can already handle a faulty sensor and might be expanded to deal with a failing arm too, but its reconfiguration is controlled by the robot component. If the computer that runs this component has to shut down temporarily, the complete robot will fail. Furthermore, most mechanisms allow reconfiguration only at certain times. But this is not very helpful since components in a more realistic environment can go down at any time.

To tackle this, every component essentially has to reconfigure itself [GMK02]. To achieve this, a designer describes the architecture by stating constraints the components have to meet to form the desired architecture. During runtime, each component is responsible itself, that it conforms to these constraints. Therefore, if external or internal events cause the creation or removal of components, the architecture will be incorrect temporarily, but after a while, the components will have rearranged themselves so that the architecture is valid again. To achieve this, every component contains a component manager and a configuration view, additionally to the implementation of its own behavior. The managers are responsible for reconfigurations of the architecture and for keeping their views consistent. They also communicate reconfiguration events with each other. The views are just graphs of the current architecture in which components are nodes and connectors are edges. To maintain a view, the respective manager will update it immediately if it is informed about architectural changes or if it performs reconfigurations itself.

If a component is spawned or deleted, all managers of the system will try to reconfigure the architecture so that every incoming port of its respective component is connected to a valid outgoing port. To evaluate which port is valid, the manager uses the constraints that specify the architecture. Since concurrent changes to the architecture can lead to race conditions, they use a distributed locking scheme to make sure only one manager applies reconfigurations at a time. For example, one manager might want to modify the architecture based on a view that is outdated, because another manager just changed the configuration.

This describes a solid decentralized distributed system. Its architecture, however, is flat. This affects that time needed for reconfiguration increases with the size of the architecture because the component managers are not allowed to work concurrently. This may be necessary but can be reduced because in large architectures there will certainly be some managers that can work independently from each other. Adding a hierarchy to the structure might therefore be helpful. Another downside of this approach is that it does not describe how it prevents losing data while redirecting connectors.

## 4.2 Replacing Components

Apart from the point that distributed systems have to put more effort in combating partial failures than non-distributed systems have, the replacement of components also has a special relevance. Replacement means removing one component from the system and adding another similar one. There are two main reasons for replacing a component. Firstly, the need to transfer a part of the application from one piece of hardware to another one may occur [HP93]. Secondly, developers may have created a new, better version of a component, that has to be deployed while keeping the system active [VVE16]. Although replacement can simply be modeled as an addition following a deletion, it is generally not desirable to do so. This kind of reconfiguration usually has an impact on the running program because the interface of the corresponding component will be temporarily unavailable. As a result of that, the affected components' behaviors have to contain certain exception handling strategies. In the following, we present two different approaches that aim at avoiding this unnecessary overhead.

An approach that deals with the transfer of components to other hardware is described by [HP93]. They let users define certain points in the code at which the system may reconfigure. If a component has to move to another computer, the reconfiguration will not happen until the program reaches such a point. Then the component saves its current state, which is then used to redeploy the component at its new location. After that, all involved connectors are redirected to allow the moved component to communicate correctly. Then the program can finally continue running usually, without even noticing that its structure has changed. The authors argue that periodically saving the states of the components instead of saving it just when it needs to be saved, may make the system less vulnerable to failures of some single components. It does, however, introduce great overhead as the saving the states frequently costs time.

An alternative that is slightly more powerful because it does not rely on interrupting the application, is offered by [VVE16]. The connectors of the architectures they model are streams of data. Since replacing a component includes removing the component itself as well as all of its connectors, simply replacing the component without interrupting the architecture, would cause losing all data that is currently processed by the component or streaming towards or away from it. Even if the data leaving the component might still be able to reach its target, there is a need for a more considerate approach. The authors accomplish this by letting the old component process for a while after the new component has been deployed and activated. That means, two versions of the same component are active for a short time. Since this can cause inconsistencies, the components have to stay synchronous to each other, while they are coexisting. Hence they must have the same state. So there is a global state which is used by both components. As soon as the old component has finished processing all data that was still flowing towards it, it can be

removed and the system will run normally again. The reconfiguration process, therefore, looks as follows: A central reconfiguration manager starts the process by adding the new component to the architecture. It then instructs all components, that have been sending data to the old one, to stop doing so and communicate with the new one instead. Each of these components then appends a special marker at the end of the stream running to the old component. The markers represent the end of their corresponding stream and there will be no data afterward anymore. As soon as the old component has received all markers and does not have output anymore, the manager removes it and the process is finished. Note that the manager used here is different from the managers described in section 4.1. This manager is a central one. Hence this approach inherits the disadvantage of a central manager as described in that section. But this approach is not aiming at controlling partial failures. Furthermore, the use of a central controller keeps the time needed for reconfiguration at a stable low level. Both approaches, however, have in common that they allow reconfiguration without the need to delay until a certain reconfiguration point is reached like described in section 3.2.

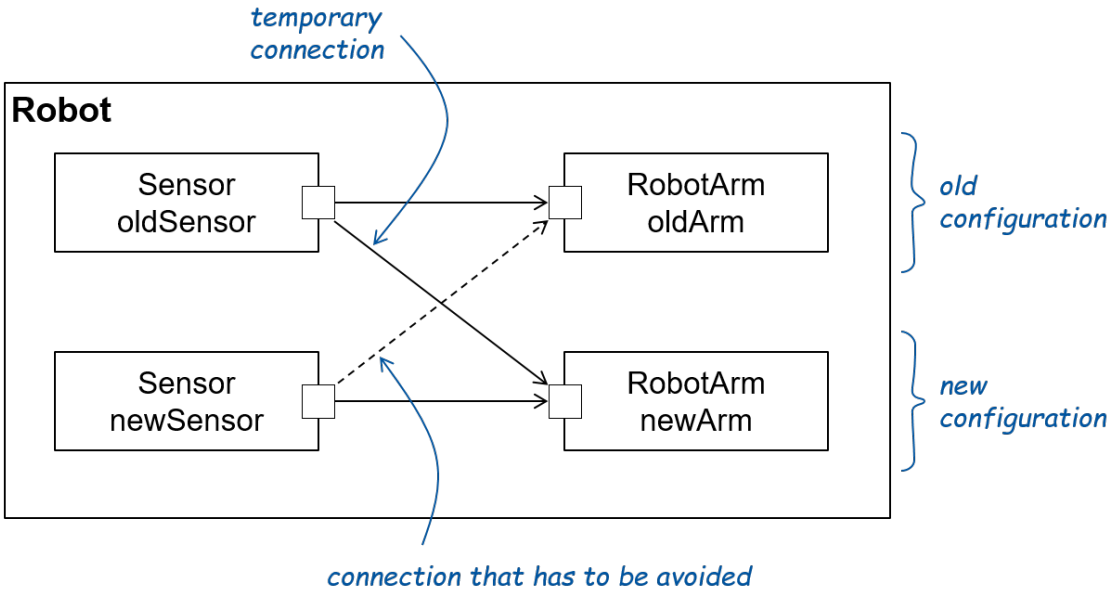


Figure 4.1: The new robot arm is the first to be deployed

This approach will be more complicated if the state of the component also has to be reconfigured. This may be necessary if the state has to be expanded to fulfill more functionality. In this case, the old state has to be translated to the new shape. Since the state is then used by both components, as described above, it has to be downwards compatible as the old state could not access it else. Further complexity is introduced if multiple components have to be replaced at the same time. Then the component that is farther down the line has to be redeployed at first, to make sure that its old version does not receive data from the new other component, which it might not be able to process correctly. For instance, fig. 4.1 illustrates a robot whose subcomponents are to be replaced. Here the arm has to be added before the new sensor is added because the arm receives data from the sensor.

# 5 Conclusion

To sum up, we have shown multiple different approaches to employing dynamic reconfiguration and their characteristics.

In the area of hardware, dynamic reconfiguration can increase the logic capacity of circuits. Due to the strict timing of circuits, the timing of reconfigurations is crucial. Depending on how it is used, dynamic reconfiguration can increase or decrease the demands for resources, like time and energy. It can also increase or decrease the complexity of hardware development.

Very different from that is the utilization of dynamic reconfiguration in software architectures. Here it mainly contributes to the flexibility of the system. Since a software application itself is already more flexible than a hardware one, the spectrum of possible usages of dynamic reconfiguration is much wider. Reconfigurations can be triggered externally or by the application itself, it can be controlled centrally or distributed, it can be hidden from the application or not. Designers have more options on how to model dynamic reconfiguration. And it may allow an architecture to grow into arbitrary configurations, or it may be restricted to allow only predefined changes whereas dynamic reconfiguration of hardware only allows the later one. Both applications of reconfigurations, however, have in common, that designers have to address the issue of timing the reconfiguration. While the duration of the process itself is not as relevant in software, as it is in hardware, determining when to start the process is also interesting in software architectures. In the best case, the reconfiguration can happen as soon as it is requested, without interrupting the running application. In the worst case, the process cannot start until all involved components have been turned off for the duration of the reconfiguration.

We have also shown, how dynamic reconfiguration enforces a more modular development approach and leads to the design of components with more loose coupling and higher cohesion in software and hardware architectures.

Finally, we have dealt with the software of distributed systems, which is just a special application of software architectures. Here dynamic reconfiguration proves to be a powerful strategy to combat partial failures of a system, which are likely to occur in distributed systems. Furthermore, it comes in handy when components of an architecture have to be replaced.

Dynamic reconfiguration is, therefore, a powerful tool to consider when developing any kind of software architecture, especially for distributed applications.

# Bibliography


[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, pages 21–37, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BML03] B. Blodget, S. McMillan, and P. Lysaght. A lightweight approach for embedded reconfiguration of fpgas. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 399–400, 2003.

[CCT99] Ernesto Pimentel Carlos Canal and José M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 107–125, 1999.

[GH04] Hassan Gomaa and Mohamed Hussein. Dynamic software reconfiguration in software product families. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 435–444, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. WOSS '02, page 33–38, New York, NY, USA, 2002. Association for Computing Machinery.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, volume 9839 of *LNCS*, pages 175–182. Springer, December 2016.

[HP93] C. Hofmeister and J. Purtilo. Dynamic reconfiguration in distributed systems: adapting software modules for replacement. In *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems*, pages 101–110, 1993.

[LCD+00] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, page 507–512, New York, NY, USA, 2000. Association for Computing Machinery.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering — ESEC '95*, pages 137–153, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[MVV+15] Nele Mentens, Jochen Vandorpe, Jo Vliegen, An Braeken, Bruno da Silva, Abdellah Touhafi, Alois Kern, Stephan Knappmann, Jens Rettkowski, Muhammed Soubhi Al Kadi, Diana Göhringer, and Michael Hübner. Dynamia: Dynamic hardware reconfiguration in industrial applications. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 513–518, Cham, 2015. Springer International Publishing.

[TCJW97] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 22–28, 1997.

[VEBD07] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007.

[VVE16] Rafael Oliveira Vasconcelos, Igor Vasconcelos, and Markus Endler. Dynamic and coordinated software reconfiguration in distributed data stream systems. *Journal of Internet Services and Applications*, 7(1):1–21, 2016.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling.* Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[WW90] David A Watt and Steven Wong. Programming languages. *Concepts and Paradigms Prentice Hall*, 1990.