Applications of scenarios in early embedded system design space exploration
van Stralen, P.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
This chapter is based on:


This thesis discusses the application of scenarios during the early embedded system design. In Part II of this thesis, we have looked at application scenarios that model the dynamism in the functional behavior of a multi-application workload. These application scenarios were utilized in order to obtain a static mapping of the multi-application workload onto the MPSoC such that the average behavior was as good as possible. The applications, however, are not the only dynamic part of the MPSoC. Another dynamic part is the MPSoC architecture where, for example, the architecture may be susceptible to faults or it allows for a reconfigurable set-up. A result of this dynamism in the architecture is that the non-functional properties (like performance and power) of an application mapping may change over time.

In this thesis, we will use the occurrences of faults in the architecture as an example of the dynamism in the architecture. Due to these faults, the output of applications that are mapped onto the architecture is not always correct. Take, for example, the binary sum 0001 + 0001. Normally, the solution of this sum would be 0010, but upon the occurrence of a fault one of the bits may flip. Therefore, on the flipping of the rightmost bit, the result can suddenly be 0011 instead of 0010. The potential presence of faults has a major effect on the embedded system design. As none of the outputs can be fully trusted, the reliability must be part of the design.

Next to the traditional objectives (e.g., performance, power and cost), reliability can be seen as an additional objective for an MPSoC design. To be precise, the reliability of an embedded system is the ability of the system to cope with errors. A higher reliability implies that the system has a greater ability to cope with the faults that are occurring in the architecture. These faults in the architecture can be of two
types: hard and soft errors. Hard errors (also called permanent faults) are permanent and once they have occurred, the affected architecture component becomes unusable. Soft errors (also called transient faults), however, are only causing temporal malfunction of the system. Examples of soft errors are Single Upset Event (SUE) [66] and Negative Bias Temperature Instability (NBTI) [1]. A SUE is a soft error where the electrical system is influenced by high-energy neutrons originating from cosmic rays that collide with particles in the atmosphere. NBTI, on the other hand, is caused by a degradation of the electrical circuit during its lifetime. Generally speaking, a soft error is a fault in one of the circuits (like processors, memory, communication, etc.) due to electrical noise or external radiation.

There are important difference between faults, failures and errors. At first, a fault is something like an incorrect step or data item in a process that leads to an unintended or unanticipated execution result of the process. A fault is a weakness in the process that might lead to a failure. This completely depends on the situation and how tolerant the application is to faults. Secondly, a failure, on the other hand, is the inability of the system to perform its task within the required performance requirements. This is undesirable and, unlike a fault, it unavoidably negatively affects the Quality-of-Service (QoS) of the system. Finally, an error is a discrepancy between the real and the computed output of a process. An error is caused by a fault and it may potentially be propagated in the process to result into a failure. Therefore, soft errors are, by definition, noticed.

It used to be the case that soft errors were not an issue for normal embedded systems, but only for avionics (electronic systems that are used in aircrafts, satellites and spacecrafts). With the improvements in technology, however, it has become also an issue on the ground level. This is illustrated in the graph of Figure 6.1 that shows the modeled number of faults for the different technology scales [66, 65]. On the
horizontal axis the technology is shown, whereas the vertical axis shows the soft error rate. This soft error rate is described using the number of Failures In Time (FIT). In a technology of 600nm a single SRAM chip (the memory part of a system) has roughly a FIT of $10^3$ (i.e., 3000 faults per 1 billion hours or a mean time to failure of $10^6$), whereas the logic (the computational part of the system) only has a FIT of $10^{-6}$. Therefore, for this technology scale it is quite obvious to make the memory storage reliable, but to ignore the soft errors that could occur during computation. However, with the decreasing technology scales, this relationship between memory and computation changed. When alternating the technology scales, the FIT rate is affected by a trade-off between a lower critical charge for the high-energy neutrons and the smaller device size. The critical charge is the minimal charge a neutron must have to be able to trigger a soft error. A smaller technology scale leads to a lower critical charge and, therefore, the FIT rate of the devices increase. On the other hand, a smaller technology scale leads to a smaller area of the device. As a result, the FIT rate decreases. For a memory the effect on the critical charge is relatively small and, thus, the memory benefits from the smaller area: from a technology scale of 600nm to 50nm the FIT rate stabilizes around $10^{-3}$. In case of a logic circuit, however, the critical charge drops much faster: the FIT rate grows from $10^{-6}$ to a FIT rate between $10^2$ and $10^3$. As a consequence, the fault rate in the logic computations becomes higher than the fault rate in the memory. Therefore, reliability becomes a first class citizen of the embedded system design and should definitely be taken into account when an embedded system is designed.

Figure 6.2 shows an example of how a single fault in the architecture can affect the output of a mapped application. The left column shows a situation where no fault is present and all the architectural components work flawlessly. In the right column, however, one of the CPUs is affected by a fault. Depending on the way the application is implemented, this may have a significant effect on the correctness of the behavior of the application. The first row shows an unmodified application. It is a pipeline of three processes (A, B and C) that operate on a single stream of data. In case the architecture is free of any faults, the output will be correct and in time (Figure 6.2a). The situation in the second architecture scenario is somewhat different (Figure 6.2b): the CPU on which process B is mapped is affected by a fault. A CPU that is in a faulty state cannot be trusted anymore. Therefore, the outcome of process B may be: 1) correct, 2) incorrect, 3) it may arrive later than usual or it may even never arrive at all. Depending on the situation, process C gets unreliable data or no data at all. As a result, the complete application becomes unreliable since it is not known if the output stream is correct or not.

In order to make an embedded system reliable, both hardware- and software-based techniques can be used to improve the reliability. Hardware-based techniques design an architecture component in such a way that it is more resilient to soft errors by, for example, using replication of architecture components [20]. The software-based techniques apply similar techniques purely in software. An example of a
Figure 6.2: An example of how a single fault affects a normal application and a fault-tolerant application using triple modular redundancy (TMR).
software-based technique is active redundancy in space or time. Active redundancy in space uses different resources to run the same task. After running the tasks, their outputs are compared using a majority voter. In case the value of one of the outputs deviates from the value of the majority of the outputs, there is assumed that the majority has the correct output value. In Figure 6.2c an example of active redundancy in space is given: *Triple Modular Redundancy (TMR)*. TMR uses three different replicas to execute the same task. When there are no faults, the behavior of the application is almost similar as the normal application. The only difference is the overhead of the majority voter in the TMR. In contrast to the normal application, however, it can tolerate a single fault. This can be seen in Figure 6.2d. The processor on which the first replica of process B is mapped has a fault. Since the other two replicas are mapped onto correctly functioning processors, the TMR voter observes the same outputs for processes B2 and B3. Assuming that the processes have a deterministic behavior, the voter observes a majority for the correct output and this output can be streamed to process C. Active redundancy in time is similar, only in this case the replicas are running on the same architectural resources and are scheduled in a sequential fashion.

Implementing software-based reliability techniques has its pros and cons. As discussed earlier, a technique like TMR already adds some voting overhead to the execution time of a single application. Apart from that, it also requires more architectural resources than a normal application without replicas. Therefore, the non-functional properties (like execution time, energy and cost) of a normal mapping are highly affected by software-based reliability techniques to make an embedded system reliable. A software-based reliability technique may boost the reliability of the system, but other metrics like performance and power may be affected negatively. Therefore, the reliability of a system must be taken into account in the early phases of the design space exploration. It makes no sense to apply software-based techniques after the mapping has been optimized (hierarchical design). The non-functional properties of such an optimized mapping will be invalidated and better mappings may be identified when the reliable application is taken into account from the start on.

In this chapter, we introduce *Sesame Automated Fault-tolerant Explorer (SAFE)*. SAFE is an extension of Sesame to explicitly model the appliance of software-based reliability techniques to MPSoC based embedded systems. The simulation of SAFE will not only results in the evaluation of traditional non-functional properties of an embedded system like execution time, energy and cost, but also metrics like frame miss rate. In Section 6.1, a detailed description is given of the application model that is required as an input for SAFE. Next, Section 6.2 discusses the way fault tolerance is modeled within the KPN, followed by Section 6.3 that introduces the fault-tolerant mapping. The extensions to the simulation model of Sesame are given in Section 6.4. Finally, the chapter is concluded by a set of experiments, related work and a short conclusion.
6.1 IO Modeling

Up to now, the MPSoC designs were optimized to be as fast as possible. However, regularly the perceived output by the user is of more importance than the total running time of an application. In case of a multimedia application, like a video player for example, the user wants a timely display of the frames. This means not too late, but also not too soon. Take, for example, a frame buffer of a computer monitor. A frame buffer of a computer monitor should not be overwritten before the frame is shown. For the fault-tolerant optimization it is even more important to be able to reason about timely delivery of frames. While searching the design space of fault-tolerant embedded systems, we are actually trying to find out how many and which reliability techniques can be applied without significantly affecting the user’s experience. Therefore, in order to be able to model frames and to capture frame rate in scenario-aware Sesame, we have made two extensions: 1) Frame barriers and 2) IO modeling.

Explicit frame barriers can be modeled by using an intra-application scenario as a single frame. As was discussed in Section 3.1, an intra-application scenario is encapsulated between a StartScenario and an EndScenario event. The discrete event simulator of the MPSoC architecture will process these events and, as a result, the architecture is able to get statistics on frame level. Examples are: the frame rate of an application or the number of missed deadlines (given a predefined frame rate). Obtaining these metrics does not mean that we are able to provide any guarantees about real time behavior. The aim of Sesame is to prune the design space and to identify interesting mappings with respect to several objectives like time and energy. This is also the case for the frame rates that are obtained from our simulator. Good mappings will be identified, but in order to provide any guarantees a more detailed (and probably more computationally expensive) analysis must be done on the pruned design space.

Another aspect required by SAFE is IO modeling. Where the frame barriers are purely meant to be able to reason about a good mapping with a timely delivery of frames, IO modeling makes a clear distinction between data that is visible to the user and data that is invisible to the user. For this purpose, the notion of an Outside World Process (OWP) is introduced that provides an interface between the application and its outside world. All output that is visible to another user must be communicated to an OWP process. Similarly, all external input is obtained by reading from an OWP process. The separation between an OWP process and a normal process also makes the fault tolerance of the application more transparent. Data coming in from a OWP process is guaranteed to be correct (as result of the fault-tolerant communication network), but the data that is sent to an OWP process for output must be verified to make the application fault tolerant. Any other communication between normal processes does not need to be verified since it is not visible to the user.

The OWP process also provides explicit knowledge of the moment that the data is sent to the user, which allows for exactly obtaining the frame rate. A frame is started
after the first external data is read using an OWP process and it is finished after the last block of data is written to an OWP process. Combined with the workload description that was introduced in Chapter 3 (see Figure 3.3 on page 39), it can be determined how long it takes to process a frame and if the given deadline was met.

Such an outside world can manifest itself in many ways. It can be a storage device that is read by another system, but it can also be a monitor. For the application layer, these different possibilities have a similar interface, but there may be different architectural components to model the IO for storage devices or monitors. Therefore, a part of the mapping is the binding of the IO components to a suitable architectural component.

Figure 6.3 shows an example of a Sesame model that is extended with IO support. It is a Motion-JPEG (MJPEG) decoder (Figure 6.3a) with a four-processor architecture that uses a shared memory (Figure 6.3b). At first, the OWP0 process reads the encoded frames from an input source like a disk or an external connection. Depending on the XML workload description, all frames can be available at once (i.e., the rate of incoming frames is only bound by the buffer sizes of the communication channels) or the frames can arrive with a certain interval. Secondly, the OWP1 process will write the decoded frames to the output device. All the processes of the application need to be mapped onto the architecture, both the normal and the OWP processes. The normal processes can be mapped on any of the processors, but OWP0 can only be mapped on the Camera component. Similarly, OWP1 must be mapped on Disk that is an IO component capable of modeling the writing to memory and to analyze if all the frames are delivered in time.

Although it is easy to obtain the frame rate of the MJPEG example, one must be careful with the used application scenario workload. In contrast to an application with a single process, a KPN has exposed parallelism and, therefore, the KPN potentially supports pipelining. That means that different parts of the application may work on different frames simultaneously. For the example MJPEG application, this means that while the OWP1 is still working on the first frame, the OWP0 process may already have been started with processing the third frame. In order to illustrate this, Figure 6.4 shows experimental results from the MJPEG application where the mapping utilizes all of the four processors and all frames are immediately
Figure 6.4: The frame processing time of a MJPEG application.

available. The horizontal axis shows the number of the processed frames and the vertical axis shows the corresponding cycle time for both the frame latency and the frame interval.

The frame latency is the number of cycles that it takes to process the complete frame from the start in the OWP0 process until the end in the OWP1 process. Until the eighth frame, the frame latency increases when the number of processed frames increases. In this warm up phase the system makes a cold start where the pipeline is empty and, over time, the pipeline becomes saturated. In case all of the processes were mapped to different processors, the frame latency would be less dependent on the saturation of the pipeline (there still is a shared communication network), but in our example there are only two processors for eight processes. As a result, the processors and the communication link are shared between the different processes and multiple frames may be competing for the same resources. After eight frames, the frame latency stabilizes until no more new frames are read in from disk. During this cool down period, the pipeline empties and the resource contention on the processors and the shared bus drops.

To determine the frame rate the frame interval is used. This is the difference between the arrival time of the frame that was processed previously and the arrival time of the current frame. When determining the frame rate only the saturated case is realistic. For the first frame the frame interval is still relatively large, but once the system is heated up every 333k cycles a new frame is finished. It is not realistic to use the interval of the last frame as a frame rate. The latency of 113k would result in a frame rate that is only valid during the cooling down of the application. To obtain realistic frame rates, an automatic warm-up procedure is used. As long as the processing time of a frame keeps increasing the frames are discarded for the frame rate calculation. After the first frame with a non-increasing processing time the frame rate calculation is started. To prevent the cooling down effect, frames keep being added until the simulation is halted.
6.2 Fault-Tolerant KPN Model

The experiment of the previous section showed a perfectly constant frame latency during the lifetime of the MJPEG application. This may well be the case for reliable architectures, but once the architecture is considered to be unreliable, the latency may fluctuate over time. In order to minimize these fluctuations, the application could be made fault tolerant.

This fault tolerance is important as faults may arise any time and at any place. As a starting point, we assume that the communication network is already fault tolerant. This assumption is necessary to provide a base for the software-based reliability techniques. Without it, we can, for example, not even be sure if the data that is sent from the IO component to the source node of the application is still valid. Apart from that, it is perfectly doable to implement the communication network in a fault-tolerant manner [54].

SAFE will automatically search for a good fault-tolerant implementation of the computation of the application. For this purpose, fault tolerance patterns are used to make the application reliable. A fault tolerance pattern is a software or hardware-based technique to make the application fault tolerant. Earlier we already have seen the software-based technique **Triple Modular Redundancy (TMR)**. **Double Modular Redundancy (DMR)** is related to TMR, only DMR uses two replicas instead of three. Hardware-based fault tolerance patterns do not require any modification to the application; they only affect the architectural component(s) on which the application processes are mapped. Software-based techniques, on the other hand, may require some transformations to the application. This modification is done automatically by SAFE.

As an example, take our example of Figure 6.3. We could make a fault-tolerant implementation by applying the active redundancy technique DMR as shown in Figure 6.5. DMR performs the computation twice and compares the outcome. In this case, the application transformation (Figure 6.5a) needs to introduce two additional components: a splitter to distribute the incoming data from the OWP process to all the replicas and a majority voter to combine the results of the different replicas such

---

**Figure 6.5:** An example MPSoC model extended for active redundancy.
that verified data can be sent to the OWP process. Additionally, the application needs to be duplicated such that two replicas will be available. The usage of majority voting implies that only deterministic processes are supported. That means that, given error-free execution, that the same input will always generate one unique outcome. Majority voting also assumes that an error is always random. The probability that exactly the same error is made by the majority of the components should be almost zero.

The DMR also requires additional architectural components. These components are shown in Figure 6.5b and are emphasized using a gray fill. At first, an architectural component is required that is capable of providing support for the splitter and the voter. The voter must compare the outputs of the different replicas. If there is already a majority for a certain output before all different outputs are in, the resulting output can already be forwarded. In this way, the application can also tolerate if a single output does not arrive at the voter at all. The component that both provide the splitting and the voting is the NMR (an abbreviation of N-modular redundancy) component. As the architecture is unreliable, this NMR component must already be implemented in a fault-tolerant manner.

Having a NMR component that provides facilities for splitting and voting is insufficient. Whenever a voter does not observe a majority on the obtained outputs (either by corrupted data or a time-out on one or more ports), the faults that are detected cannot be masked and in this situation there are basically two options: restarting and skipping. Skipping will discard the rest of the frame and the computation is resumed with the next frame. As SAFE focuses on soft real time systems, a low number of skipped frames can still lead to a decent QoS. When too many frames are skipped, however, the QoS quickly drops. In this case, restarting may help. Restarting will retry the decoding of the frame, but to assure that the input data is still present (it may be IO that is only temporarily available) the input data must be cached. As the architecture is unreliable, this is done in Stable Storage (SS). Not only the splitter has stable storage, but the voter and processors may also have stable storage. The presence of stable storage for the voter and the processors may allow the use of checkpoints as will be discussed later on.

Although this discussion will focus on active redundancy as a software-based reliability technique, other techniques are also possible. As checkpoints are modeled in SAFE (as will be discussed later), a technique like Roll Forward Checkpointing Scheme (RFCS) is also possible. RFCS is similar to DMR, only RFCS activates a spare replica to break the tie when a majority is absent. Other techniques can easily be added as fault tolerance patterns are defined separately as transformations on an existing application. Therefore, on the introduction of a new fault tolerance pattern it can, without further modifications, immediately be applied to all the existing applications.
6.2.1 Patternization

A fault tolerance pattern does not necessarily need to be applied to a complete application. This technique is just one extreme in the potential ways an application can be made fault tolerant. Another extreme is to apply fault tolerance patterns individually to the application processes. If, for example, on each individual process DMR is applied then each process has its own splitter and voter. In comparison to a voter that is used for a complete application, this approach may be able to intercept a fault earlier, but it also involves significantly more overhead. Therefore, a DMR per process is not necessarily beneficial for the quality of an application. In between these solutions, there are many more solutions that leverage the overhead by dividing an application network into an arbitrary number of subnetworks on which fault tolerance patterns are applied.

Hence, to make an application fault tolerant, the application is segregated into a number of subnetworks. On each of these subnetworks, a fault tolerance pattern is applied. The transformed application in Figure 6.5a, for example, has only a single subnetwork that is equal to the complete KPN. If more than one subnetwork is used, there are some requirements on the subnetworks. First, the complete set of subnetworks must cover the complete application and none of the subnetworks may overlap. It is absolutely crucial that the complete application is covered. When in the transformed application in Figure 6.5a, for example, the DMUX would not part of a subnetwork, the data that is processed by the rest of the application cannot be trusted anymore. A fault of the DMUX cannot be detected and, therefore, other processes can potentially operate on faulty data. Consequently, the input(s) of a subnetwork must always be verified.

Figure 6.6 gives an example of the segregation of an application. The application, as shown in Figure 6.6a, can be segregated in 15 different ways. Not all of these segregated networks, however, are valid. In our example, a subnetwork that is only consists process A and D is invalid. Since these two processes are not connected, a subnetwork like this can be considered as being two separate subnetworks. If, for example, a soft error results into a fault of process A, it may be decided to restart the frame. As process D is not directly connected to process A, it is useless to restart D as it is not dependent on process A, nor does it provide input to process A. Therefore, the subnetwork must be a weakly connected graph (i.e., if the directed edges are replaced by undirected edges there is a path between each pair of vertexes). In Figure 6.6b, a segregation is given where there are three subnetworks: A, BC and D. The idea of segregating the graph implies that for each individual subnetwork a fault tolerance pattern is applied. Based on the pick of the fault tolerance patterns, the application can be transformed into a fault-tolerant application. For our example, one of the potential fault-tolerant graphs is shown in Figure 6.6c. In this case, DMR is applied on subnetwork BC and TMR is applied to the subnetworks A and D.

The example illustrates that there are not only many ways to segregate the application, but there are also many ways to apply fault tolerance patterns to a segregated
application. If only DMR and TMR would be used, there are not less than 82 different fault-tolerant graphs for our toy application. For a real application with more processes, the search space of potential fault-tolerant application graphs even grows exponentially with respect to the number of processes and fault tolerance patterns. The combination of segregating an application and assigning fault tolerance patterns to the individual subnetworks is called patternization.

By exploring the design space of potential patternizations, a trade-off can be made between the overhead of fault tolerance and the QoS of the MPSoC design. The granularity of the subnetworks can be manipulated, where a single process per subnetwork is the highest granularity and a single subnetwork per complete application is the lowest granularity. Apart from that, the used fault tolerance pattern can be varied. The more extensive fault tolerance patterns check for the validity of the computed outputs, the more reliable the design becomes. However, this likely comes at a price of a higher overhead like computation and / or communication overhead that needs to be performed to implement the fault tolerance patterns. To automatically explore these options an automated way must be developed to describe the patternizations of an application.

Figure 6.6: An example of application segregation in order to apply active redundancy on parts of the application instead of applying it to the complete application.
6.3 Fault-Tolerant Mapping

SAFE extends the Sesame model (see Section 2.1) to support patternization by adding an additional layer: the pattern layer. To make a clear distinction between the functional behavior of the application (i.e., the application layer) and the fault tolerance behavior, the pattern layer describes the fault tolerance patterns that can be used to make the application fault tolerant. These patterns are described using an application transformation such that they are completely independent of the used application. In this way, a database of possible fault tolerance patterns can be maintained for usage in different projects.

The mapping layer extends the mapping of the original Sesame model. Before the application tasks are mapped to the architecture, the fault-tolerant mapping first performs the complete patternization by segregating the applications and transforming their application graphs using the selected fault tolerance patterns. All the tasks in the transformed application(s) can then be mapped onto the architecture.

In this section a formal description is given of the fault-tolerant mapping. The implementation details of SAFE will be discussed in later sections. At first, Subsection 6.3.1 formally describes the SAFE system model, after which the next subsection will formally describe the fault-tolerant mapping. To illustrate the fault-tolerant mapping, the MJPEG example of Figure 6.5 is used.

6.3.1 System Model

The main purpose of the SAFE system model is to transparently integrate the fault tolerance into the early DSE of MPSoC based embedded systems. In this subsection the SAFE model, as shown in Figure 6.7, will be described formally. The most
important mathematical definitions used in this subsection are also shown in their corresponding layer in Figure 6.7.

**Application layer** The application layer describes the functional behavior of the application. This is the pure behavior, not the extensions that are made to make the application fault tolerant. As an application within SAFE is represented by a KPN, the applications are modeled by a directed graph $G_K(V, E_k)$. The vertexes $V$ represent the processes and the edges $E_k = V \times V$ represent the communication channels. To be able to make the application fault tolerant, a distinction is made between two types of processes: the normal processes $V_N$ and the outside world processes $V_{OWP}$. A process can only be one of the types: $V_N \cap V_{OWP} = \emptyset$. From these types of processes, the OWP processes are the only processes that are allowed to interact with the world that is external to the application (like reading from a disk that is writable by other systems or displaying an image on a monitor). They are not allowed to perform any computation. Only the normal processes can do computation. For this reason, it is not allowed for OWP processes to communicate directly with each other:

$$\forall v_1, v_2 \in V_{OWP} : (v_1, v_2) \notin E_k$$  \hspace{1cm} (6.1)

**Architecture layer** In the architecture layer the structure of the architecture, as well as the non-functional behavior, is described using a directed graph $G_R(R, E_R)$. In this case, $R$ represents the available architectural resources of the MPSoC. Among these resources, there are multiple dedicated resources like processors $R_P \subset R$, IO elements $R_I \subset R$ and reliability elements $R_R \subset R$. The way in which the different architectural elements are connected is described using the edges in $E_R = R \times R$.

Processors $R_P$ are components that can be used to map normal process nodes. The IO elements $R_I$, however, are only meant for OWP processes. This means that, in contrast to processors, they do not perform any processing, but they only perform input and output tasks for the mapped applications. Next to the processors and IO elements, there are also reliability elements $R_R$. These are the main addition to the architecture that allows us to automatically add support for fault tolerance in MPSoC design. Conceptually, each reliability element consists of a splitter and a voter. The splitter takes care that the incoming data is available to all the processes in the fault-tolerant subnetwork. Not only does this involves sending data to different replicas, but it also may be required to cache the input data such that the processes can be re-executed. Another part of a reliability element is the voter that is responsible for detection of faults and handling them. Every data item that passes the voter is verified by the corresponding fault tolerance pattern. Therefore, each of the reliability elements is linked to one or more fault tolerance patterns. An example is a fault tolerance pattern where reliable computation is used. Reliable computation can only be done using a fault-tolerant processor that has built in support for detecting and handling faults. It does not need a splitter or voter and, therefore, conceptually the
splitter and voter of this reliable component just forward the data. DMR, on the other hand, has a splitter that sends the data to the two replicas and a voter to do a majority vote on the outputs of these replicas.

Referring to our example of Figure 6.5b, all of the shown resources are part of the architectural resources $R$. From the architectural resources, PROC-1, PROC-2 and PROC-FT are part of the processors $R_P$, whereas Disk and Display belong to the IO elements $R_I$. In the architecture there are two reliable elements ($R_R$): the NMR that implements active redundancy and PROC-FT that is an implementation of a fault-tolerant processor. This example also shows that it is perfectly possible that the same component belongs to two types of processors: PROC-FT both provides the ability the execute processes as well as it provides reliability.

**Pattern layer** The pattern layer describes the transformations that need to be done to make a subnetwork fault tolerant and to which architectural resources the fault-tolerant subnetwork can be mapped on. All the possible fault tolerance patterns are gathered in collection $F$. An example of fault tolerance pattern $f \in F$ is active redundancy [13] where the processes are executed multiple times and the outcome is compared. Such a fault tolerance pattern does not only describe the reliability technique. In case of the active redundancy, for example, action must be taken in case a fault is detected. Next to that, a fault tolerance pattern may involve taking checkpoints to be able to re-execute the code at arbitrary points in time. Therefore, parameters like the action to be taken on the detection of faults and the frequency of checkpoints (see Section 6.4.3) are also part of a fault tolerance pattern.

As an example take the three fault tolerance patterns DMR, TMR and FPROC. The DMR and TMR are active redundancy techniques, whereas the FPROC is a fault-tolerant processor. Each of the fault tolerance patterns uses a different number of replicas. Therefore, a function $n_{\text{proc}}(f)$ is defined that describes the number of processors that are required for fault tolerance pattern $f \in F$. In our MJPEG example, the function $n_{\text{proc}}(\text{FPROC})$ returns 1, whereas $n_{\text{proc}}(\text{DMR})$ and $n_{\text{proc}}(\text{TMR})$ are, respectively, 2 and 3.

**Mapping layer** Finally, the mapping layer glues the application, pattern and architecture layers together. For this purpose, three types of edges are used: patternization edges, mapping edges and IO edges.

**Patternization edges** $E_P$ describe both the segregation of the application and the selection of the fault tolerance patterns for each subnetwork. Each edge $(v, f) \in E_P$ represents a possible appliance of the fault tolerance pattern $f \in F$ for the process $v \in V_N$. That means that all the processes that are connected to the same fault tolerance pattern $f$ belong to the same subnetwork. A consequence of this approach is that the same pattern must be present multiple times in the collection $F$ in case the same pattern must be available for multiple subnetworks. This duplication is done automatically during the initialization of SAFE such that each individual process
can use the same pattern in its own subnetwork. The user, however, may limit the number of duplications of a fault tolerant pattern.

**Mapping edges** $E_M$ assign the architectural resources to the application-level components that will be used by a fault tolerance pattern. More precisely, the edge $(f,r) \in E_M$ assigns architectural element $r \in R_P \cup R_R$ to pattern $f \in F$. Not all mapping edges are valid:

\[
(f,r) \in E_M \iff \text{feasible}(f,r) \quad (6.2a)
\]

\[
\forall f \in F : |\{r|(f,r) \in E_M \land r \in R_R\}| \geq 1 \quad (6.2b)
\]

\[
\forall f \in F : |\{r|(f,r) \in E_M \land r \in R_P\}| \geq 1 \quad (6.2c)
\]

Depending on the fault tolerance pattern, other types of architectural resources may be used for the mapping of a transformed fault-tolerant application. For our MJPEG example, the feasible function is shown in Table 6.1. The first two columns show the two fault tolerance patterns DMR and FPROC. A DMR makes the execution reliable by voting on the results using the NMR component. Therefore, its replicas can be mapped onto the generic processors PROC-1 and PROC-2. For the FPROC pattern, on the other hand, only a special fault-tolerant processor PROC-FT can be used. In general, each fault tolerance pattern must have at least one voter component on which it can be mapped (Equation 6.2b) and one processor component (Equation 6.2c).

Next, the **IO edges** $(E_{IO})$ bind the OWP processes to an IO component in the architecture. To be precise, an IO edge $c = (v,i)$ assigns the OWP process $v \in V_{OWP}$ to an IO component $i \in R_I$ in the architecture. Similarly to the mapping edges, the IO edges also must comply with the feasibility requirements. As there are many types of IO, there also are a large potential number of architectural components to map onto. The feasibility requirement is as follows:

\[
(v,r) \in E_{IO} \iff \text{feasible}(i,r) \quad (6.3a)
\]

\[
\forall v \in V_{OWP} : |\{r|(v,r) \in E_{IO}\}| \geq 1 \quad (6.3b)
\]

<table>
<thead>
<tr>
<th>feasible(e, r)</th>
<th>e = DMR</th>
<th>e = FPROC</th>
<th>e = OWP0</th>
<th>e = OWP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r = \text{PROC-1}$</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$r = \text{PROC-2}$</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$r = \text{PROC-FT}$</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$r = \text{NMR}$</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$r = \text{CAMERA}$</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>$r = \text{DISK}$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: The values for the feasible function for the MJPEG example in Figure 6.5. In this table ‘✓’ is True and ‘−’ corresponds to False.
This requirement enforces that there is at least one potential mapping for any of the OWP processes. Without a potential IO edge there would not be a valid fault-tolerant mapping. The feasible IO edges of the MJPEG example are shown in Table 6.1. The last two columns show the OWP processes OWP0 and OWP1. Each of them complies with the requirement that there is at least one IO edge (Equation 6.3b), namely, CAMERA for OWP0 and DISK for OWP1.

Mapping the transformed application graph and the OWP processes is not sufficient. Due to adding processes and replicating processes while transforming the application graph, there will be more messages in the embedded system than in the original application. All of these messages need to be dispatched, where special care must be taken if a message need to be sent to multiple destinations (the splitter) or when multiple messages needs to be sent to the same destination (the voter). Therefore, a final step in the mapping layer is the message dispatch that determines the destination of individual messages that are sent.

Let \( Q = V_N \times V \) be the set of requests. A request is done by a process node and is an act to ask for reading or writing data to another process node. Only after the destination confirms the request, the read or write can be handled. Read requests can only be handled if the data is present, whereas write request require available buffer space. The normal process node \( v_s \) initiates the request \((v_s, v_d) \in Q \) (An OWP process is passive and can therefore not initiate a request) and sends it to the other process \( v_d \). There are two types of requests: one set of read requests \( Q_{\text{READ}} \subseteq Q \) and one set of write requests \( Q_{\text{WRITE}} \subseteq Q \). These sets of requests need to comply with the following requirements:

\[
Q = Q_{\text{READ}} \cup Q_{\text{WRITE}} \tag{6.4a}
\]
\[
Q_{\text{READ}} = \{(v_s, v_d) | (v_d, v_s) \in E_k \land v_s \in V_N\} \tag{6.4b}
\]
\[
Q_{\text{WRITE}} = \{(v_s, v_d) | (v_s, v_d) \in E_k \land v_s \in V_N\} \tag{6.4c}
\]

Per communication link a single request is added, where Equation 6.4b is for a read request and Equation 6.4c is for a write request. Process \( v_s \) is the process node that initiates the request and process \( v_d \) is the node where the request is sent. Both of the read and write request must be done from a normal process node \( v_s \in V_N \). The difference between the requirement of the read request (Equation 6.4b) and the write request (Equation 6.4c) is the direction of the communication link. For a write request, the communication link starts at the requesting nodes and ends at the destination node \((v_s, v_d) \in E_k\), whereas for a read request the communication link is reversed: it starts at the destination process and it ends at the requesting node \((v_d, v_s) \in E_k\).

As SAFE deals with fault-tolerant application networks, a request is not sufficient to exclusively define a single message. A node may be replicated and, hence, the same requests may be initiated from different locations. For that reason, a message is the
combination of a request and a location: $M = Q \times R$. Based on the request and the location, the dispatch edges $E_D$ determine the target architectural element for the request. When $(m, r) \in E_D$, message $m \in M$ needs to be sent to architectural resource $r \in R$.

### 6.3.2 Mapping Procedure

The previous section introduced the system model of SAFE. A next step is to use this system model to map the applications onto the architecture. During such a mapping the fault tolerance must be taken into account. For this purpose, the Y-chart approach [38] that is used by Sesame must be extended to incorporate the fault tolerance patterns. This modified approach, as shown in Figure 6.8, has three inputs: 1) the applications, 2) the fault tolerance patterns and 3) the architecture. Due to its three inputs it looks more a Ψ-chart than a Y-chart, but the concept is the same. With the application, fault tolerance patterns and the architecture, a fault-tolerant mapping is made. Using SAFE, the performance of the fault-tolerant mapping is obtained using simulation and based on these results the fault-tolerant mapping can be optimized. For our Ψ-chart, this means that the applications, the fault tolerance patterns and the architecture can be changed, but also that a different fault-tolerant mapping is used.

This subsection focuses on the fault-tolerant mapping that not only performs the binding of the applications onto the architecture, but before binding it also transforms the applications into fault-tolerant applications. More specifically, there are three steps: 1) patternization ($\psi$), 2) binding ($\beta$) and 3) message dispatch ($\delta$). First, the patternization splits the application into subnetworks that each gets their own fault tolerance pattern. Next, the binding binds the components of the fault-

![Figure 6.8: The modified Y-chart for synthesis with fault tolerance support.](image-url)
Patternization

A fault-tolerant mapping starts with a patternization that divides the applications into subnetworks by selecting fault tolerance patterns for each individual process. Formally, the patternization $\psi$ contains one pattern edge for each normal node in $V_N$ such that $\psi \subseteq E_P$:

$$\forall v \in V_N : |\{ f | (v, f) \in \psi \}| = 1$$  \hspace{1cm} (6.5)

Only normal nodes need to be taken into account during patternization. OWP processes do not perform any processing, but only communication. Since, we assume that the communication is already fault tolerant, OWP processes do not need to be guarded. Implicitly, this assignment of fault tolerance patterns $f \in F$ divides each application into subnetworks that use the same architectural support for their fault tolerance. The definition of a fault-tolerant subnetwork ($G_f$) with fault tolerance pattern $f \in F$ is as follows:

$$G_f := \{ v | (v, f) \in \psi \}$$  \hspace{1cm} (6.6)

$$\forall v_1, v_2 \in G_f : \text{weakly_connected}(v_1, v_2, E_k)$$  \hspace{1cm} (6.7)

As discussed before, performance wise it is important that the processes in the subnetworks are weakly connected (which is enforced by Equation 6.7). If this would not be the case, processes may, for example, be unnecessarily restarted after a transient fault of an unrelated process in the subnetwork.

Figure 6.9 shows one of the potential patternizations of the MJPEG application. Basically, the six normal processes are split into two subnetworks $G_{DMR}$ and $G_{FPROC}$. The first subnetwork $G_{DMR}$ uses the DMR technique that makes two replicas of the subnetwork \{DMUX, RGB2YUV, DCT\}. For the other part of the application, the subnetwork $G_{FPROC}$ uses a specialized fault-tolerant processor to execute the processes Q, Control and VLE in a reliable manner. If the complete patternization
ψ would be written down, the result would be as follows:

\[
\psi = \{(\text{Control, FPROC}), (\text{DCT, DMR}), \\
(\text{DMUX, DMR}), (\text{RGB2YUV, DMR}), \\
(\text{Q, FPROC}), (\text{VLE, FPROC})\}
\]

**Computational Binding** Conceptually, the computational binding maps the transformed fault-tolerant application graphs onto the architecture. However, this mapping is not done per individual process. The reason for mapping one fault-tolerant subnetwork at the time is that some fault tolerance patterns require the different processes to be on the same processor to work efficiently. Take, for example, explicit checkpoints (see Section 6.4.3) where all the processes of a subnetwork must be halted before the checkpoint can be taken. If the replica processes are on the same processor (the other replicas may still be mapped on a different processor), the local checkpoint procedure will not require any external communication.

The computational binding \(\beta_x\) will map the fault-tolerant subnetworks onto two types of architecture components. At first, a voter is selected for splitting and combining the in and outgoing data. Recall that for some fault tolerance patterns these components will simply forward the data (like the fault-tolerant processor that does not need to split and combine the data). Voters and splitters are, just like replicas, provided as a general description of a fault tolerance pattern. Based on this general description of a fault tolerance pattern, a general mapping procedure can be defined. The real implementation of voters and splitters, however, may be an empty function. Secondly, a fault-tolerant mapping requires the selection of a set of processors that is used to run the individual processes. To be precise, the computational binding \(\beta_x \subseteq E_M\) uses mapping edges to select architecture components for the subnetworks:

\[
\forall f \in F : G_f = \emptyset \lor (\text{HAS\_VOTER}(f) \land \text{HAS\_REPLICA}(f)) \tag{6.8a}
\]

\[
\text{HAS\_VOTER}(f) := |\{r| (f, r) \in \beta_x \land r \in R_R\}| = 1 \tag{6.8b}
\]

\[
\text{HAS\_REPLICA}(f) := |\{p| (f, p) \in \beta_x \land p \in R_P\}| = n_{\text{proc}}(f) \tag{6.8c}
\]

These equations make certain that architectural resources are selected for processing the tasks and a voter element is selected to make the outcome reliable. The capacities of these components, however, are not taken into account during the creation of the fault-tolerant mapping. A fault-tolerant mapping can, for example, map ten fault-tolerant subnetworks onto the same NMR component. It depends on the NMR component if it really has the capabilities to support ten fault-tolerant subnetworks. The simulation verifies if a fault-tolerant mapping adheres to the maximal capabilities of the architectural resources. As simulation knows exactly the data requirements of the applications, it can analyze if the voter capacity is exceeded.
An example of a computational binding is shown in Figure 6.10. As a computational binding cannot be defined without a patternization of the application, this example binding is based on the patternization in Figure 6.9. There are two subnetworks that need to be mapped: $G_{DMR}$ and $G_{FPROC}$. The first subnetwork $G_{DMR}$ uses the NMR component as a voter component and as replicas the processors PROC-1 and PROC-2 are used. Looking to Table 6.1, NMR is in this architecture the only potential voter that can be chosen for the DMR pattern. For the processors, however, there are multiple options. Where this example chooses to map each replica to a different processor, there can also be chosen to map the replicas onto the same processor. This leads to replication in time instead of in space. The mapping of the FPROC is even more restricted than the DMR pattern. Both the voter and the single replica (which is strictly speaking not a replica, but the original application subnetwork) can only be mapped onto one architectural component: the PROC-FT. This means that the same mapping edge is added twice to the computational binding: once for the voter and once for the application processes. The mapping of the voter is purely for defining a formal and generic model of a fault-tolerant mapping. The real implementation will not contain the voter (only a fault-tolerant processor that runs the original processes). Hence, the complete computational binding becomes:

$$\beta_x = \{(DMR, \text{PROC-1}), (DMR, \text{PROC-2}), (DMR, \text{NMR}), (FPROC, \text{PROC-FT}), (FPROC, \text{PROC-FT})\}$$

**IO binding** Binding the transformed application graph is not sufficient since the OWP processes must also be bound onto the architecture. As can be expected, IO
binding $\beta_{io}$ defines exactly one IO edge for each of the OWP processes such that $\beta_{io} \subseteq E_{IO}$:

$$\forall v \in V_{OWP} : |\{ i | (v, i) \in \beta_{io} \}| = 1 \quad (6.9)$$

The IO binding of the MJPEG application is, just as the computational binding, shown in Figure 6.10. There are two OWP processes (OWP0 and OWP1) that need to be mapped onto the architecture. Due to the feasibility rules (see Table 6.1), there is only one possible IO binding:

$$\beta_{io} = \{(OWP0, CAMERA), (OWP1, DISK)\}$$

**Message Dispatch** After determining the computational and the IO binding, the only remaining aspect is the message dispatch $\delta$ that deals with the communication between the processes. The message dispatch $\delta \subseteq E_D$ can be seen as the routing tables for the complete MPSoC. If $((q, r_s), r_d) \in \delta$, then the routing table at architectural element $r_s$ will contain the destination element $r_d$ for the request $q$. For the routing, there may be, depending on the communication architecture, different possibilities for the routing. Although this is also a potential design decision that can be explored, we have chosen to keep the routing fixed. Therefore, the architectures that we explore in our experiments only have one routing possibility between the different architectural components.

In order to verify if the dispatch is valid and complete, the set of messages in the mapped system needs to be determined. As discussed before, a message is the combination of a request (read or write) and the architectural resource the request is sent from. Therefore, the set $M_\delta \subseteq M$ contains the requests and the architecture resource of all the processes in the transformed application graph. The validity of the set of messages and the dispatch can be verified as follows:

$$\forall ((v_s, v_d), r) \in M_\delta : \exists f \in F \mid (\langle v_s, f \rangle \in \psi \land (f, r) \in \beta_x) \quad (6.10a)$$

$$\forall m \in M_\delta : |\{r \mid (m, r) \in \delta\}| \leq 1 \quad (6.10b)$$

The first requirement (Equation 6.10a) checks if the source node of the request is really mapped onto the resource that is specified in the message. Next, Equation 6.10b ensures that each message can have at most one destination.

These requirements, however, only verify that each message that is listed in the dispatch contains correct information. Another requirement is that it is complete. For each request $q \in Q$ that may be encountered during the lifetime of the embedded system, the dispatch should have the correct destination node for the communication:

$$\forall (v_s, v_d) \in Q : \begin{cases} 
\text{Internal}(v_s, v_d, f) & \exists f \in F : v_s, v_d \in G_f \\
\text{External}(v_s, v_d) & \notin f \in F : v_s, v_d \in G_f 
\end{cases} \quad (6.11)$$
There are two types of communication: internal (the process nodes are in the same subnetwork) and external communication (the other process is an OWP process or in a different subnetwork). Figure 6.11 shows the communication types. *Internal communication* is communication between two process nodes in the same fault-tolerant subnetwork (Figure 6.11 shows an example of such communication between processes A and B). Since the communication is within a single subnetwork, the data messages do not need to be verified. Similarly, it is known that internal communication is done within one and the same processor (as a complete replica of a subnetwork is mapped to a single processor). Therefore, the entry in the message dispatch $\delta$ is as follows (notice that the message is sent to the same resource $r$ from which it is initiated):

$$\text{Internal}(v_s, v_d, f) := \{ r | \left( \left( (v_s, v_d), r \right), r \right) \in \delta \land r \in R_p \} = n_{\text{proc}}(f) \quad (6.12)$$

In case the processes of the link are not in the same subnetwork, the communication is external. *External communication* passes the voter and consists of two steps:

$$\text{External}(v_s, v_d) := V(v_s, v_d) \land \begin{cases} \text{IO}(v_s, v_d) & v_d \in V_{\text{OWP}} \\ \text{Transfer}(v_s, v_d) & v_d \in V_N \end{cases} \quad (6.13)$$

First, *verified communication* ($V$) takes care of the communication from the replicated processes in subnetwork $G_f$ to the voter $r_d$. The voter $r_d$ (Equation 6.14a) will verify the contents of the messages of the different replicas (Equation 6.14b) of $v_s$:

$$V(v_s, v_d) := (v_s, f) \in \psi \land (f, r_d) \in \beta_X \land r_d \in R_R \land$$

$$\left| \{ r | \left( \left( (v_s, v_d), r \right), r \right) \in \delta \land r \in R_p \land r_d \in R_R \} \right| = n_{\text{proc}}(f) \quad (6.14a)$$

Equation 6.14b ensures that the number of replicas is exactly the same as the fault tolerance pattern defines (by means of $n_{\text{proc}}(f)$).

**Figure 6.11:** The different types of communication and the required buffers to enabling restarting.
Figure 6.12: The synthesized design for the mapped MJPEG example in Figure 6.10.

The second step of external communication depends on the type of the process node $v_d$. If the destination process is a normal process node ($v_d \in V_N$), there will be a transfer (T) of the message to the subnetwork of $v_d$. In case of an OWP process ($v_d \in V_{OWP}$), IO is performed:

$$\text{IO}(v_s, v_d) := \{ r | ((v_s, v_d, r), r_d) \in \delta \land r \in R_R \\
\land (v_d, r_d) \in \beta_{\text{io}} \land r_d \in R_I \} = 1$$

$$\text{Transfer}(v_s, v_d) := \{ r | ((v_s, v_d, r), r_d) \in \delta \land r, r_d \in R_R \\
\land (v_d, f_d) \in \psi \land (f_d, r_d) \in \beta_x \} = 1$$

This second step only needs to be done once, irrespective of the number of replicas. For a transfer, the destination resource must be the voter on which the subnetwork of $v_d$ is mapped. In case of IO, the destination resource must be the IO element on which process $v_d$ is mapped. If these requirements are all met, the internal and external communication is completely defined within message dispatch $\delta$.

An example of the dispatch for the MJPEG application is shown in Figure 6.12. This illustration shows the complete synthesized design of the mapped application from Figure 6.10 where the processes of subnetwork $G_{DMR}$ (DCT, DMUX and RGB2YUV) are replicated twice and mapped onto PROC-1 and PROC-2. Internal communication links (like DMUX, DCT) are duplicated for each replica and, therefore, are present twice in the dispatch: both for PROC-1 and for PROC-2. As the source and destination elements are the same, an internal buffer within the processor can handle the communication. For this communication link no other messages are required.

This is different for external communication. As shown in Equation 6.13, it consists of verified communication from the replicated process to the voter and an additional
communication step to get the message to the target process. An example of verified communication is the write request between the DCT process on processor PROC-1 and the voter on the NMR component. A verified communication of a read request, on the other hand, is connected to the splitter. An example would be the communication between the DMUX process on processor PROC-2 and the splitter on the NMR component. After the verification of the data, the splitter must forward the request to its destination. Depending on the type of destination node, different types of communication are involved: IO and Transfer.

A transfer is between two normal process nodes that are on different fault-tolerant subnetworks. In Figure 6.12, this is the case for the communication between processes DCT and Q. Upon the write request of DCT the verified data will be transferred from the voter component on the NMR element to the splitter component of the PROC-FT element. Before the DCT will read the data, it will send a read request. This read request is passed on to the splitter. If the data has already arrived earlier with a write request, the splitter will immediately provide the data. Otherwise, it will send the read request to the voter of the NMR and wait until the data is available.

Another possibility is an IO request that is addressed to an OWP process. As an OWP process is passive, its only activity is to wait on requests and to process them as soon as the data can be read or written. In Figure 6.12 the communication link (VLE, OWP1) involves IO. After the voter of the NMR component verifies the data, it will be sent to the DISK element that is now able to reliably store the sequence of frames encoded by the MJPEG encoder.

A fault-tolerant mapping is the complete combination of the patternization $\psi$, the computational binding $\beta_x$, the IO binding $\beta_{io}$ and the dispatch $\delta$. To be valid, the conditions in Equations 6.1, 6.2, 6.3, 6.5, 6.7, 6.8, 6.9, 6.10, and 6.11 must be met.

### 6.4 Simulation Model

A valid fault-tolerant mapping can be used to synthesize a fault-tolerant MPSoC design, as illustrated in Figure 6.12. In order to obtain the quality (like performance, energy, but also reliability) of such a design, SAFE simulates the multi-application workload in an unreliable environment. For this purpose, the SAFE simulation model extends the Sesame simulation framework in several ways. The first subsection discusses the fault injection into the architecture, followed by a subsection that describes the fault detection. After the detection of a fault, it must be handled in some fashion. Therefore, fault correction is introduced in the simulation model by means of restart with or without the usage of checkpoints. These aspects are elaborated on in the third subsection, after which the last subsection will discuss the obtained performance metrics.
6.4.1 Fault Injection

SAFE is able to model a fault-tolerant MPSoC design, but in order to test the quality of the design, it must be determined how faults will affect the execution of a multi-application workload on the MPSoC. Given the complexity of the simulation that is done within Sesame, an analytical analysis method would not be applicable. Sesame models the behavior of each individual component, including contention and other metrics that are hard to capture in an analytical method, and, therefore, the effect of the faults on the behavior of each individual component must also be taken into account. Examples are the potential communication bottleneck due to all the additional communication that is required, but also the effects of making checkpoints on a processor and the maximal amount of storage that such checkpoints require can be obtained when simulation is used.

We assume that only the processor elements are susceptible to faults. Other components like the communication network, reliability elements and the IO elements are assumed to be fault tolerant. The processors may have many components that can be affected by faults [5] like the register file and the logical units. Our architecture model, however, uses a high level of abstraction to describe the processor: it is known what function the processor is executing, when the function is running, how long it takes and when data is read or written. Therefore, we use the SoftWare Initiated Fault Injection (SWIFI) method [61]. Although both transient and permanent faults can be modeled this way, currently we only focus on transient faults.

For the occurrence of faults an exponential random distribution is used. As Figure 6.13 illustrates, the waiting time between the different faults is described using an exponential distribution. This resembles a Poisson process where the value \( \lambda^{-1} \) is equal to the Mean Time To Failure (MTTF). This distribution is very suitable for modeling the fault injection times [18], as the events in a Poisson process occur continuously and independently at a constant average rate. This is also the case for

![Figure 6.13: The cumulative distribution function of the exponential distribution that is used to model the waiting times between the faults.](image)
transient errors that are caused by a SUE: they are independent of earlier faults and they happen infrequently [29]. Some other classes of transient errors do show correlation between different errors. An example is the common mode failure [29]. These correlated faults are out of scope of this thesis and should be avoided using external techniques [47].

During the simulation, faults are injected one by one by iteratively picking a random number that is exponentially distributed. When a processor is hit by a transient fault, it does not automatically mean that all the applications are affected. The fault will only have consequences for the process that is running at the moment that the fault occurs. In case there is an active process, all future output of the process for the current frame will be considered erroneous. In reality, a fault may not affect the output, but in order to analyze this we require more details than can be provided by our high level model. Therefore, we take the most pessimistic assumption.

6.4.2 Fault Detection

Erroneous data will be propagated through the network, until it reaches a point where the fault-tolerant subnetwork verifies the data. For active redundancy, this is only the case when the data leaves the subnetwork. At this point in time, the majority voter will compare the results of the different replicas. For other techniques, like the fault-tolerant processor, the fault detection may almost be instantaneous.

Upon the detection of a fault, it depends on the fault tolerance pattern what is done to handle it. Generally, there are two possible actions that can be taken after the detection of a fault: fault masking and fault correction. Fault masking is applied when the correct data is known. When TMR is used, for example, a single fault among the three replicates can be masked as the correct value still has a majority. Without any time overhead, the fault can be masked by forwarding the correct data. As we currently only take transient faults into account, the corrupt replica is allowed to continue its execution. Since there is no explicit state between different frames in the KPN based applications (as discussed in Section 3.3), the next frame will not be affected by this transient fault anymore.

When the correct data is not known, the fault must be corrected. This is, for example, the case with DMR where a single fault can be detected, but the majority vote cannot identify which replica is incorrect. Depending on the available slack time, the current frame can be skipped or the fault is corrected. In the next section we will show how restarting is one of the potential ways of correcting a fault.

6.4.3 Fault Correction

In contrast to fault masking, fault correction will involve some resource overhead (like cycle time, energy or storage space). Therefore, it may seem to be the case that fault correction is not usable for embedded systems as the resources are quite
Figure 6.14: The timeline of a simple application to illustrate an implicit versus an explicit checkpoint.

stringent. However, with the increasing number of faults in time, the QoS of the embedded system may be severely affected. To be able to provide insight into the trade-off between QoS and resource overhead, SAFE provides the capability to model and simulate checkpoints and restarting the system from one of these checkpoints. Therefore, the fault correction techniques of SAFE can be extended upon the introduction of new fault tolerance patterns.

Checkpoint Budget

As illustrated in Figure 6.14, there are two types of checkpoints: implicit and explicit. Implicit checkpoints are located at the frame barriers. In the example of Figure 6.14 checkpoints $C_{A.3.0}$ and $C_{B.3.0}$ form the implicit checkpoint at frame 3. Notice that an implicit checkpoint is not taken at once. Process A reaches the barrier of frame 3 much earlier than process B. The complete implicit checkpoint is available once every process in the subnetwork has reached the specific frame barrier. A complete implicit checkpoint does not require storage (as there is no implicit state between frame barriers), but it allows us to perform message cleanup as will be discussed later on.

Similarly, the restart from an implicit checkpoint is trivial because no state needs to be restored for the processes. Still, restarting from an implicit checkpoint has some disadvantages. Not only is there a need to recalculate the complete frame, but it can also be the case that the application is not restarted at its full capacity. Take the simple application in Figure 6.14. On a restart from an implicit checkpoint at the end of frame 2, both processes start at the barrier of frame 2. In this case, process B needs to wait for output of process A before it can do any work.

To resolve this, explicit checkpoints can be taken during the lifetime of an ap-
Explicit checkpoints are initiated by the voter and store the state of all the processes in the active redundancy network and their internal communication channels at a specific point in time. In contrast to implicit checkpoints, the complete subnetwork is halted to capture the state of the current processes and the internal communication channels. To obtain a consistent explicit checkpoint, the voter will ensure that all the replicas stop at the same point in the application. One of the processes in the subnetwork will be responsible for collecting and sending the checkpoint. This results in a checkpoint for each replica, which will be compared by the voter. In Figure 6.14, $C_{A.4.1}$ and $C_{B.2.1}$ are illustrations of an explicit checkpoint.

Implicit and explicit checkpoints can also enhance each other. Take, for example, the checkpoints in Figure 6.14. If there is a restart just after the checkpoint of $C_{B.3.1}$, the restart can take place from a combination of the explicit checkpoint $C_{*,*1}$ and the implicit checkpoint $C_{*,3,*}$. This means that process A starts from $C_{A.4.1}$ and process B starts from $C_{B.3.1}$. Without the presence of explicit checkpoints, process A would have processed frame 3 again. Similarly, with only explicit checkpoints, process B would have been required to process frame 2 again.

For each fault-tolerant subnetwork, a checkpoint budget is defined. The checkpoint budget determines the checkpoint granularity by describing the number of explicit checkpoints per frame (possibly zero). The size, and thus the overhead, of the checkpoint is dependent on the application processes and the amount of data in the internal communication channels. This also means that the voting time of a checkpoint is variable. That is, the explicit checkpoints of the different replicas in a subnetwork must be verified against each other to ensure that on a restart the process state is valid. After verification by the voter, the explicit checkpoint is stored locally at the processor.

Explicit checkpoints are not only useful to minimize the amount of work that has to be redone, but they also allow to implement fault tolerance patterns like RFCS [58] or assertion-based techniques [28] where a fault is corrected by reprocessing the frame. This is, however, beyond the scope of the current paper.

**Restart Budget**

Using the checkpoints, a subnetwork is able to restart the frame upon the detection of a non-maskable fault, but this is not sufficient. As restarting requires overhead, it must be prevented that a subnetwork keeps restarting the same frame or that the restart of a frame leads to the deadline miss of the next frame. Therefore, a restart budget is specified that limits the number of times that a subnetwork can restart per completed frame. Similarly to the checkpoint budget, this is a nonnegative number that also can be zero to disable restarting. Additionally, a maximal processing time per frame can be defined after which the frame cannot be restarted anymore. In case the restart budget is depleted and a non-maskable fault is observed, the current frame is skipped.

In order to facilitate restarting, some data must be stored. These buffers are
shown in Figure 6.11. At first, the explicit checkpoints can optionally be stored locally at each of the processors that are used to run the replicas. In this case restarting becomes somewhat more efficient at the cost of the additional storage. In case no local copy of the checkpoint is available, the checkpoint will be stored and distributed by the voter. Additionally, a message buffer must be available at the splitter in order to ensure that the input data is still available when the subnetwork is restarted. Input data can be volatile due to two reasons: 1) input data is only temporarily available as input IO or 2) other subnetworks generate the input data. In the case of volatile data, the buffer prevents a cascading effect of other subnetworks that need to be restarted to provide the input data of the subnetwork. Both of the buffers can be protected from errors in several ways, but as SAFE is currently only targeted towards transient faults an ECC protection of the buffer should be sufficient.

The buffers that are modeled in SAFE have a limited capacity. Therefore, a policy is required to periodically clean up the unused messages. For this cleanup, two moments are chosen: during implicit checkpoints and during explicit checkpoints. After a successful explicit checkpoint, the complete contents of the message buffer can be discarded. As the explicit checkpoint contains the state of the process at that moment, it is guaranteed that the earlier messages are not required anymore. In case of an implicit checkpoint, the messages of the earlier frames can be discarded. In case the message buffer is out of capacity, all the new read request are blocked until the next explicit checkpoint (as discussed before, the explicit checkpoint will empty the message buffer and, therefore, will restore the full buffer capacity).

6.4.4 Performance Metrics

All the decisions with respect to the fault detection and handling will affect the quality of the system. Whenever an explicit checkpoint is taken the complete subnetwork needs to be halted. On top of that, local checkpoints need to be sent to the voter to be compared for correctness. A restart requires the restoring of the process states and the internal communication channels (only for an external checkpoint, not for an implicit checkpoint) and the splitter must resend all the cached incoming messages. All these aspects will affect the cycle time, the amount of communication and the required buffer space.

SAFE completely simulates the system including all the aspects at a high abstraction level such that an early fault-tolerant design space exploration can be performed. Metrics that can be obtained are, besides performance and power, frame rate and amount of skipped frames. It should be noted, however, that SAFE is not aimed to give any guarantees for real time behavior. SAFE will prune the design space such that only a limited number of designs need to be studied in detail during later design phases.
6.5 Experiments

The design space pruning is shown using a set of experiments that explore the fault-tolerant design space. For this purpose, three different applications have been selected: a MJPEG encoder, a Sobel edge detector and an MP3 decoder. All of these applications work on a stream of frames, where each frame is manipulated. The Sobel edge detector, for example, detects edges for each image in a video stream. Pedestrian detection is one of the techniques that utilize the detected edges in a video frame. Earlier in this chapter, we already saw an example of the MJPEG application. In the experiments, we use a slightly modified version that has more communication links to provide dynamic quality control of the image encoding.

Figure 6.15 shows the architecture on which the applications need to be mapped. It has four general-purpose processors that are connected to two communication buses. Both of these buses have a shared memory and a NMR component to support active redundancy. In this thesis, we have limited the fault tolerance patterns to the different flavors of DMR (two replicas) and TMR (three replicas). For both the DMR and TMR the checkpoint and restart budget can be varied from zero to six. If the restart budget is zero, the checkpoint budget is by definition also zero. As no restart is done, the use of a checkpoint is very limited. This results into 74 different fault tolerance patterns. There is one exception for the MP3 application, where the checkpoint budget is always zero. This is due to the fact that our model only models explicit checkpoints at communication events. As the MP3 application is modeled quite coarse grained, there are only two communication events per frame (read and write).

The fault tolerance patterns are meant to make the application fault tolerant. In our target architecture the four processors are the unreliable parts. Each of the processors has the same mean time to failure: $10^6$ FIT. Although one fault per 1000000 time units is relatively large, the use of such a high fault rate allows us
to magnify the differences between the different fault tolerant mappings that are explored. As a result, the trends become more clear\(^1\). All the other components, like the communication buses and the NMR components are already fault tolerant. Important to note is that for each experiment the same seed is used for the exponentially distributed soft errors that occur during the system simulation. As a result, the same sequence of transient faults is used to compare the fault-tolerant mappings. After a complete generation, the seed is changed and another sequence of transient faults is used to evaluate the fault-tolerant mappings.

The experiments will use the fault tolerance patterns to transform the three applications into a fault-tolerant application and to map them onto the unreliable architecture. In the first experiment, the patternization of the MJPEG application is fully analyzed by showing some of the optimal patternizations and to observe some general trends. The next three sections will use SAFE to do exploration of the fault-tolerant design space. Only a search of a small part of the design space will be done for each application in isolation, as in this chapter still an exhaustive search is performed (the next chapter will introduce an efficient technique to explore the complete design space).

### 6.5.1 MJPEG Patternization

When making a fault-tolerant mapping, three steps are performed: patternization, binding and dispatching. To study the process of patternization, we performed an exhaustive patternization for the MJPEG application and an MPSoC where only TMR patterns are used. There are multiple instances of the TMR in the set of fault tolerance patterns, involving different choices for the restart budget and the checkpoint budget. By limiting the set of available fault tolerance pattern to the

\(^1\)The fault rate that we use is currently only applicable to the extreme environments such as space applications, but with the decreasing technology scales the fault rate of the embedded systems at ground level are quickly increasing to these high fault rates [66]. For current systems, SAFE can perfectly be used after profiling the fault rate of the processors and to adapt the parameter in the architecture model.

**Figure 6.16:** Optimal subnetworks for a specific number of fault-tolerant subnetworks.
Figure 6.17: The effect of different numbers of subnetworks on the frame drop ratio of the MJPEG application.

TMR, the exhaustive search remains feasible.

Figure 6.16 shows the optimal patternization for a different number of fault-tolerant subnetworks. In this case, we have taken frame drop ratio as a primary objective and power consumption as a secondary objective. A first observation is that in this design space the patternization is incremental. By adding an additional subnetwork, one of the processes is moved into the new subnetwork. In the case of two subnetworks, the application is split into two equally sized subnetworks. Not only are these subnetworks equally sized, but also the number of external communication channels is kept minimal. Apart from the I/O communication channels (which are external by definition), only the channels (DCT, Q) and (DMUX, CONTROL) are external. Due to this minimum of external channels, the amount of majority voting (only done on external communication) is minimized.

When increasing the number of subnetworks to three, the DCT process is put into a separate subnetwork. The rationale is not the minimization of external communication, but the guarding of the compute intensive tasks. As the DCT is the most computationally expensive operation, it is beneficial to ensure that verified data is used in the computation. If it would have been unverified, it can be the case that unnecessary computation will be done. The same is true for the optimal patternization with four subnetworks. In this case, the quantization (Q) process is separated, being the second most compute intensive operation in MJPEG.

Having more fault-tolerant subnetworks may increase the quality of the application (with respect to frame drop ratio). However, it also increases overhead. This can be seen in Figure 6.17. Up to four subnetworks, the frame drop ratio reduces to 0 percent. With five or more subnetworks, the frame drop ratio quickly climbs up to
69 percent when each process is placed in a separate subnetwork.

In general, the patternization of the MJPEG application on the given architecture benefits from: 1) equally sized subnetworks, 2) a minimization of external communication channels and 3) the guarding of compute intensive tasks. These conclusions, however, only hold for this specific architecture. An architecture where, for example, the fault rate of the processors is much higher, may benefit from a larger fraction of external communication channels. Therefore, for such an architecture the fraction of external communication channels would be maximized instead of minimized in order to intercept faulty data as soon as possible.

### 6.5.2 Power versus Frame Drop Ratio

Figure 6.18 shows the frame drop ratio and power consumption for all the evaluated design instances for the MJPEG, Sobel and MP3 applications. To keep the experiment feasible, only a part of the design space is explored. The fault-tolerant mapping can freely segregate the application into different subnetworks, but only one and the same fault tolerance pattern can be used for all the subnetworks. Given this patternization, all possible bindings are explored. Therefore, the design instances in Figure 6.18 are colored based on the fault tolerance pattern that is selected. Two decisions are highlighted: 1) the usage of DMR or TMR and 2) restarting enabled or not.

An obvious conclusion is that TMR requires more power than DMR. Both DMR with and without restarting take less power than the design instances where TMR is chosen as the fault tolerance pattern of a design instance. The exception to this case can be seen in Figure 6.18c where a DMR based design instance uses both more power and has a higher frame drop ratio than the TMR based design instances. This is an example of a restart policy that is poorly suited to the specific application because often a frame is restarted just before the deadline of the frame. Although the restart policy may manage to correctly fix the frame, the frame will arrive too late and will be dropped. In this way a lot of power is invested to restart frames without any potential gain (the frame drop rate is above 80%).

Since power is not the only objective, it depends on the application if it makes sense to use TMR as a fault tolerance pattern or not. In case of the MJPEG application (Figure 6.18a) the design instances that make use of DMR dominate all the design instances that use the TMR pattern. None of the points in the Pareto front (the blue line) of the MJPEG application use the TMR pattern. For the other two applications Sobel (Figure 6.18b) and MP3 (Figure 6.18c) the Pareto front contains TMR based design instances that, at the cost of a higher power usage, obtain a lower frame drop ratio. It is up to the designer if the investment in power is worth the improved frame drop ratio.

Figure 6.19 summarizes the gain of the different fault tolerance patterns by showing the optimal frame drop ratio given a type of fault tolerance pattern. From the different types of fault tolerance patterns, plain DMR (without restart possibilities)
Figure 6.18: The evaluated design space of the three test applications.
has the highest frame drop ratio. This is to be expected, as plain DMR can only detect the faults and drop the frames when one of the replicas has suffered a fault (DMR is not able to identify which one is correct). As a result, it has the lowest power usage. If, for example, halfway a frame a fault is detected, the rest of the frame does not need to be processed. As the plain DMR only detects faults, it also shows the vulnerability of the application to faults in the architecture. The MJPEG application is the most vulnerable with 67% of the frames being dropped, whereas the MP3 application only has 9.9% of the frames that are affected by a fault. These frame drop ratios, however, give a worst-case scenario. Due to our pessimistic assumption that each fault leads to an unusable frame (which is not always the case for applications like MJPEG), some of the unaffected frames may not be dropped in the real execution of the fault-tolerant design. As SAFE provides early design space exploration of the fault-tolerant designs, the relative performance of the fault-tolerant designs is the most important. After the early design space exploration, the remaining fault-tolerant designs can be investigated in more detail by lower level analysis tools.

In case an additional replica is introduced, a single fault can be masked. The plain TMR technique already greatly reduces the frame drop ratio for all the tested applications. For both the MJPEG application and the Sobel application the frame drop ratio is reduced by approximately 36% to 31% and 16% respectively. MP3 was already less vulnerable to faults in the architecture and with plain TMR almost no frames are dropped (only 0.1% frame drop ratio remains). So, if no restart is taken into account, TMR definitely pays off.

For the MJPEG application it means that in at most 31% of the frames at least
two of the replicas are affected by a fault while processing the same data packet. To reduce the number of faults that cannot be masked one could adapt the MJPEG application to divide the frame into smaller subblocks. In this case, less time is spent for processing individual data packets. Consequently, this leads to a smaller probability on faults at two different replicas. Another technique is to use the restart possibility to retry the processing of the frame. For the MJPEG application this completely prevents the dropping of frames for both the DMR and the TMR technique. As the TMR requires more power, without restarting the TMR is worse than the DMR based design instances. Just as restarting allows us to reduce the frame drop ratio of the MJPEG application from 31% to 0%, the Sobel application also benefits from the restart mechanism. With DMR the frame drop ratio drops to 5% and with TMR only 1.4% of the frames are dropped. This is considerably lower than the 16% of frames that are dropped when no restarting is available. For the MP3 application, however, restarting is not necessarily beneficial. The overhead of the restarting leads to a frame drop ratio of 1.2% when restarting is used in combination with the DMR technique. This is slightly larger than the frame drop ratio of plain TMR that was able to reduce the frame drop ratio to almost zero. As TMR masks almost all the faults, adding the restarting possibility does not significantly affect the design instances. This is true for both frame drop ratio as well as energy (In Figure 6.18c the plain TMR design instances are hardly visible as they almost completely overlap with the design instances that use TMR with a restarting possibility).

We already mentioned that restarting requires overhead with respect to cycle time. This can clearly be seen in the case of the MJPEG application (Figure 6.18a). The design space of the evaluated MJPEG design instances shows two clusters for each of the fault tolerance types: on with a high and one with a low frame drop ratio. These clusters are not present in the design space of the Sobel and MP3 applications. Further analysis on the MJPEG application revealed that when the power consumption of the fault-tolerant design becomes below 5.8 the frame drop ratio quickly grows from 50% to values above 70%. In these cases, the fault handling capability of the MJPEG encoder is too low to process the frames correctly in the given timing requirements. As a result, the encoder cannot keep up pace with the incoming frames. Hence, many frames are skipped as the deadline of the frame is already passed before starting the processing of the frame. This higher number of unprocessed frames has a positive side effect: even less power is consumed.

To conclude, both the MJPEG and the Sobel applications can greatly benefit from the restart mechanism. Using the restart mechanism, the MJPEG application does not even need TMR to prevent the dropping of frames. The MP3 application is only negatively affected by the restart mechanism, as can be seen by the shape of the cluster of design instances that use DMR with the restart mechanism (Figure 6.18c). In this case, the power and cycle time that is spent to make the application more fault-tolerant only leads to a higher frame drop ratio. The way the fault tolerance overhead influences the frame drop ratio is discussed in the next experiment.
6.5.3 Breakdown of Frame Drop Ratio

Dropping a frame can have several causes, which is illustrated in Figure 6.20. First, there are corrupted frames. In these frames, a fault is detected (due to a transient error), but the restart budget was too small to correct these faults. Second, there are deadline misses. In these cases, the application is too late to retrieve or deliver a frame from/to the OWP process. For the experiment in Figure 6.20, we have taken all the design points of the previous experiments where the explicit checkpoint budget was zero. These design points are differentiated by restart budget and for each category the average frame drop ratio is given. Not only the total drop ratio is given, but also the fractions that are due to corrupt frames and deadline misses.

A larger restart budget can both improve the frame drop ratio (less corrupted frames) and degrade the frame drop ratio (more deadline misses). The more effort is put in fault correction, the lower the number of corrupted frames. However, the effort has a negative effect on the number of deadline misses. The optimal restart budget is thus application dependent. For all our applications, the gain in the reduction of corrupted frames is overshadowed by the increase of deadline misses at about one or two restarts per frame. However, the MP3 application is not influenced anymore once the restart budget is above three. In this case, the MP3 application is already able to circumvent corrupted frames and the additional restarts will not be used.

Clearly, the MJPEG application is more challenging with respect to fault-tolerant mapping than the Sobel and MP3 application. The high average number of deadline misses show that the requirements of this application are quite stringent. On average, the fault-tolerant mappings do not seem to have sufficient slack time for utilizing the complete restart budget to correct the faults within the corrupt frames.

6.5.4 Buffer Requirements

Earlier experiments clearly showed the trade-off between the advantages and disadvantages of the fault tolerance patterns on the frame drop ratio of an application. Potentially, fault tolerance patterns consume more power, but, depending on the application and architecture, this may improve the reliability of the design. One of the costs of fault tolerance patterns, however, has not yet been studied. Although restarting and the explicit checkpoints improve the frame drop ratio, it also involves an additional cost with respect to buffer requirements. To show these requirements, the best design instances for the MJPEG and Sobel applications have been selected for a checkpoint budget between zero and six. For all of these design instances restarting was enabled, as without restarting checkpoint buffers are not required. The outcome of this experiment is shown in Figure 6.21, with a horizontal axis that shows the checkpoint budget and a vertical axis showing the normalized buffer size (Figure 6.21a) or the frame drop ratio (Figure 6.21b). Normalization of the buffer sizes is done per application (i.e., the buffer sizes of the different applications are incomparable) such that the maximal buffer size is 1.0.
Figure 6.20: The frame drop ratio versus the size of the restart budget.
Figure 6.21 shows the relation between the explicit checkpoint budget size and the required buffer size. When explicit checkpointing is disabled, the system still takes implicit checkpoints. For these implicit checkpoints, the message cache (see Section 6.4.3) must contain all the incoming data messages since the last complete implicit checkpoint. As a result, the size of the message cache quickly grows. To reduce the size of the message cache, explicit checkpoints can be taken that will periodically flush the message cache. The more often an explicit checkpoint is made, the lower the maximal size of the message cache. Still, explicit checkpoints also need to be stored. The required size of the checkpoint buffer, however, is significantly smaller than the message cache and less influenced by the frequency of taking checkpoints (not shown in the graph). Combining these two trends, the required buffer size only reduces whenever the explicit checkpoint buffer size increases.

Although a larger explicit checkpoint budget may lead to a reduced buffer size, it will have a negative effect on other system objectives. Figure 6.21b shows how the explicit checkpoint budget affects the frame drop ratio. The MJPEG application can take up to six checkpoints per frame without any misses, but the Sobel application suffers from dropped frames. Initially, from an explicit checkpoint budget from zero to one the frame drop ratio is reduced since less work needs to be redone upon a restart. With two or more explicit checkpoints per frame, however, the overhead of taking the explicit checkpoints leads to a larger amount of dropped frames due to a higher amount of deadline misses.

### 6.6 Related Work

Due to the technology scaling, reliability becomes a major design objective [66, 46] for the embedded system world. A lot of research is dedicated to fault-tolerant embedded processors that, for example, increase reliability by replicating internal hardware [20, 27] or by making use of idle components [7, 49]. Introducing reliability, however, potentially has a major effect on other objectives. This effect must be quantized to allow the designers to reason about the amount of reliability they are willing to
put into the system and how this reliability will affect the different objectives of the systems. It is therefore important that reliability is explored as a separate design objective [24] and not hierarchically.

Therefore, DSE environments must be reliability aware. Most of the research in reliability aware DSE environments [3, 11, 28, 29, 32] is based on static task scheduling. Generally, these environments make use of task replication and re-execution to detect and handle faults for different fault scenarios. A fault scenario is a certain sequence of faults that occurs at specific processing elements. The approach in [32] uses a Tabu search to find a static fault-tolerant schedule that is able to make the application deadlines. It will automatically replicate and re-execute tasks to cope with the worst case fault scenario. Just as COFTA [11], this approach does not give reliability as an outcome of the DSE. The approaches of [28, 29] and [3] calculate the reliability of a static schedule of an application by determining the set of fault scenarios for which the schedule will meet the deadlines of the application (for [28, 29] this set of scenarios is encoded into a system fault tree and [3] calls this set of scenarios a working set). Next, the probability for each fault scenarios is combined to obtain the reliability of the static schedule. All of these approaches only consider sets of independent tasks, although COFTA [11] provides a trade-off between fault tolerance overhead and reliability by clustering tasks in different groups.

The method of Bolchini [6] provides an analysis framework that is capable of performing reliability analysis for an embedded system where dynamic scheduling is used. This analysis tool provides, given a fault scenario and an application schedule, a classification of how the faults affect the system. Examples of a classification scheme are no-effect, safe or dangerous.

One of the main novelties of SAFE is that it provides exploration of the fault tolerance patterns in a transparent fashion. Due to the separation of concerns, the fault tolerance patterns are completely separated from the application model and the architectural model such that the set of fault tolerance patterns can be extended. On top of that, SAFE is capable of fully simulating the complete embedded system with dynamic scheduling on a high abstraction level. As a result, the framework is capable of providing the reliability of the system next to the other potential objectives of the system. As a part of this system a unique fine-grained and parametrized checkpoint model is included. Most of the other fault-aware DSE frameworks only take implicit checkpoints (the strict separation between frames) into account [41]. This provides a unique view on all the effects of the provided fault tolerance patterns. Not only the way that the fault influences the system is classified, but also the effect of faults and fault tolerance patterns on all the objectives can be observed. This can be the buffer requirements, but also the number of dropped frames and the reason why the frames are dropped.

Additionally, SAFE operates on application networks instead of a set of independent tasks that is used by the aforementioned frameworks. In contrast to the clustering technique of COFTA, the clustering of processes that are part of a process
network takes the dependencies of tasks into account to leverage the overhead of the fault tolerance patterns. As a result, a real fault-tolerant DSE can be performed.

6.7 Conclusion

In this chapter SAFE was introduced that provides a framework to facilitate the fault-tolerant design of embedded systems. Fault-tolerant design takes into account the dynamism in the architecture with respect to transient faults that can occur. These faults lead to unreliability due to erroneous application outcomes. To make the output of the application more reliable, fault tolerance patterns are used. An example of a fault tolerance pattern is active redundancy where multiple replicas are used to run the program. By voting on the outcome, the detected faults can be masked or handled in other ways like skipping the frame or restarting it. Strictly, a fault tolerance pattern can be seen as a transformation on an application to introduce techniques to detect and handle faults. The policies (like the frequency of taking checkpoints) used during the fault detection and handling are also part of the fault tolerance pattern.

A fault-tolerant mapping automatically segregates the application into subnetworks, applies fault tolerance patterns on the subnetwork and binds the fault-tolerant application onto the architecture. By exploring the fault-tolerant mappings, the trade-offs of different fault-tolerant designs can be analyzed. Fault tolerance patterns may, for example, reduce the number of frames that need to be skipped or restarted due to faults, but they also introduce overhead. This overhead can be in buffer space for checkpoints, but also processor cycles that may lead to dropped frames due to deadline misses. Our experiments showed that the optimal fault-tolerant mapping is completely application dependent: for our Sobel and MJPEG application, for example, a lower frame drop ratio was achieved when the restarting mechanism was applied, whereas the frame drop ratio of the MP3 application was hardly affected. As a result, in case of the MP3 application the overhead in power consumption due to the restarting mechanism did not improve the fault-tolerant mapping.

This means that an automated way of exploring all fault-tolerant mappings is crucial to gain insight in the way fault tolerance patterns can improve applications that are mapped onto unreliable architectures. In the next chapter, the DSE framework will be presented that allows for performing a real automated exploration. Additionally, we take a closer look at the dynamism in the architecture and how this can play a role during the automated DSE.