Fault-tolerance at your Finger Tips with the TeamPlay Coordination Language

Wouter Loeve*
University of Amsterdam
Amsterdam, Netherlands
wouter@loeve.dev

Clemens Grelck
Friedrich Schiller University Jena
Jena, Germany
clemens.grelck@uni-jena.de

ABSTRACT

Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches. The functional coordination language TeamPlay follows the approach of exogenous coordination and organises an application as a streaming data-flow graph of independently operating, state-free components.

In this work we capitalise on this stringent application architecture for fault-tolerance against both permanent and transient hardware failure. We extend the TeamPlay language by a range of fault-tolerance features to be selected by the system integrator. We further propose a multi-core runtime system that is able to isolate hardware faults and manages to keep an application running flawlessly in the presence of hardware failure by adaptively morphing the application.

CCS CONCEPTS

• **Software and its engineering** → **Concurrent programming structures**; **Control structures**; **Functional languages**; **Concurrent programming languages**; **Data flow languages**; **Very high level languages**; • **Computer systems organization** → **Real-time languages**; **Real-time system specification**; **Embedded software**.

KEYWORDS

TeamPlay language, coordination, robustness, fault-tolerance, redundancy, functional programming, cyber-physical systems

ACM Reference Format:
Wouter Loeve and Clemens Grelck. 2023. Fault-tolerance at your Finger Tips with the TeamPlay Coordination Language. In *The 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), August 29–31, 2023, Braga, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3652561.3652571

1 INTRODUCTION

Cyber-physical systems (CPS) are computing systems that interact with the physical world via sensors and actuators [5]. Examples are manifold and range from self-driving cars and autonomous drones to automated manufacturing and building control.

Cyber-physical systems commonly have non-functional requirements on timing, energy consumption or fault-tolerance [28, 32] requirements that involve trade-offs: e.g., executing a task in less time usually incurs higher energy consumption. Adding fault-tolerance through some form of redundancy likewise increases energy consumption and response time, or requires additional hardware to meet deadlines. Safety-critical systems require extensive testing, validation and verification, which is hard to do if non-functional properties, like fault-tolerance, are intertwined with functional code [28, 32, 37].

We propose the coordination language TeamPlay [35] for a complete separation of concerns between functional code and the management of non-functional properties of cyber-physical systems running on heterogeneous parallel platforms. Functional code implements individual software components with defined behaviour, manageable size and determined non-functional properties, mainly worst-case and average execution time and energy consumption when executed across the range of compute facilities of a heterogeneous platform. The TeamPlay coordination code integrates the components into a cyber-physical system and manages the non-functional properties on global high level. Our design ensures that computational code within components can focus on functional correctness while letting the coordination code actively manage system-level integration and non-functional objectives.

In this paper we describe our current work capitalising on the two-layered software architecture enforced and implemented by TeamPlay for increasing the robustness of cyber-physical systems against permanent and transient hardware faults. Permanent failure of one core in a heterogeneous multi-core system does not necessarily incur total system failure provided the faulty core can be isolated and spared in the execution of the cyber-physical system through migration or generally re-allocation of software components to hardware units. Transient faults, also known as single event upsets, can occur any time and anywhere and may lead to erroneous computational results, non-termination (if a bit flip occurred in the control logic of the code) or premature termination by means of some form of exception.

Both permanent and transient hardware failure can lead to *crash faults*, i.e., faults that are directly observable in the software and that would lead to immediate system failure if not treated properly. In contrast, transient faults or single event upsets may likewise incur erroneous computational results that would normally remain unnoticed from a software point of view. We call these transient faults *data faults*. In the world of cyber-physical systems (and not only here) their consequences could be equally sever or even more dramatic than blunt but visible system failure.

*Wouter Loeve is now with the Royal Netherlands Aerospace Centre (NLR).

IFL 2023, August 29–31, 2023, Braga, Portugal

ACM ISBN 979-8-4007-1631-7/23/08
https://doi.org/10.1145/3652561.3652571

Such consequences can be avoided or at least their effect mitigated if we capitalise on the redundancy potential of (heterogeneous) parallel hardware. More precisely, we aim at exploiting some of the redundant hardware facilities for redundant computing including detection and mitigation of hardware faults. In principle, software fault-tolerance schemes, such as check-point restart, standby, N-modular redundancy or multi-version programming are well-known. However, their conventional application in a software system clutters the code, seriously reducing its maintainability. Furthermore, in cyber-physical systems the impact of increased fault-tolerance on non-functional objectives such as time and energy is at best unclear.

Here, we capitalise on the TeamPlay coordination model for the design of cyber-physical systems. We extend the TeamPlay language by facilities to augment coordination code with fault-tolerance directives on a per-component granularity. This provides the system integrator with *fault-tolerance at the finger tips*. At the same time, our design permits schedulers and schedulability analyzers to take redundant execution of components into account. Last but not least, we present a fault-tolerant runtime environment that executes TeamPlay code on heterogeneous parallel architectures implementing the various fault-tolerance options provided. We make the following contributions:

(1) facilitate the management of fault-tolerance strategies in a coordination context;
(2) devise an adaptive, fault-tolerant runtime environment for TeamPlay

all while maintaining the strict separation of concerns between functional code on one hand side and the management of non-functional properties on the other hand side.

The remainder of the paper is organised as follows. In Section 2 we revisit the exogenous coordination model underlying the TeamPlay language and introduce the language itself to the degree necessary. In Section 3 we introduce our fault-tolerance extensions while we explain our adaptive, fault-tolerant runtime environment in Section 4. We sketch out related work in Section 5 before we draw conclusions and illustrate directions of future work in Section 6.

2 COORDINATION MODEL AND LANGUAGE

The term *coordination* goes back to the seminal work of Gelernter and Carriero [15] and their coordination language Linda. Coordination languages can be classified as either *endogenous* or *exogenous* [3]. Endogenous approaches provide coordination primitives within application code; the original work on Linda falls into this category. We pursue an exogenous approach that fully separates the concerns of coordination programming and application programming.

2.1 Components

Our exogenous approach fosters the separation of concerns between intrinsic component behaviour and extrinsic component interaction. The notion of a component is the bridging point between low-level functionality implementation and high-level application design. We illustrate our component model in Figure 1. Following the keyword `component` we have a unique component name that serves the dual purpose of identifying a certain application functionality and of locating the corresponding implementation in the object code.

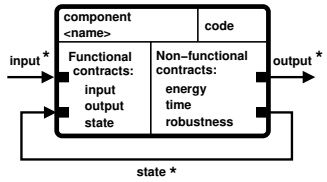

Figure 1: Illustration of component model

A component interacts with the outside world via component-specific numbers of typed and named input ports and output ports. As the Kleene star in Figure 1 suggests, a component may have zero input ports or zero output ports. A component without input ports is called a *source component*; a component without output ports is called a *sink component*. Source and sink components form the quintessential interfaces between the physical world and the cyber-world characteristic for cyber-physical systems. They represent sensors and actuators in the broadest sense.

Technically, a component implementation is a function adhering to the C calling and linking conventions whose name and signature can be derived from the component specification in a defined way. This function may call other functions using the regular C calling convention. However, the execution of the function, including execution of all subsidiary functions, must not interfere with the execution environment. Exceptions are source and sink components that are supposed to control sensors and actuators.

2.2 Stateful components

Our components are conceptually stateless, and, hence, our approach is functional. However, some sort of state is very common in cyber-physical systems. We model such a state in a functionally transparent way, as illustrated in Figure 1. We employ so-called *state ports* that are short-circuited from outport to inport.

Our approach to state is in an interesting way not dissimilar from main-stream purely functional languages, such as Haskell or Clean. They are by no means free of state either, for the simple reason that many real-world problems and phenomena are stateful. However, purely functional languages apply suitable techniques to make any state fully explicit, be it monads in Haskell [30] or uniqueness types in Clean [1]. Making the state explicit is key to properly dealing with state and state changes in a declarative way. In contrast, the quintessential problem of impure functional and even more so imperative languages is that the state is potentially scattered all over the place. And even where this is not the case in practice, proving this property is hardly possible.

2.3 Non-functional properties

One of the goals in the design of the TeamPlay coordination language is the active management of non-functional properties, namely energy and time. Hence, any component comes with non-functional contracts in addition to functional contracts. Both energy and time can only be considered in relation to some concrete execution platform, individual core types for heterogeneous architectures and dynamic voltage and frequency (DVFS) settings where applicable. Any mentioning of energy or time in the coordination source code would inherently make the code hardware-specific, which is

not what we want. In contrast, we employ a *non-functional properties file (NFP)* that functions as a data base storing per component time and energy consumption values for the variety of hardware units of interest including their DVFS settings. Concrete values are derived by static analysis, dynamic profiling or simply asserted by the user.

2.4 Component interplay

Components are connected to each other via channels, as illustrated in Figure 2 for an imaginary subsystem of a car, where two sensors feed messages to a decision controller, which synchronises the messages pair-wise and sends commands to two subsequent actuators. Channels serve both data exchange as well as dependency modelling. We distinguish between *point-to-point channels* connecting one outport to one inport of some subsequent component and *broadcast channels* connecting one outport to multiple inports simultaneously. Source components start computing at statically defined time slots. All other components are activated by the presence of input data on all inports.

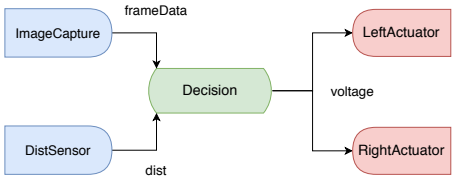

Figure 2: Example for TeamPlay component coordination

2.5 The TeamPlay Language

Figure 3 shows the TeamPlay coordination code that implements the example of Figure 2. A TeamPlay application starts with the keyword `app` followed by an identifier that serves as the application's name. Enclosed within curly brackets we can identify the two major code regions of any TeamPLay coordination program: components and channels.

Coming back to Figure 3, we can easily identify the definitions of the five components of our example from Figure 2. They are enclosed in curly brackets following the key word `components`. A component definition starts with a name followed by a pair of curly brackets enclosing further information about the component. As components communicate with other components via (FIFO) channels, the corresponding ports are the most vital functional properties (or contracts) of components. Following the key words `inports`, `outports` or `state` (The latter is not shown in the running example, but the syntax is identical.) we have a list of pairs of port type and port name. Again, we adopt a syntax inspired by that of C struct definitions. For example, the `Decision` component has two input ports, i.e. port `dist` of type `num` and port `frameData` of type `frame`, and one output port `voltage` of type `num`. Port types must refer to types previously defined in the `datatypes` section of the coordination code. Port names are freshly introduced identifiers. The number of ports a component may have is fixed but unbounded.

```
app car {
  components {
    DistSensor {
      outports {num dist}
    }
    ImageCapture {
      outports {frame frameData}
    }
    Decision {
      inports { num dist; frame frameData}
      outports { num voltage}
    }
    LeftActuator {
      inports {num voltage}
    }
    RightActuator {
      inports {num voltage}
  } }
  channels {
    DistSensor.dist        -> Decision.dist;
    ImageCapture.frameData -> Decision.frameData;
    Decision.voltage       -> LeftActuator.voltage
                            & RightActuator.voltage;
} }
```

Figure 3: TeamPlay code for example of Figure 2

Optionally and not shown in the example, ports can specify a multiplicity other than the default of one token, for instance as in `inports {frame in[3]}`. The multiplicity of an inport indicates the number of tokens required for firing while the multiplicity of an outport indicates the number of tokens produced on this port in a single round of firing.

Components are connected with each other via *channels*. In Figure 3 we can identify two kinds of channels: regular channels connect a single outport to a single inport while broadcast channels using an ampersand send a single data item from one outport to multiple inports. The TeamPlay compiler applies static analyses to guarantee type correctness, absence of cycles and that any port is connected to at most one channel. If an outport is not connected to any subsequent component, data items are systematically discarded. Input ports must always be connected to exactly one channel. Ports are identified by a component name and a port name separated by a dot, but the port name may be omitted if a component only has one outport or one inport.

2.6 Multi-version components

As illustrated in Figure 4, a component may have multiple versions, each with its own non-functional contracts, but otherwise identical functional behaviour, similar to [38].

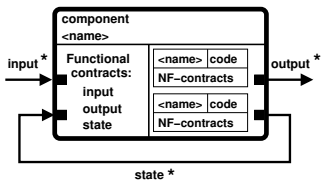

Figure 4: Multi-version component with individual non-functional contracts

With our focus on non-functional properties, it becomes particularly interesting to have multiple versions of a component that

expose identical functional behaviour, but that implement different trade-offs of the non-functional properties of interest.

```
Decision {
  inports { num dist; frame frameData}
  outports { num voltage}
  version Simple {...}
  version Optimal {...}
}
```

Figure 5: Example of a multi-version component: we refine the Decision component from Figure 3 with two versions named Simple and Optimal, which are likely to show different dynamic behaviour with respect to time and energy.

Figure 5 demonstrates how this can be accomplished in TeamPlay. The new definition of the `Decision` component features two different versions with different quality/complexity and, hence, different time/energy behaviour. Different versions of one component all share the same port specifications and must behave identically from a functional perspective.

3 EXTENSIONS FOR FAULT-TOLERANCE

In this work we extend the TeamPlay language by a new category of non-functional contracts: robustness or fault-tolerance methods, as already included in Fig. 1. We opt for a user-directed approach where the user can add fault-tolerance methods from a range of predefined options and their parameters to individual components or their versions. This design acknowledges the fact that different components are of different criticality for the well-being of the system as a whole, and only the system integrator can make educated choices where to add which method. With this approach tolerance against permanent and transient hardware faults can be achieved with a minimum of additional coding and completely transparent for the functional component implementations. The TeamPlay language also provides syntactic means to annotate fault-tolerance properties to subsets of components or to all components in a uniform way [23]. In the following, however, we omit these language features and rather focus on the various supported fault-tolerance methods and their parameters.

3.1 Checkpoint/restart

Checkpoint/restart lets the system return to a stable (backup) state when a (permanent or transient)crash fault has occurred [39, 41]. Generally, the downside of checkpoint/restart methods is the difficulty to assess the concrete state of a failing software unit that needs to be saved. Thus, in the worst case the entire process image needs to be saved at each checkpoint, which would be very expensive, both in storage space and execution time.

Here, the architecture of our coordination-based approach pays off. It creates a middleware layer where our system software can precisely keep copies of the arguments of an individual component invocation before giving control to the third-party component implementation. The stateless nature of TeamPlay components ensures that no other data affects the computation. Note here that this property remains valid even if state ports are used as described in Section 2.2. Backup copies of argument values only need to be stored while the component is computing and can be discarded automatically as soon as the component has emitted data on its outports.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  checkpoint {}
}
```

Figure 6: TeamPlay specification of checkpoint/restart

Figure 6 shows how checkpoint/restart can be specified in TeamPlay based on the example of the `Decision` component from Figure 3. Currently, checkpoint/restart has no options, hence the empty pair or curly brackets. The advantage of checkpoint/restart is the absence of any coordination between hardware components or potential replicas. We merely need to keep the arguments, which we must do anyhow as component implementations are generally not permitted to destructively change argument data. The disadvantage lies in time-constrained scenarios where a deadline must be met. Since the restart only occurs after a problem has been detected, the runtime can double in the worst case. Of course, there is no guarantee that the second attempt to run a component (on a different hardware unit) is more successful than the first one, but we do not consider this in time-constrained scenarios.

3.2 Primary/backup

With the primary/backup method, a standby (backup) component can take over the role of the primary component in case of failure, as illustrated in Figure 7. Both the primary and the standby component are scheduled to run, usually on different hardware units. Should the primary component terminate without issues, its produced results are used for further computing. Otherwise, the standby component takes over, and its results are routed to subsequent components in the TeamPlay application. Robustness can further be improved by multiple standby components. Like with checkpoint/restart we address permanent and transient crash faults with the primary/backup method.

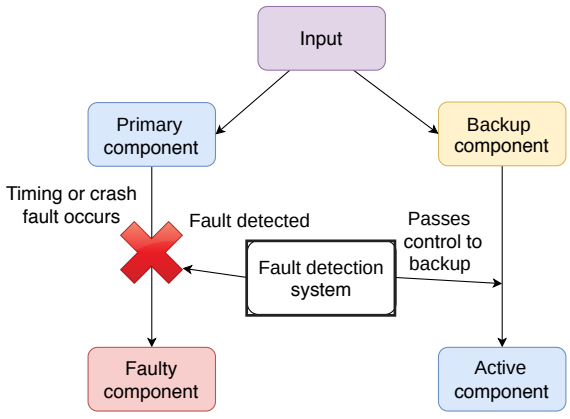

Figure 7: Illustration of recovery of a failing primary component using the primary/backup method.

The advantage of the primary/backup technique compared to checkpoint/restart is that we do not wait for a fault to occurr and be detected to replicate the computation. This saves time at the expense of additional computing resource utilization and energy consumption. Figure 8 shows how primary/backup can be specified in TeamPlay. The `replicas` option denotes the number of replicas to run for this component. The default of two replicas means one primary and one backup component, but we could employ multiple backup components as well.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  standby {replicas 2}
}
```

Figure 8: TeamPlay specification of primary/backup with default settings

In TeamPlay, component execution is discrete, i.e., component execution produces output tokens exactly once. This means that it is not necessary to synchronise primary and standby components. We can simply provide the input tokens to both and take the first component that delivers output as the primary component. If it fails, one of the standby components will deliver output instead.

3.3 N-Modular redundancy

In this classic fault-tolerance method *n* identical replicas of a component are run with identical input [24, 27, 41], just as in the case of primary/backup. However, this time these *n* replicas are complemented by a voting layer that allows us to detect transient data faults, a capability that goes beyond checkpoint/restart and primary/backup.

With two replicas (*double modular redundancy*) we can detect transient data faults, and with three (*triple modular redundancy*) or more replicas we can even correct such faults. In case one or another replica does not produce any output (so-called *crash faults*), double modular redundancy behaves similar to primary/backup in that the remaining result is taken for further computing. In the general case of N-modular redundancy, voting is restricted to the results effectively produced. Hence, N-modular redundancy addresses all fault classes introduced: permanent and transient faults just as crash and data faults. In this sense it can be considered a true extension of primary/backup.

Figure 9 illustrates N-modular redundancy. In the simpler case each replica sends its result to a single voting component that does the majority voting as described above and sends the result deemed correct, or perhaps error tokens, to subsequent components in the component graph. However, voting components may fail as well. Hence, we support multiple replicas of voting components as well, thus apply the idea of N-modular redundancy to the voters [4]. As can be seen in Figure 9, TeamPlay supports a whole layer of identical voting components, where the number of voting component replicas is independent of the chosen number of original component replicas. The default number of voting component replicas is one. If more than a single voter exists, all voting components send their results to a super voter that makes the final decision and forwards the results to to subsequent components in the component graph.

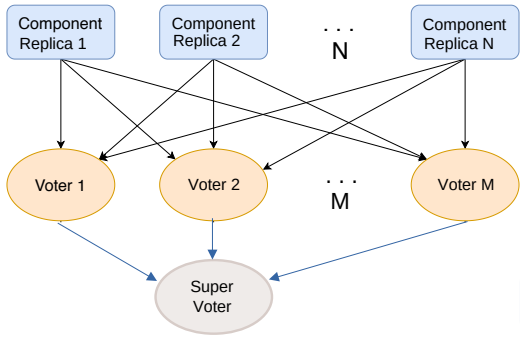

Figure 9: Illustration of N-modular redundancy

Now, one question remains: what if the super voter fails? This question illustrates that with any fault-tolerance method whatsoever we can only reduce the likelihood of system failure, not rule out system failure entirely. Hence, we stop at this point. Furthermore, we generally assume that likelihood of transient faults is linear in the execution time of a component. Here, we further assume that voting is considerably less time-consuming than actual computing, and, hence, it is less likely that a voter is subject to a transient fault than a compute component. Super voters, again, could be implemented more efficiently than general voters, and as such would be even less prone to transient faults.

Figure 10 shows how N-modular redundancy can be specified in TeamPlay; we support the following options:

- `replicas` (line 5), integer signifying the number of replicas. Default is 3, i.e. triple modular redundancy.
- `votingReplicas` (line 6), integer signifying whether and how much the voting processes need to be replicated.
- `waitingTime` (line 7), how long processes should wait before initiating the voting process. Given as a percentage of the average execution time of the finished components, the percentage can be higher than `100%`.
- `waitingStart` (line 8), defines the starting point of waiting. When `waitingStart` is `majority`, processes start waiting based on the execution time when a majority of processes are done. In the case of `single`, the waiting will start when a single process is ready.
- `waitingJoin` (line 9), flag defining whether processes that are finished later should be added in the waitingTime calculation, applies to both a `waitingStart` value of `majority` and `single`.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  nModular {
    replicas 3
    votingReplicas 2
    waitingTime 30
    waitingStart majority
    waitingJoin true
} }
```

Figure 10: TeamPlay specification of N-modular redundancy with default settings

N-modular redundancy has in common with primary/backup that it is time-predictable and, hence, better suited for real-time systems than checkpoint/restart [31].

3.4 Multi-version programming

In multi-version programming multiple functional equivalent implementations of the same component are created [29]. Our coordination runtime environment can run them in the same way as with N-modular redundancy. Multi-version programming extends the capabilities of N-modular redundancy in detecting and mitigating hardware faults to software or implementation faults. The disadvantage of multi-version programming is that in addition to high runtime resource overhead it likewise incurs high development cost for multiple functionally equivalent implementations of software components.

As explained in Section 2, the TeamPlay language already supports the concept of multi-version components. What has initially been intended to exploit different energy/time trade-offs can now be reused for fault-tolerance in a rather straightforward way.

```
Decision {
  inports { num dist; frame frameData}
  outports { num voltage}
  version SimpleDecision {...}
  version OptimalDecision {...}
  nVersion {
    versions [ (SimpleDecision, 2),
               (OptimalDecison, 1) ]
} }
```

Figure 11: TeamPlay specification of multi-version fault-tolerance

Figure 11 illustrates the specification of N-version programming in TeamPlay. Here, the `versions` parameter defines which of the existing versions should be used and how many replicas of each of these versions should be run. We reuse our multi-version example from Figure 4 here to illustrate the additional syntax. To properly reuse multi-version programming for fault-tolerance, however, all versions must be guaranteed to yield the same result for the same input data. This would presumably not be the case in the example of Figure 11 with a simple and an optimal decision procedure.

4 FAULT-TOLERANT RUNTIME SYSTEM

In order to support the various TeamPlay language extensions for fault-tolerance we propose a corresponding fault-tolerant runtime environment that dynamically re-configures running applications upon detection of hardware faults. This runtime environment addresses both permanent and transient as well as crash and data faults. The runtime environment comes with a fault injection facility for demonstration purposes.

4.1 Base runtime system

Cyber-physical systems run on a huge diversity of computing platforms ranging from simple signal processors to system resembling fully-fledged servers in complexity as well as compute capabilities. With our work we target (at least) high-performance embedded systems that feature a number of different types of cores and multiple to many cores of each type. We further assume a minimum operating system layer that supports, for instance, multithreading with basic synchronization facilities. Examples of such devices would be the Jetson or Odroid families of embedded architectures, just as the whole range of modern mobile phones.

When programming such systems we usually do not consider the continuously looming possibility of both permanent or transient hardware faults. The former would usually incur immediate whole system failure, even if only a single core is faulty. The latter would usually remain unnoticed, unless it affects the control logic of the software with the effect of non-termination. Now, with our proposed extensions to the TeamPlay coordination language we explicitly target these cases. To this effect, we make two further assumptions. Firstly, the hardware and (potentially shallow) operating system layer tolerate the permanent fault of individual cores while the system as a whole remains operational. Secondly, we assume one hardware unit that is specifically hardened against both security attacks as well as hardware failure and, thus, can be assumed not to fail, whereas all other units may indeed fail sooner or later in one or another way.

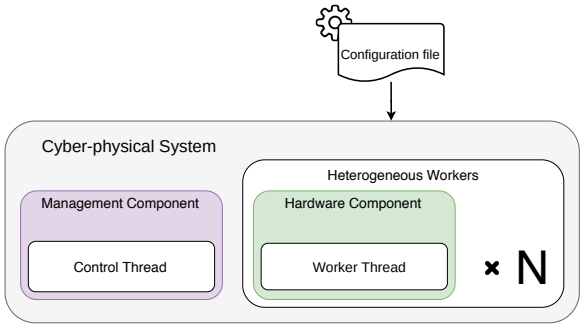

Figure 12: Base architecture of the proposed fault-tolerant runtime system with one control thread for management and N worker threads for computation

We control spatial and temporal mapping of TeamPlay components to hardware execution units (i.e. cores) by creating one thread per core and by pinning each thread to one specific core. As illustrated in Figure 12, we have two types of threads: a *control thread* and multiple *worker threads*. The control thread runs on the aforementioned specifically protected hardware unit, and its responsibility is to launch component-level computations on suitable available threads (and thus cores) while obeying the various fault-tolerance schemes described throughout the previous section.

4.2 Control vs worker thread(s)

We illustrate the interaction between the control thread and the worker threads in Figure 13. The light-green background on the right hand side marks the realm of the control thread while the light-blue background on the left hand side indicates actions taking place in the worker threads. For reasons of readability we only show one worker thread in Figure 13. Data structures on the boundary between control thread and worker threads are the means of communication between them. Looking here first and from top

to bottom, we can identify the *thread task queue* of each worker thread where the control thread pushes jobs. In the middle we see a representation of the coordination task graph. And towards the bottom we see the finished list which is used to communicate that a thread has completed a job and tokens are added to its output buffers. The tables next to task queue and finished list show the first four iterations on the task graph in the centre of Figure 13.

First, we explain how the interaction works in general terms, then we go over the concrete example shown in the two tables. When the control thread launches (top right node), it goes through the source nodes of the task graph and adds them to the task queues of the assigned threads. Each of these threads has a (counting) semaphore which corresponds to the number of items in their queue. When the semaphore reaches zero, the threads wait until new items appear in the task queue.

When a thread is alerted that new items appeared in the task queue (top data structure), it pops a component from the task queue (task queues are FIFO) to execute. After execution, it stores the output data in the task graph data structure and appends the id of the executed component into the finished list. The control thread is alerted that components are finished, so it can check whether new items can be added to the task queues. The computing thread will then wait for the task queue semaphore. If the semaphore's value is greater than zero, it can continue popping another item from the task queue to resume computing.

After items are added to the task queues of the threads and the threads are alerted, the control thread will wait until items appear in the finished list. This is indicated in Figure 13 by the bottom data structure with the dotted line facing right. This mechanism is implemented with a condition variable as one cannot reset a counting semaphore when the finished list is emptied. When items appear in the finished list, we need to check which components can fire again. First, we need to check whether the predecessors of the finished component can fire since, by firing, it can have opened a spot in the (bounded) FIFO buffers of the predecessors. Then, we check whether the successors of this component can fire since it has produced a token on its outports which may trigger the firing rule of the successor(s). Finally, we check whether the component itself can fire again. This way of checking ensures we only have to traverse the parts of the graph that have been changed. The components that can fire are added to the task queues belonging to the threads, and the threads are triggered to continue computing. The components that are ready are added to the task queue, which marks the completion of a cycle.

Now, we explain the concrete example characterised by the various tables in Figure 13. First, both threads will launch. The worker thread sees that there are no items in the task queue (i.e. the semaphore is zero), so it will wait. In the first cycle, the control thread adds the `Source` component to the task queue. As the source component does not have any dependencies, it can fire as long as the buffers can hold the data and it is not already present in any task queue. The task is put in the task queue associated with the computing thread to which `Source` is assigned. The control thread will increment the semaphore. This leads to the awakening of the worker thread, which will pop `Source` from the queue. The worker thread then executes the code associated with `Source`. After computation, the output token of `Source` is added to the buffer on the edge leading to the subsequent component `A`. The worker thread puts the id of the `Source` component into the finished list and sends a signal to the condition variable on which the control thread is waiting. The worker thread loops back to the first item (after initialisation) and waits until the control thread has added new items to the task queue owned by the worker thread.

When the control thread receives the signal for the condition variable, it will loop trough the finished list and check the task graph for components that are ready. This is done by looping trough the predecessors, successors and the component itself, to see if they can fire. Component `Source` has no predecessors, but it does have one successor, namely `A`, which can fire since `Source` just fired. Likewise, `Source` can fire again. The components which can fire again are put into the task queue. Next, component `A` is fired and the result is again stored in the buffer after the fired component, this time leading to `Sink`. Then the control thread is again alerted that the worker thread has finished a computation. The control thread notices that `Source` can be fired since it has no dependencies, but it is already in a task queue, so it cannot be added again. Following the execution of `A`, `Sink` can be fired, but `A` has insufficient tokens from `Source` to fire again. Now, `Source` is taken from the task queue and executed, as it was added the previous cycle. The component checking process of this cycle is identical to the first cycle, as `Source` and `A` are added again. Then, for the last round of this example, `Sink` is popped from the task queue and executed. In the control thread, `A` cannot be added to the task queue again since it was already added when `Source` finished. `Sink` cannot fire again since the buffer on the edge coming from `A` does not have sufficient tokens.

4.3 Crash fault detection

In order to implement the fault-tolerance mechanisms added to the TeamPlay coordination language, we make several additions to our base runtime system. We start out by describing how we detect crash/permanent faults required by both checkpoint/restart and primary/backup. When a system suffers from a crash failure, we cannot assume that restarting the hardware component will solve the problem. As depending on the hardware configuration, data can be stored in non-volatile storage, which is lost after the restart. To prevent deviation from correct service caused by lost data, we introduce error tokens. For the reconfiguration process, mechanisms are required which reassign tasks to other threads with a compatible architecture. Furthermore, we introduce a mechanism to deal with heterogeneous architectures and spatial assignment of components to threads.

There are various methods for detecting node failure [21, 41], the perhaps most prominent one being *heartbeat*. This method actively pings nodes at given time intervals to check if they are still alive. Knowing the worst-case execution time of a component, we could alternatively wait for the expiration of a component's time slot, but given that we also must meet a deadline, we prefer to use a heartbeat mechanism to detect faults early and, hence, to be able to commence mitigation methods as quickly as possible.

Implementing heartbeat is not straightforward since using signals and signal handlers on individual threads does not guarantee which thread effectively handles a signal. Furthermore, we aim to treat the components as black boxes, so we cannot intrude into the

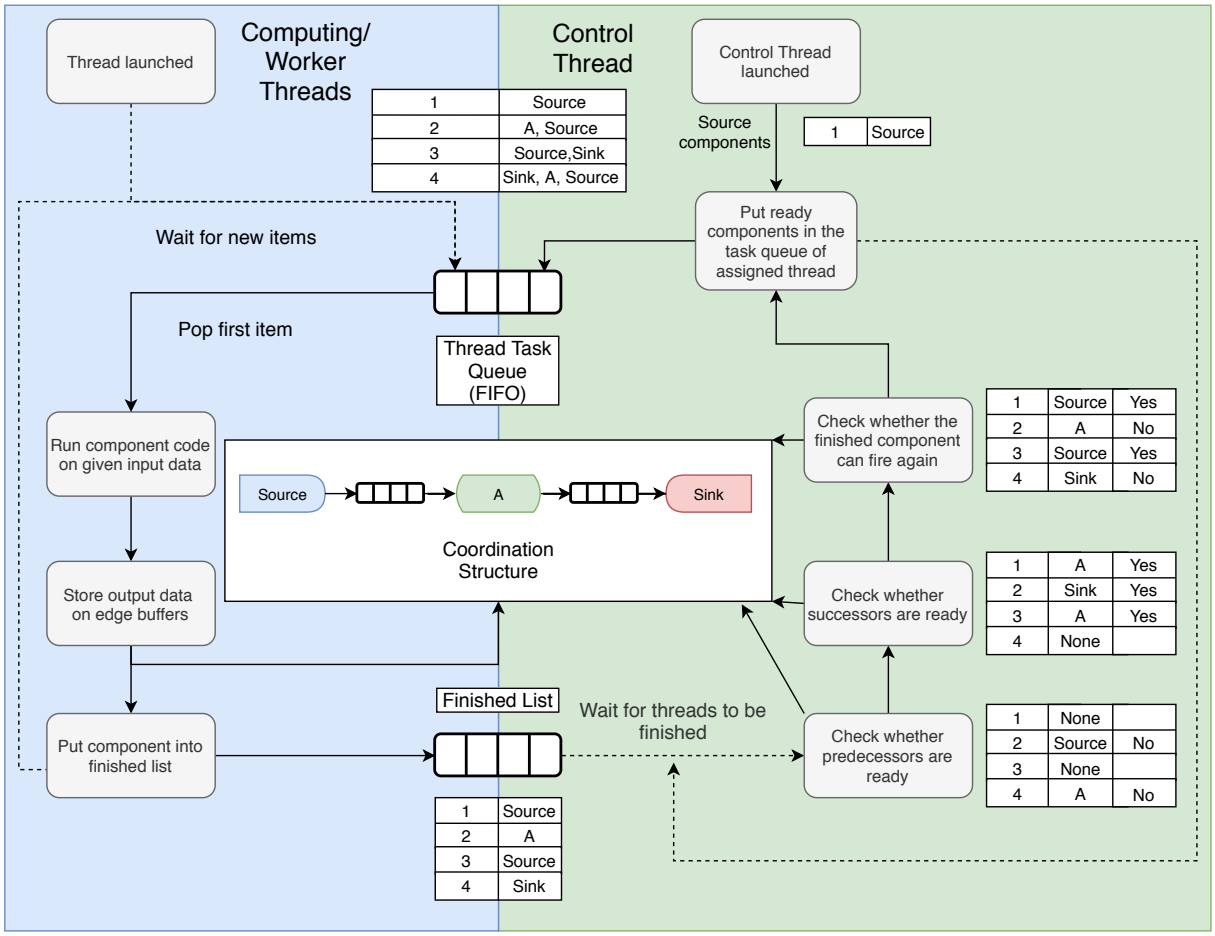

Figure 13: Schematic overview of the interaction between the control thread (right) and the worker threads (left). Dotted lines indicate waiting via thread communication, e.g., condition variable or semaphore. The double-column tables show the components in the list per iteration. The triple-column tables show the components which are checked for firing. The third column shows whether or not they are added to the task queue.

user code to send the signal periodically from there. To solve this problem, we add a separate *heartbeat worker thread* to each worker thread. The heartbeat worker thread and the corresponding worker thread are pinned to the very same hardware unit, hence they are guaranteed to both fail if that hardware unit fails. Additionally, we introduce a *heartbeat control thread* running on the control unit, which periodically checks whether the heartbeat threads are still alive. The heartbeat control thread periodically loops trough the heartbeat worker threads and increments a worker-specific counter in shared memory. The heartbeat worker thread periodically resets the counter incremented by the heartbeat control thread. The heartbeat control thread in turn checks whether the counter is higher than a specified threshold. If so the corresponding hardware unit is deemed faulty. Our final runtime system architecture is illustrated in Figure 14.

To avoid false positives, i.e. threads that are detected as having crashed but effectively are still alive, we let the user set thread sleep times as well as the threshold used by the heartbeat control

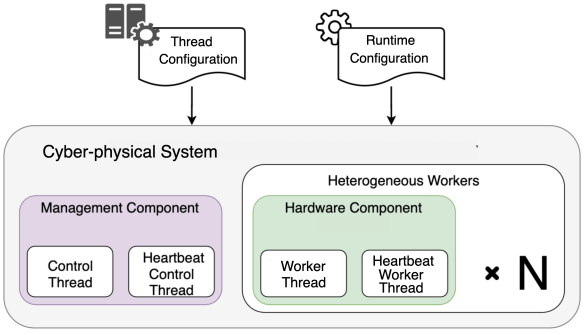

Figure 14: Fault-tolerant runtime system architecture extended with heartbeat threads

thread. Choosing the heartbeat worker thread sleep time too close to the heartbeat control thread sleep time increases false positives, but decreases the error detection time. Lowering the threshold

has a similar impact: the lower the threshold, the faster errors are detected at the price of increasing the chance of false positives.

In order to further decrease the amount of false positives, we change the scheduling type of both types of heartbeat threads to use real-time scheduling policies supported by pthreads. The function `pthread_setschedparam` supports two scheduling policies: FIFO and round robin. FIFO scheduling (`SCHED_FIFO`) runs a thread to completion in FIFO order. Round-robin scheduling (`SCHED_RR`) aims to give each thread equal execution time, but involves a larger number of context switches. FIFO scheduling does not work in our system because the heartbeat threads are continuous tasks. Thus, we enable round-robin scheduling on both types of heartbeat threads. Additionally, we enable the user to set the priority of the heartbeat control thread and the heartbeat worker threads through configuration parameters. We elaborate on the configuration parameters of our runtime system in general in Section 4.10.

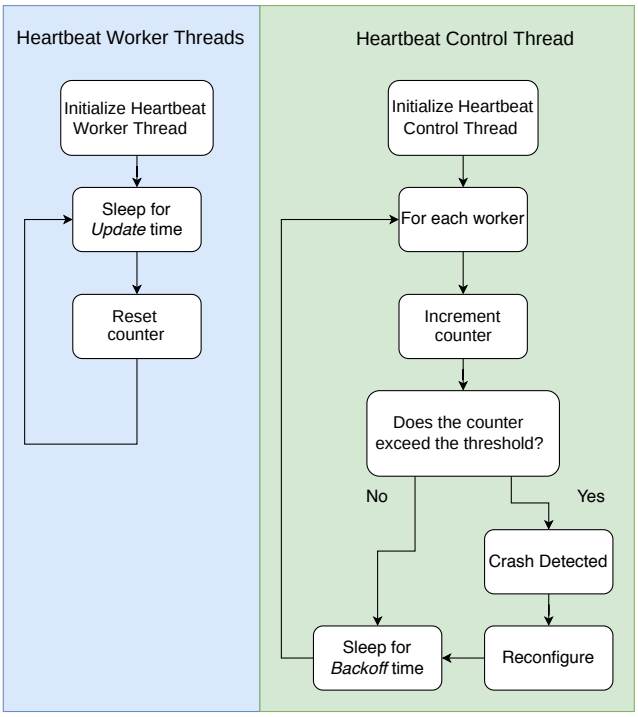

Figure 15: Overview of the heartbeat error detection mechanism with the procedure for the worker threads to the left and the heartbeat control thread to the right

4.4 Error tokens

TeamPlay channels are not only data streams, but at the same time represent dependency relations between components. If a hardware unit fails and the running computation cannot be recovered by some fault-tolerance mechanism or no such mechanism had been requested for said component, subsequent TeamPlay components in the task graph lack their proper input tokens. For a component that synchronises multiple incoming channels, e.g. `Decision` in Figure 2, this could lead to a misalignment of input tokens, i.e. the component may only receive an input token on one input channel but not on the other. Consequently, it is not activated and the successful token remains in the queue. If we recover operations of the failed component through re-allocation to a different hardware unit in the next epoch, we keep the imbalance in the token balance, and the component would always synchronise and further process tokens originating from different epochs. In some applications this might be tolerable, but generally it is not.

To address this problem we introduce error tokens. These error tokens indicate that some dependence relation could not be satisfied, but at the same time components such as `Decision` in Figure 2 are properly activated through the regular firing rule. If a component encounters an error token on any of its inports, it skips the regular computation and produces error tokens on all its outports. Thus, error tokens propagate through the entire graph and discard any intermediate data tokens that can not properly synchronised.

4.5 Checkpoint/restart

Checkpoint/restart can be implemented in TeamPlay by merely checkpointing the channels between components. Effectively, the entire state of an application resides in these buffers. This is done in practice by adding an extra buffer on each inport of a protected component. After the execution of the previous components, (i.e., the dependency components), copies of the output tokens ares made. For primitive types, this is an easy task but for user-defined types for which only a pointer is passed, the user needs to provide a copy function. When the thread executing the component fails, a new structure of input tokens is created from the checkpointed buffer and assigned to the task which takes place during the rejuvenation phase. The entire rejuvenation process, in which checkpoint/restart plays a crucial role, is illustrated in Figure 16. We will revisit this figure in full detail in Section 4.11. In normal operation, we need to remove the checkpointed data upon finishing execution and delivering the output, in order to prevent the buffers from overflowing.

4.6 Primary/backup

In primary/backup, a standby component takes over the main component when a failure is detected. Usually, this is done directly as the backup component synchronises with the active component to ensure a quick switch. In our coordination language, we do not need this behaviour as, again, the state of the application is completely saved in the FIFO buffers. We assign copies of the input tokens of the component to a number of threads equal to the number of replicas. The first thread that finishes the computation actually delivers the output. If a thread starts the computation after another thread has already delivered its answer, the thread starting the second computation can skip the task. However, it is unlikely that this behaviour is schedulable on real-time systems. Thus, we add an option to the runtime environment whether or not this form of task completion is done. Disabling this setting gives us the worst case: all threads compute even if the task is already delivered. When this setting is enabled, the task is only computed multiple times if the backup threads start while the thread that finishes first has not yet

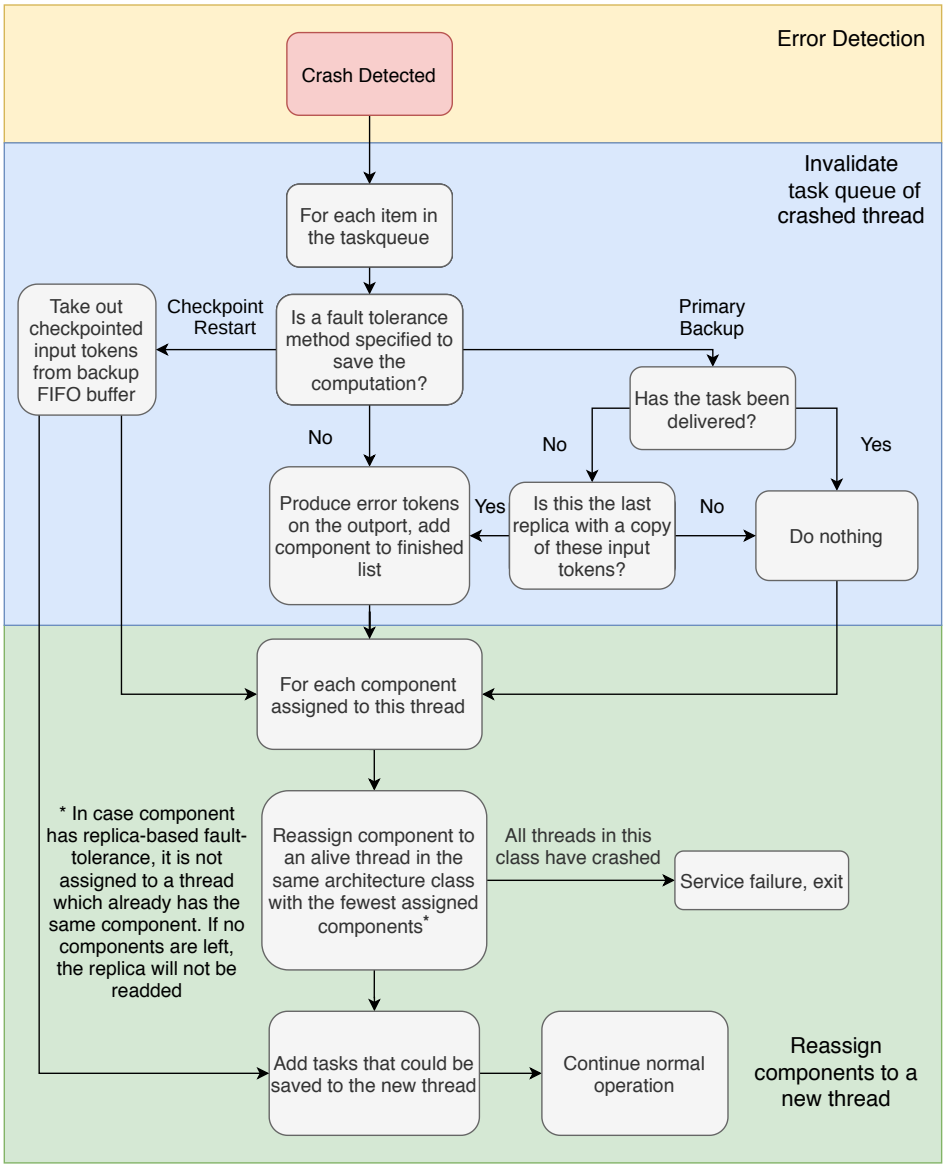

Figure 16: Illustration of our rejuvenation mechanism. The process can be split into two parts. The first part is the invalidation of the task queue of the crashed thread (the flowchart shown in the blue box up top). The second part is the reassignment of components assigned to the crashed thread, which is shown in the green box below.

finished. Just like rejuvenation for checkpoint/restart, we illustrate the primary/backup rejuvenation process in Figure 16.

4.7 N-modular redundancy and multi-version programming

For the time being our runtime system does not yet (fully) support N-modular redundancy and multi-version programming for fault-tolerance. Both can be implemented in a similar manner since from a runtime system perspective multi-version programming is a rather simple variation of N-modular redundancy. Instead of running a fixed number of identical clones of some component, we merely need to run a fixed number of different versions of the same component. Multiple versions in general are fully supported

Both N-modular redundancy and multi-version programming require the extension of our primary/backup implementation by a voting layer. What we can re-use from the primary/backup implementation, is that these processes do not need to execute at the same time. When all components finished computing, crashed or sustained a timeout according to the coordination settings, a (generated) voter component is executed. This voter requires a copy of all output tokens in order to execute the majority voting process. The

voter component is scheduled and mapped alongside all regular components of the application.

In the case of effective dual-modular redundancy, i.e. two replicas successfully completed the task and produced output tokens, we can merely detect errors, but neither correct nor mitigate them. If the voter finds the results to diverge, it generates error tokens on the outports. In the case of effective triple-modular redundancy, i.e. three or more replicas successfully completed the task, we can apply error correction, provided there is a majority for some result value(s). In this case, we deem the result(s) of the majority of replicas to be correct and forward it/them to the outports. Otherwise, we again generate error tokens on the outports. Results not forwarded anywhere are automatically discarded.

4.8 Scheduling and mapping

For rejuvenation on heterogeneous platforms we present an extension to the base runtime system. For this we need to know which thread can be reconfigured to do which tasks. In other words we need a mapping of components to threads/cores of a heterogeneous system. In time-critical systems, TeamPlay components are assigned in both the time and space dimensions to a hardware component.

For the spatial dimension of the problem we introduce the concept of a thread class. A thread class is a set of hardware cores that can run a component with identical time and energy behaviour. Our current model is still simplified as in practice components may well run on different types of hardware, albeit with different time and energy properties. The initial mapping of components to cores is inferred by the TeamPlay compiler and passed to the runtime system at deploy time.

4.9 Memory layout organisation

Usually, using threads on a system means that data is passed around using shared memory and stored in core-private caches. In our runtime, this raises the question about where the input and output data of the components should be stored. First of all, we want to keep the strain on the internal network that connects the hardware units low. Thus, we do not send the actual data back and forth between control thread and worker threads. With larger data structures, e.g. images and videos which are common in CPS(oS), this would quickly become prohibitively expensive. Instead, we store data on the hardware unit that executes a component and let components load data on request.

However, this solution causes problems with fault-tolerance methods like checkpoint/restart, as they require a copy of the data. This copy should not reside in the same memory if we take physical hardware failure into account. Therefore, we assign a *memory companion* to each core/thread. This memory companion holds the data copies as needed by the specified fault-tolerance method.

4.10 Runtime configuration file

We make use of a configuration file through which the user can control crucial parameters of the runtime system; Figure 17 shows an example with the default settings.

The individual control parameters are the following:

- `numThreads` determines the number of worker threads.
- `debug` turns debug logging on/off.

```
numThreads = 6
debug = false
sleepTime = 100
controlSleep = 1000
heartbeatTries = 10
heartbeatCheckerPrio = 10
heartbeatWorkerPrio = 15
standbyEarlyTaskCompletion = false
edgeBufferSize = 20
```

Figure 17: Example runtime configuration file

- `sleepTime` is the period of the heartbeat threads in microseconds.
- `controlSleep` is the period of the heartbeat control thread in microseconds.
- `heartbeatTries` is the threshold of the counter incremented by the heartbeat control thread.
- `heartbeatCheckerPrio` sets the real-time scheduling priority of the heartbeat control thread.
- `heartbeatWorkerPrio` sets the real-time scheduling priority of the heartbeat worker threads.
- `standbyEarlyTaskCompletion` (de-)activates preemption of backup threads in primary/backup if the primary thread has successfully delivered results.
- `edgeBufferSize` determines the global channel (FIFO buffer) size in tokens.

4.11 Rejuvenation

Strategies that deal with crash faults, checkpoint/restart and primary/backup require a rejuvenation mechanism. The rejuvenation process is illustrated in Figure 16. The process can be split into two parts: the invalidation and recovery of the task queue of the crashed thread and the reassignment of the components originally mapped to the failed hardware unit. The concrete path of the rejuvenation process depends on which fault-tolerance mechanisms are specified.

First, we explain the middle path which is taken when no fault-tolerance method is specified on the component. On this path, error tokens are produced on the outports of the components in the task queue. The component is marked as finished as the computation could not be saved by a fault-tolerance method. Then we arrive at the rejuvenation process, which works by finding the alive thread (core) of the same thread class with the least number of assigned components. The latter condition is meant for load balancing. We do not produce error tokens if a source component is present in the task queue of the crashed thread as it can simply fire again since it does not have any input tokens that need to be invalidated.

In checkpoint/restart the computation can be saved by rerunning the task with the checkpointed input tokens. Before, we must reassign components to a different thread, before the checkpointed data can be rerouted there. This rejuvenation step is the same as without fault-tolerance methods except that with checkpoint/restart, tasks exist that could potentially be saved. Note that a task cannot be saved with checkpoint/restart if it has crashed twice.

In primary/backup, a component is assigned to multiple threads. The number of threads depends on the number of replicas defined

in the coordination language. During the invalidation of the task queue, we need to check whether a task has been delivered or not. If it has been delivered then the computation does not need to be saved or invalidated. However, if a task has not delivered outputs we need to check if the crashed thread is the last replica of this task. If it is the last one, the computation cannot be saved and we execute the same steps as without a fault-tolerance mechanism. If there are still replicas assigned to this task we do not need to do anything since they can deliver the task as they already have copies of the input data. Again, we assume no repeated crashing. In primary/backup, we add an extra requirement for finding a new thread to which the task can be reconfigured to. This requirement is that there cannot be multiple assignments of the same component to the same task as this would defeat the purpose of primary/backup. If there are no threads left that follow these requirements the replica is not reassigned.

In our runtime system, the task queue is in shared memory as the control thread needs to assign new components with input data to the task queue of the thread. However, we also need to save this information in the control thread. During rejuvenation, we require a copy of this task queue in order to reassign the components of the crashed thread.

5 RELATED WORK

Coordination is a well-established computing paradigm with a plethora of languages, abstractions and approaches, surveyed in [10]. Yet, we are neither aware of any adoption of the principle in the broader domain of mission-critical cyber-physical systems, nor are we aware of energy- or time-aware approaches to coordination, let alone approaches targeting fault-tolerance.

In the area of exogenous coordination languages we must foremost mention the work on Reo [2]. The objective of Reo is in the modelling and formal property verification of coordination protocols. Reo has a graphical syntax, in which every Reo program is a labelled, directed hyper-graph, and a (or rather many) formal semantics [19]. Compared to our work, Reo is a much more theoretical approach to exogenous coordination, whereas our objective lies in the creation of a practical (and pragmatic) domain-specific language (DSL) to create executable energy-, time- and security-aware programs running on concrete machinery.

Another example of an exogenous coordination language is S-Net [16], from which we draw some inspiration and experience for TeamPlay. However, S-Net merely addresses the functional aspects of coordination programming and has left out any non-functional requirements, not to mention energy, time or fault-tolerance.

A notable exception in the otherwise fairly uncharted territory of resource-aware (functional) languages is Hume [18]. Hume was specifically designed with real-time systems in mind, and, thus, guarantees on time (and space) consumption are key. However, the main motivation behind Hume was to explore how far high-level functional programming features, such as automatic memory management, higher-order functions, polymorphism, recursion, etc can be supported while still providing accurate real-time guarantees. In particular, we are not aware of any work towards fault-tolerance in the context of Hume.

XBW [11] is a conceptual graphical computing model that uses entities similar to components to specify time behaviour and distribution properties. Contrary to our approach, this work makes use of uniform fault-tolerance that applies one fault-tolerance techniques to the whole system, whereas TeamPlay supports a fine-grained per-component specification.

Metaobject protocols (MOPs) [12, 20] change the behaviour of object-oriented language building blocks to provide non-functional concerns, like fault-tolerance, in a systematic way. When a MOP is established, these behavioural changes result in a mostly user-transparent approach. An example of using this way of working is the FRIENDS system [13].

Fault-tolerant Linda systems [6, 12, 40] are extensions of the Linda coordination language [14, 15, 42]. This coordination language uses a tuple-space in which messages can be shared between processes. Extensions focus mostly on making tuple-space operations safer and fault-tolerant utilizing redundancy, checkpoint/restart and atomics.

The Message Passing Interface (MPI) also has extensions to enable the construction of fault-tolerant programs [8, 9, 17, 22, 39]. Some extensions add structures and functions to the library, e.g., agreement algorithms and graceful error handling procedures for when nodes fail. Others focus on systematic fault-tolerance, usually with checkpoint/restart due to its low intrusiveness.

6 CONCLUSIONS

The engineering of cyber-physical systems is still dominated by low-level tools, frameworks and languages [28, 32, 37]. With the TeamPlay coordination language [35] we aim at drastically improving software engineering productivity in this domain. To this end we leverage the concept of exogenous coordination and the strict separation of concerns between computation and coordination code. We create the opportunity to actively manage non-functional properties like time and energy consumption on heterogeneous parallel computing platforms.

In our current work we leverage the design of the TeamPlay coordination approach to incorporate tolerance mechanisms against permanent and transient hardware faults while maintaining the strict separation of concerns. With the proposed extensions to the TeamPlay coordination language programmers and system integrators are equipped with fine-grained control over fault-tolerance capabilities without any cluttering of functional code. We support a total of four fault-tolerance mechanisms, namely checkpoint/restart, primary/backup, N-modular redundancy and multi-version programming. Furthermore, we devise a fully-fledged runtime environment that seamlessly runs TeamPlay coordination code with the specified fault-tolerance settings. Together language design and runtime environment facilitate experimentation with fault-tolerance mechanisms and the explicit but productive exploration of the trade-offs between improved fault-tolerance, execution time and energy consumption.

The four fault-tolerance mechanisms that we focus on for the time being themselves are not novel and have been studied in various contexts. What is to the best of our knowledge indeed novel in our work presented here is the orthogonality and the high-level nature in which we make fault-tolerance available to application

engineering in the context of cyber-physical systems. We provide fault-tolerance literally at the finger tips of programmers and system integrators. TeamPlay enables them to explore fault-tolerance features without embarking on an engineering adventure that may potentially even jeopardize the functional correctness of code.

We are currently pursuing two directions of research. First, we plan to integrate the fault-tolerance extensions of TeamPlay as described in this paper with the time- and energy-aware heterogeneous multi-core scheduling techniques that we have developed over recent years [33, 34, 36]. Second, we work on statistical methods to quantify the impact of fault-tolerance techniques on the system reliability in the presence of single-event upsets [25, 26]. Here, we particularly address weakly-hard real-time systems, where components are permitted to fail a bounded number of times in a gliding average before disaster strikes [7].

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH project). We further would like to thank the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

[1] P. Achten and M.J. Plasmeijer. 1995. The ins and outs of Clean I/O. *Journal of Functional Programming* 5, 1 (1995), 81–110.
[2] F. Arbab. 2004. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14, 3 (2004), 329–366.
[3] F. Arbab. 2006. Composition of Interacting Computations. In *Interactive Computation*, D. Goldin, S. Smolka, and P. Wegner (Eds.). Springer, 277–321.
[4] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33.
[5] R. Baheti and H. Gill. 2011. The impact of control technology: Cyber-physical systems. *IEEE Control Systems Society* 12, 1 (2011), 161–166.
[6] D.E. Bakken and R.D. Schlichting. 1995. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems* 6, 3 (1995), 287–302. https://doi.org/10.1109/71.372777
[7] G. Bernat, A. Burns, and A. Liamosi. 2001. Weakly Hard Real-Time Systems. *IEEE Trans. Comput.* 50, 4 (2001), 308–321.
[8] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. 2013. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254.
[9] A. Bouteiller. 2015. Fault-Tolerant MPI. In *Fault-Tolerance Techniques for High-Performance Computing*, T. Herault and Y. Robert (Eds.). Springer, 145–228. https://doi.org/10.1007/978-3-319-20943-2_3
[10] G. Ciatto, S. Mariani, M. Louvel, A. Omicini, and F. Zambonelli. 2018. Twenty years of coordination technologies: State-of-the-art and perspectives. In *Coordination Models and Languages, 20th International Conference, COORDINATION 2018, Madrid, Spain (Lecture Notes in Computer Science, Vol. 10852)*. Springer, 51–80.
[11] V. Claesson, S. Poledna, and J. Soderberg. 1998. The XBW model for dependable real-time systems. In *International Conference on Parallel and Distributed Systems (ICPADS 1998), Tainan, Taiwan*. IEEE, 130–138.
[12] V. de Florio and C. Blondia. 2008. A Survey of Linguistic Structures for Application-level Fault Tolerance. *Comput. Surveys* 40, 2 (2008), 1–37.
[13] J. Fabre and T. Perennou. 1998. A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach. *IEEE Trans. Comput.* 47, 1 (1998), 78–95.
[14] D. Gelernter. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7, 1 (1985), 80–112.
[15] D. Gelernter and N. Carriero. 1992. Coordination Languages and their Significance. *Commun. ACM* 35, 2 (1992), 97–107.
[16] C. Grelck, S.B. Scholz, and A. Shafarenko. 2010. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming* 38, 1 (2010), 38–67.
[17] W. Gropp and E. Lusk. 2004. Fault Tolerance in Message Passing Interface Programs. *International Journal of High Performance Computing Applications* 18, 3 (2004), 363–372. https://doi.org/10.1177/1094342004046045
[18] K. Hammond and G. Michaelson. 2003. Hume: a domain-specific language for real-time embedded systems. In *Generative Programming and Component Engineering, 2nd International Conference, GPCE 2003, Erfurt, Germany (Lecture Notes in Computer Science, Vol. 2830)*, F. Pfenning and Y. Smaragdakis (Eds.). Springer, Springer, 37–56.
[19] Sung-Shik Jongmans and Farhad Arbab. 2012. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science* 22, 1 (2012), 201–251.
[20] G. Kiczales, J. des Rivieres, and D.G. Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT Press.
[21] H. Kopetz. 2011. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer.
[22] I. Laguna, D.F. Richards, T. Gamblin, M. Schulz, B.R. de Supinski, K. Mohror, and H. Pritchard. 2016. Evaluating and extending user-level fault tolerance in MPI applications. *International Journal of High Performance Computing Applications* 30, 3 (2016), 305–319. https://doi.org/10.1177/1094342015623623
[23] W. Loeve and C. Grelck. 2020. Towards Facilitating Resilience in Cyber-physical Systems using Coordination Languages. In *13th Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE 2020)*, E. Constantinou (Ed.), Vol. 2754. CEUR Workshop Proceedings.
[24] R.E. Lyons and W. Vanderkulk. 1962. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development* 6, 2 (1962), 200–209. https://doi.org/10.1147/rd.62.0200
[25] L. Miedema and C. Grelck. 2022. Strategy Switching: Smart Fault-tolerance for Weakly-hard Resource-constrained Real-time Applications. In *Software Engineering and Formal Methods, 20th International Conference, SEFM 2022 (Lecture Notes in Computer Science, Vol. 13550)*. Springer, 129–145.
[26] L. Miedema and C. Grelck. 2023. Change of plans: optimizing for power, reliability and timeliness for cost-conscious real-time systems. In *26th Euromicro Conference on Digital System Design (DSD 2023), Durrës, Albania*. Euromicro.
[27] M. Oriol, T. Gamer, T. de Gooijer, M. Wahler, and E. Ferranti. 2013. Fault-tolerant fault tolerance for component-based automation systems. In *4th International ACM SigSoft Symposium on Architecting Critical Systems (ISARCS 2013), Vancouver, Canada*, P. Kruchten and S. Malek (Eds.). ACM, 48–59. https://doi.org/10.1145/2465470.2465471
[28] K.J. Park, R. Zheng, and X. Liu. 2012. Cyber-physical systems: Milestones and research challenges. *Computer Communications* 36, 1 (2012), 1–7. https://doi.org/10.1016/j.comcom.2012.09.006
[29] Z. Peng. 2010. Building reliable embedded systems with unreliable components. In *IEEE International Conference on Signals and Electronic Circuits (ICSES 2010), Gliwice, Poland*. IEEE, 9–13.
[30] S.L. Peyton Jones and J. Launchbury. 1995. State in Haskell. *Lisp and Symbolic Computation* 8, 4 (1995), 293–341.
[31] S. Poledna. 1996. *Fault-tolerant real-time systems: The problem of replica determinism*. Springer International Series in Engineering and Computer Science (SECS), Vol. 345. Kluwer Academic Publishers.
[32] R.Y. Rajkumar, I. Lee, L. Sha, and J.A. Stankovic. 2010. Cyber-physical systems: The next computing revolution. In *47th Design Automation Conference (DAC 2010), Anaheim, USA*. IEEE, 731–736. https://doi.org/10.1145/1837274.1837461
[33] J. Roeder, A.D. Pimentel, and C. Grelck. 2023. GCN-based Reinforcement Learning Approach for Scheduling DAG Applications. In *Artificial Intelligence Applications and Innovations, 19th IFIP WG 12.5 International Conference, AIAI 2023, León, Spain (IFIPAICT, Vol. 676)*. Springer, 121–134. https://doi.org/10.1007/978-3-031-34107-6_10
[34] J. Roeder, B. Rouxel, S. Altmeyer, and C. Grelck. 2020. Energy-aware Scheduling of Multi-version Tasks on Heterogeneous Real-time Systems. In *36th ACM/SIGAPP Symposium on Applied Computing (SAC 2021)*. ACM, 500–510.
[35] J. Roeder, B. Rouxel, S. Altmeyer, and C. Grelck. 2020. Towards Energy-, Time- and Security-aware Multi-core Coordination. In *Coordination Models and Languages, 22nd International Conference, COORDINATION 2020, Malta (Lecture Notes in Computer Science, Vol. 12134)*, S. Bliudze and L. Bocchi (Eds.). Springer, 57–74.
[36] J. Roeder, B. Rouxel, and C. Grelck. 2021. Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems. In *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore*. IEEE.
[37] A. Romanovsky. 2007. A looming fault tolerance software crisis? *ACM SIGSOFT Software Engineering Notes* 32, 2 (2007), 1–4.
[38] C. Rusu, R. Melhem, and D. Mossé. 2005. Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing* 1, 2 (2005), 271–283.
[39] N. Sultana. 2019. *Toward a Transparent, Checkpointable Fault-Tolerant Message Passing Interface for HPC Systems*. Ph. D. Dissertation. Auburn University, USA.
[40] F. Tam, M. Woodward, and G. Toppong. 1995. FT-Linda: a coordination language for programming distributed fault-tolerance. In *IEEE Singapore International Conference on Networks and International Conference on Information Engineering*. IEEE, 649–653. https://doi.org/10.1109/SICON.1995.526368
[41] A.S. Tanenbaum and M. van Steen. 2007. *Distributed Systems: Principles and Paradigms*. Prentice-Hall.
[42] G. Wells. 2005. Coordination languages: Back to the future with Linda. In *2nd International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT 2005), Glasgow, UK*. 87–98.