Towards Facilitating Resilience in Cyber-Physical Systems using Coordination Languages

Wouter Loeve
University of Amsterdam
Amsterdam, Netherlands
wouterloeve0@gmail.com

Clemens Grelck
University of Amsterdam
Amsterdam, Netherlands
c.grelck@uva.nl

June 15, 2020

Abstract

Failures in cyber-physical systems can be detrimental. This is why these systems often employ fault-tolerance methods that allow their service to keep working even in the event of hardware or software failures. These methods are often intertwined in computation code, which makes it complex. We take the first steps in reducing this complexity by means of coordination. This coordination language [JSC20] uses a component-based approach and aims at separating the structure and non-functional property management from the computation code. The computation code can then focus on being functionally correct. We extend this coordination language by introducing the specification and management of fault-tolerance strategies and we introduce additional structures to manage these strategies in an expressive way.

1 Introduction

Cyber-physical systems (CPS) is a recent computing paradigm which involves systems that interact with both the hardware they run on and the physical world around us [BG11]. Examples of this can be found in several areas such as self-driving cars, autonomous drones, robotics, and building & environmental control.

Commonly, these systems also have non-functional requirements. These requirements are manifested in facets such as timing, energy-consumption, security, and robustness [PZL12; Raj+10]. These requirements often involve trade-offs; e.g., executing an action in less time often involves a higher power consumption [JSC20]. Adding robustness or fault-tolerance often involves redundancy which also incurs a higher power consumption and a higher response time.

Additionally, many safety-critical systems (especially in multi- or heterogeneous systems) require extensive testing, validation, and verification. This is often hard to do when non-functional properties, like fault-tolerance, are intertwined with computational code. In the CPS paradigm, the scientific community has been calling out to the creation of higher-level abstraction layers to alleviate the burden on the programmer [PZL12; Rom07; Raj+10]. The solution we propose resides in the paradigm of coordination languages [JSC20; GC92].

Coordination languages can be used to manage the interaction between separate activities or components into an often parallel system [ACH98]. For our use-case within CPS, we envision that the coordination language can run on heterogeneous architectures thus requiring synchronisation and parallelisation [JSC20]. Roeder et al. have taken the first steps towards a coordination language that enables the management of non-functional aspects of cyber-physical systems [JSC20]. This language facilitates separation of concerns between coordination and computation code. Coordination code defines the structure of the application and manages non-functional properties on a high level. In this way, the computation code can focus mostly on being functionally correct while letting the coordination code manage the non-functional properties.

In this work, we make the following contributions:

1. Facilitate the management of fault-tolerance or resilience strategies.

Copyright © by the paper's authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In: A. Editor, B. Coeditor (eds.): Proceedings of the 13th Seminar on Advanced Techniques Tools for Software Evolution, Amsterdam, Netherlands, 01-07-2020, published at http://ceur-ws.org

2. Introduce templating that facilitates code reuse and add other structures such as cascading and inheritance that makes management of fault-tolerance and other future options easier.

2 Existing coordination language

We illustrate the existing coordination language [JSC20] by means of an example. Figure 1 shows a subsystem in a car with two sensors leading to a decision controller which in turn leads to an actuator which can interact with its environment. Figure 2 shows the code that goes with this example.

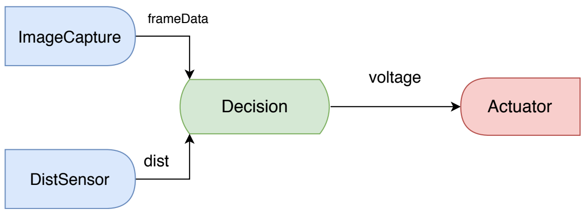

Figure 1: Example illustrating how a program looks like in the existing coordination language.

```
1  app {
2    datatypes {
3      (num, "uint32_t")
4      (frame, "jpegFrame*")
5    }
6    components {
7      DistSensor {
8        outputs [ (dist, num) ]
9      }
10     ImageCapture {
11       outputs [ (frameData, frame) ]
12     }
13     Decision {
14       inputs [ (dist, num) ]
15       outputs [ (voltage, num) ]
16     }
17     Actuator {
18       inputs [ (dist, num) ]
19     }
20   }
21   edges {
22     DistSensor.dist -> Decision
23     ImageCapture.frameData -> Decision
24     Decision.voltage -> Actuator
25   }
26 }
27
```

Figure 2: Example illustrating the existing coordination language.

The existing coordination language is a component-based language [JSC20]. An application definition (line 1) has three major regions: `datatypes` (line 2) in which data-types used in ports are defined, `components` (line 6), in which components are defined and `edges` (line 21) in which edges between components are specified.

Components themselves are stateless, state is captured in data-tokens that are consumed when a component fires. Like Petri-nets, a component can fire once all required input tokens are ready. Each component can have the following ports: inputs (line 14) for input tokens and outputs (line 15) for output tokens.

Working with stateless components ensures that the program structure is captured by the coordination model while the code for a single component can focus mostly on the functional aspects of computation, i.e., getting the correct result. The coordination language can then be used to manage **e**nergy, **t**ime and **s**ecurity (ETS) by supplying different versions of the same component with different properties. These properties have to be defined separately and are used by the scheduler to decide which version should be used. For example, a component used for encryption can use a faster (but weaker) algorithm to save time and energy while compromising on security. Which version is being used depends on the mission's state, so that the resources available are used optimally and are appropriate for the situation as directed by the scheduler. We leave the details of this approach out due to limited space.

Ports between components are connected with edges as can be seen in line 21-24 in the code example and the schematic example (Figure 1). Edges are created by connecting a component and output with another component and an input separated by an arrow. Edges have a FIFO buffer which holds tokens waiting to be consumed by components once they have enough tokens for their inputs. It is also possible to define that multiple tokens have to be consumed before a component can be fired.

3 Extension 1: Fault-tolerance

We extend the language to facilitate fault-tolerance or resilience, meaning that it aims to avoid service failures when faults occur [Avi+04]. We opted for a user-directed approach so that the user can specify which of the pre-defined options to apply in different parts of the application. This is due to major semantic challenges in having a compiler or scheduler analyse the criticality of a component in the application as a whole. Furthermore, the way fault-tolerance is implemented and achieved needs to be transparent to the programmer, since, in critical applications, errors or ambiguity can be costly.

We selected the following methods for implementation in the language.

N-Modular Redundancy (NMR) (`nModular`) runs n identical processes followed by a voting process to reduce the risk of deviating components caused by faults [LV62; TV07; Ori+13]. In this method, it is possible to change the value of n but also the number of voting processes to run since these can also be faulty, thus decreasing resilience against faults. Without a dedicated monitoring component, it is impossible to tell whether a component or process involved in triple-modular redundancy has crashed. For this, we have a timeout system in which can be specified how long the finished processes should wait before the slow process is killed off because it may have crashed.

Primary-backup or standby (`standby`) runs multiple backup processes in addition to an active or primary process. The output of the latter is used while the backup processes can take over the job of the active component in case it fails [Ori+13]. Like in NMR, the number of replicas can be changed. It is also possible to tweak the degree of synchronisation the standby component has compared to the primary component. To this end, we supply three flavours: cold, warm, and hot [Ori+13].

N-version programming (`nVersion`) runs n different processes conforming to the same specification [Pen10]. For this method, one can specify the n value of each version separately, as we use the `version` specification already present in the language to define versions.

Checkpoint-restart (`checkpoint`) saves a backup of the application's state which can be restored when a failure occurs [TV07; Sul+19]. For this method, we allow setting the frequency of the checkpoints as well as enable the removal of faulty resources from the resource pool (also called shrinking recovery).

Fault-tolerance options are specified with the keywords defined above in brackets, optionally followed by a set of method-specific options to tweak the way fault-tolerance is used.

Fault-tolerance methods can be applied directly to a component by adding the above options directly in a component definition as can be seen in Figure 3, line 13. Alternatively, these options can be put into generic profiles (defined on line 2) which in turn can be added inside the `profiles` keyword (line 12). The order in which profiles are added to a component matters, a given profile is overwritten by subsequent ones. This way of cascading enables the programmer to define generic profiles to use in the whole application, adding, and removing options to suit the specific need of a component. In the example, we have a mix of inline-specification and profiles. Directly defined options always overwrite profiles given to a component.

```
1  profiles {
2    TMR {
3      nModular {
4        replicas 3
5        votingReplicas 2
6      }
7    }
8  }
9  components {
10   Decision {
11     outputs [ (voltage, num) ]
12     profiles [ TMR ]
13     nModular { replicas 4 }
14   }
15 }
16
```

Figure 3: Cascading inline and separated profile definition. The inline option will overwrite the `replicas 3` option of the TMR profile because the inline option is more specific than a profile. Other options, in this case the `votingReplicas` option will be merged with the inline profile.

4 Extension 2: Introducing reusability in options and components

To assist in managing the fault-tolerance options defined above and reducing duplication in the coordination language code we will introduce new constructs in this section. While the motivation for these additions originates from managing fault-tolerance options, they are also meant to work with existing and future options of the language.

Groups of components responsible for a specific functionality may have a similar criticality, this means that they require the same or similar options. To support this, we add the notion of sub-graphs to the language as shown schematically in Figure 4 and in code in Figure 5. Sub-graphs can contain multiple components, as can be seen on lines 11 and 15. On the outside, they are treated like normal components as they have the same kinds of ports as regular components. For example, the sub-graph has an output on line 8. Like in regular components, fault-tolerance options can be specified directly and by using profiles (line 10).

Options specified on the top level of the sub-graph are automatically inherited by the components inside unless indicated otherwise. One can stop inheritance by removing a fault-tolerance method or option, which can be done by placing the `remove` keyword before the name of an option or method, e.g., `remove nModular`. In addition, to remove, we introduce the `vital` keyword, which can be put in front of an option or fault-tolerance method: `vital replicas 4`. `vital` indicates that an option cannot be overwritten.

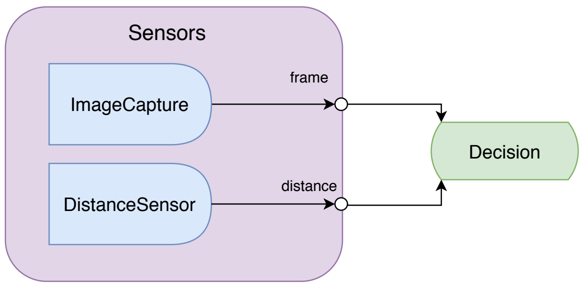

Figure 4: Sub-graph schematic example, `Sensors` contain two sensors, `ImageCapture` and `DistanceSensors` which present their outputs to the sub-graph level which in turn lead to the decision-component.

Edges between components inside a sub-graph are defined inside the sub-graph itself using the same `edges` keyword as root-level components (line 18-23). A major difference is that edges going to ports in and outside the component need to be defined by using the `in` and `out` keywords instead of using the id of a component in the edge definition. This can be seen on line 20 and 23, `out` is a handle to the current sub-graph. These two components present their output to the outside of the graph in the form of an output. This output can be used in the top-level `edges` specification coming from that sub-graph.

Another use-case of sub-graph inheritance is to have a way to specify the application of fault-tolerance methods to edges between components. Checkpoint restart requires such an approach as tokens are saved in FIFO-buffers which correspond to the edges they reside in. The content of these buffers can then be written to a backup-space to be utilised as a checkpoint. Checkpoint restart can (only) be applied by specifying the method in the sub-graph itself using a profile or in-line profile definition. This approach is only valid under the assumption that the components inside sub-graphs have similar criticality.

The existing language has no way of reusing components, a component definition is tied to an instance of the component. This inhibits code reuse. In order to solve that, we have to decouple the instantiation of a component from its definition. To this end, we define a new keyword and application region, called `templates`. Contrary to components or sub-graphs defined in `components`, components, and sub-graphs defined inside `templates` are not automatically instantiated but require explicit instantiation inside the `components` keyword.

Figure 6 shows an extension of the previous example to show the templating mechanism. `Sensors` has been moved to `templates` and is instantiated twice as seen on lines 26 and 28. This is indicated by the name of the template followed by the de-

```
1  profiles {
2    TMR {
3      nModular { replicas 3 }
4    }
5  }
6  components {
7    Sensors {
8      outputs [ (distance, num)
9        (frameData, frame) ]
10     profiles [ TMR ]
11     ImageCapture {
12       outputs [ (frameData, frame) ]
13       nModular { replicas 4 }
14     }
15     DistanceSensor {
16       outputs [(dist, num)]
17     }
18     edges {
19       // out refers to subgraph output
20       ImageCapture -> out.frameData
21       // frameData references specified
22       // output
23       DistanceSensor -> out.distance
24     }
25
26   }
27   Decision {
28     inputs [ (frameData, frame)
29       (distance, num) ]
30   }
31 }
32 edges {
33   Sensors.distance -> Decision.distance
34   Sensors.frameData -> Decision.frameData
35 }
36
```

Figure 5: Sub-graph example which mirrors 4. The TMR profile applies to all sub-components: ImageCapture and Distance-Sensor.

sired handle (i.e., id) to the instance, the brackets will be featured in the next paragraph. The handle is then used to define the edges between the instance and other components (lines 38-45).

To assist in slight variations of options between different instances of templates, we introduce a parameter-mechanism. Parameters can be added in the template definition, as can be seen on line 8 in Figure 6. Occurrences of the parameter-names in settings-related areas will be replaced by the value given during instantiation (lines 26 and 28). If no parameters are desired, the brackets can be left out.

For now, a single parameter corresponds to a single value or keyword. This is because of two reasons: we have not encountered a use-case where entire lines put into a template's parameters was beneficial and it is difficult to give useful user feedback by the compiler when entire lines or blocks can be given as parameters.

```
profiles {
  TMR {
    nModular { votingReplicas 3 }
  }
}
templates {
   // Parameters can be used throughout
    the template
  Sensors(numReplicas, rootProfile) {
    outputs [(dist, num),
      (frameData, frame)]
    profiles [ rootProfile ]
    ImageCapture {
      outputs [ (frameData, frame) ]
      nModular { replicas numReplicas }
    }
    DistanceSensor {
      outputs [(dist, num)]
    }
    edges {
      ImageCapture -> out.frameData
      DistanceSensor -> out.dist
    }
  }
}
components {
  Sensors(3, TMR) SensorFront
  // Parameters can be left empty
  Sensors(4, ) SensorRear

  Decision {
    inputs [
      (distFront, num) (frameFront,frame)
      (distRear, num) (frameRear, frame)
    ]
  }
}
edges {
  SensorFront.dist -> Decision.distFront
  SensorFront.frameData ->
    Decision.frameFront

  SensorRear.dist -> Decision.distRear
  SensorRear.frameData ->
    Decision.frameRear
}

```

Figure 6: Example of specifying parameters in a template. In `SensorFront`, NMR will be utilised with three replicas and the TMR profile will be applied to all sub-components. `SensorRear` on the other hand will have four replicas and no other profile will be applied.

5 Related work

For related work, we have compiled a short list of approaches to providing fault-tolerance. We have been unable to find an existing approach that provides application-specific (i.e., non-systematic) fault-tolerance whilst being non-intrusive to computation code.

XBW is a conceptual graphical computing model that uses entities similar to components and sub-graphs to specify time behaviour and distribution properties [CPS98]. Contrary to our approach, this work makes use of systematic fault-tolerance which effectively uses the same techniques (active replication, i.e., NMR) on the whole system.

Metaobject protocols (MOPs) [KDB91; FB08] 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 [FP98].

Fault-tolerant Linda systems [TWT95; BS95; FB08] are extensions of the Linda coordination language [Gel85; GC92; Wel05]. 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.

Message Passing Interface (MPI) also has extensions to enable the construction of fault-tolerant programs [Lag+16; GL04; Bou15; Bla+13; Sul+19]. 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 Conclusion and next steps

In cyber-physical systems, there is still a lot to be done in the area of creating tools, frameworks, and languages to aid the programmer in processes related to software evolution like creating and maintaining [PZL12; Rom07; Raj+10]. With this project, we take the first steps towards facilitating separation of concerns between computation and coordination code. This provides opportunities to manage non-functional properties like fault-tolerance separately from computation code.

In this work, we have extended and improved an existing coordination language [JSC20]. We have introduced: templating with parameters, multiple instances of the same component and profiles (along with cascading and inheritance). Furthermore, we have introduced a way to specify fault-tolerance methods into the coordination language and how they can be managed.

In terms of next steps in this research, we have already started to implement the language features in a compiler. We aim at creating a simulator to validate our approach and allow prototyping of the specification (i.e., the language), management, and execution of fault-tolerance strategies. Additionally, we started working on a visualisation for the simulator to make the fault-tolerance behaviour more transparent to the programmer.

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH project)

References

[ACH98] F. Arbab, P. Ciancarini, and C. Hankin. "Coordination languages for parallel programming". In: *Parallel Computing* 24.7 (1998), pp. 989–1004. ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(98)00039-8. URL: http://www.sciencedirect.com/science/article/pii/S0167819198000398.

[Avi+04] A. Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[BG11] Radhakisan Baheti and Helen Gill. "Cyber-physical systems". In: *The impact of control technology* 12.1 (2011), pp. 161–166.

[Bla+13] Wesley Bland et al. "Post-failure recovery of MPI communication capability: Design and rationale". In: *The International Journal of High Performance Computing Applications* 27.3 (2013), pp. 244–254.

[Bou15] Aurélien Bouteiller. "Fault-Tolerant MPI". In: *Fault-Tolerance Techniques for High-Performance Computing.* Ed. by Thomas Herault and Yves Robert. Cham: Springer International Publishing, 2015, pp. 145–228. ISBN: 978-3-319-20943-2. DOI: 10.1007/978-3-319-20943-2_3. URL: https://doi.org/10.1007/978-3-319-20943-2_3.

[BS95] D. E. Bakken and R. D. Schlichting. "Supporting fault-tolerant parallel programming in Linda". In: *IEEE Transactions on Parallel and Distributed Systems* 6.3 (1995), pp. 287–302. ISSN: 2161-9883. DOI: 10.1109/71.372777.

[CPS98] V. Claesson, S. Poledna, and J. Soderberg. "The XBW model for dependable real-time systems". In: *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No.98TB100250).* 1998, pp. 130–138.

[FB08] Vincenzo De Florio and Chris Blondia. "A Survey of Linguistic Structures for Application-Level Fault Tolerance". In: *ACM Computing Survey.* 40.2 (May 2008). ISSN: 0360-0300. DOI: 10.1145/1348246.1348249. URL: https://doi.org/10.1145/1348246.1348249.

[FP98] J. Fabre and T. Perennou. "A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach". In: *IEEE Transactions on Computers* 47.1 (1998), pp. 78–95.

[GC92] David Gelernter and Nicholas Carriero. "Coordination languages and their significance". In: *Communications of the ACM* 35.2 (1992), pp. 96–108.

[Gel85] David Gelernter. "Generative communication in Linda". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.1 (1985), pp. 80–112.

[GL04] William Gropp and Ewing Lusk. "Fault Tolerance in Message Passing Interface Programs". In: *The International Journal of High Performance Computing Applications* 18.3 (2004), pp. 363–372. DOI: 10.1177/1094342004046045. eprint: https://doi.org/10.1177/1094342004046045. URL: https://doi.org/10.1177/1094342004046045.

[JSC20] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. "Towards Energy-, Time and Security-aware Multi-core Coordination". In: *22nd International Conference on Coordination Models and Languages (COORDINATION 2020), Malta* (2020).

[KDB91] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol.* MIT press, 1991.

[Lag+16] Ignacio Laguna et al. "Evaluating and extending user-level fault tolerance in MPI applications". In: *The International Journal of High Performance Computing Applications* 30.3 (2016), pp. 305–319. DOI: 10.1177/1094342015623623. eprint: https://doi.org/10.1177/1094342015623623. URL:

https://doi.org/10.1177/1094342015623623.

[LV62] R. E. Lyons and W. Vanderkulk. “The Use of Triple-Modular Redundancy to Improve Computer Reliability”. In: *IBM Journal of Research and Development* 6.2 (1962), pp. 200–209. ISSN: 0018-8646. DOI: 10.1147/rd.62.0200.

[Ori+13] Manuel Oriol et al. “Fault-tolerant fault tolerance for component-based automation systems”. In: *Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems*. 2013, pp. 49–58.

[Pen10] Z. Peng. “Building reliable embedded systems with unreliable components”. In: *ICSES 2010 International Conference on Signals and Electronic Circuits*. 2010, pp. 9–13.

[PZL12] Kyung-Joon Park, Rong Zheng, and Xue Liu. “Cyber-physical systems: Milestones and research challenges”. In: *Computer Communications* 36 (2012), pp. 1–7.

[Raj+10] R. Yogesh Rajkumar et al. “Cyber-physical systems: The next computing revolution”. In: *Design Automation Conference* (2010), pp. 731–736.

[Rom07] Alexander Romanovsky. “A looming fault tolerance software crisis?” In: *ACM SIGSOFT Software Engineering Notes* 32.2 (2007), pp. 1–4.

[Sul+19] Nawrin Sultana et al. “Toward a Transparent, Checkpointable Fault-Tolerant Message Passing Interface for HPC Systems”. In: (2019).

[TV07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[TWT95] F. Tam, M. Woodward, and G. Toppong. “FT-Linda: a coordination language for programming distributed fault-tolerance”. In: *Proceedings of IEEE Singapore International Conference on Networks and International Conference on Information Engineering ’95*. 1995, pp. 649–653. DOI: 10.1109/SICON.1995.526368.

[Wel05] George Wells. “Coordination languages: Back to the future with linda”. In: *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05)*. 2005, pp. 87–98.