Towards Time-, Energy- and Security-aware Functional Coordination

Julius Roeder
University of Amsterdam
Amsterdam, Netherlands
J.Roeder@uva.nl

Benjamin Rouxel
University of Amsterdam
Amsterdam, Netherlands
Benjamin.Rouxel@uva.nl

Clemens Grelck
University of Amsterdam
Amsterdam, Netherlands
C.Grelck@uva.nl

ABSTRACT

Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches. An application organised according to the coordination paradigm consists of a collection of interacting components. Yet, we are neither aware of any adoption of the principle in the broader domain of low-powered safety-critical embedded systems, nor are we aware of time, energy and security aware approaches to coordination.

We aim at contributing to the community by bringing a coordination approach to real-time applications with the establishment of a Domain Specific Language (DSL). Our coordination workflow considers time and other non-functional properties, such as energy and security, as first-class citizens in application designs. We therefore aim at building a complete toolchain and workflow to compile a coordinate application to a final executable.

ACM Reference format:
Julius Roeder, Benjamin Rouxel, and Clemens Grelck. 2020. Towards Time-, Energy- and Security-aware Functional Coordination. In *Proceedings of International Symposium on Implementation and Application of Functional Languages, Singapore, September 2019 (IFL'19),* 7 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Designing secure and safe Cyber Physical Systems (CPS) requires a tremendous amount of validations and certifications. To facilitate the whole process, a good design practice is to separate code source development from application structure design. This separation allows to start the validation at an early stage of the system design.

Coordination programming paradigm seems to us a promise solution as it allows to separate the high-level view of the structure from the low-level application units. An application organised according to the coordination paradigm consists of a collection of interacting components. Components, also known as actors or tasks, represent application features, sequential building blocks of application, implemented in a general-purpose programming language [11]. Given the focus of the safety-critical embedded systems domain, we exclusively work with the system-level programming language *C*. Hence, a component is technically a callable *C* function with certain restrictions on its functional behaviour, together with a set of non-functional properties, i.e. timing, energy and security.

The coordination language emphasizes communication, concurrency and synchronisation (referred to by the term *coordination*). It aims at describing component interactions in terms of precedences and data exchange. In contradiction with streaming languages (e.g. [23]) or data-flow languages (e.g. Lustre [1]), a coordination language is independent from the actual code, but it dictates, to the scheduler, how this code should be executed.

We aim at contributing to the community by bringing a coordination approach to real-time applications with the establishment of a Domain Specific Language (DSL). Our coordination workflow considers time and other non-functional properties, such as energy and security, as first-class citizens in application designs. We therefore aim at building a coordination language along with a complete toolchain to compile a coordinate application to a final executable.

Security with Cyber Physical System (CPS) relies on the implementation of proven protocols, encryption algorithms or scheduling obfuscation techniques. Such methods harden the system but also increases its requirement regarding computation and memory. Nevertheless some systems do not require to operate at a maximum security level all the time. For example, a military drone used for area recognition could adapt its security protocol in accordance with the mission state such as:

- low security level: taking off or landing from/to the base station,
- medium security level : navigating to/from the mission area,
- high security level : on site recognition.

When performing in low security level state, a drone can use a less resilient encryption system to send data to the base station, while it can use the more secure one when on mission site. This adaptation can result in less computation and therefore in power energy saving and longer effective mission time, while still guaranteeing the security and integrity of the data at all time.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Our solution is to embed different versions of the same task but with different security, time, and energy properties similarly to [19]. The scheduler will then be responsible to chose proper task versions according to current mission state.

We target CPS platforms executing on COTS multi-core heterogeneous processors where time, security and energy are safety and mission-critical. Our approach is not yet to improve timing analyses or, at first, scheduling policies themselves, but to make those secure and efficient by controlling the whole design path from specification to code generation. Hence, our first goal is to describe how to exploit our coordination language and how to integrate its semantic into state-of-the-art scheduling techniques, e.g. [18, 21]. And second, we will devise new scheduling policies that will execute smart scheduling decisions to achieve our goals regarding time, security and energy.

2 LITERATURE REVIEW

2.1 Coordination Language

Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches surveyed in [5]. Yet, we are neither aware of any adoption of the principle in the broader domain of critical embedded systems, nor are we aware of time&security-aware approaches to coordination. An example is the coordination language S-Net [11], from which we draw inspiration and experience for our Teamplay Coordination Language. However, like other coordination approaches S-Net merely addresses the functional aspects of coordination programming and has left out any non-functional requirements, not to mention time and security, in particular.

Notable exceptions in the otherwise pretty much uncharted territory of time&energy-aware coordination is Hume [13]. It was specifically designed with real-time systems in mind. Thus, guarantees on time (and space) consumption are key to Hume. 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, ... can be supported while still providing accurate real-time guarantees.

Another term referring to Coordination technology is "in-the-large" programming. It aims at describing a large view of an application, or its structure, while "in-the-low" programming considers only the implementation of each feature. Bondavalli et al. [4] presents a simple "in-the-large" programming language to describe the structure of a graph-based application. However they only model what we call component and simple edge, leaving time and security property aside. Their simple language also do not account for multi-version component nor complex edge structure as we do.

2.2 Data-flow programming language

Lustre [3, 12] was designed to program reactive system, such as automatic control and monitoring systems. Compared to general purpose programming language, it models the flow of data designates a low level programming language. The idea is to represent actions done on data at each time tick, link an electric circuit. The tick can be extended to represent periods and release times for tasks, but still an action is required to describe outputs for each tick (like reusing the last produced data). Compared to most dataflow programming language, Lustre is synchronous which seems necessary for time-sensitive applications. However, Lustre does not decorrelate the program source code from its structure as we do. The flow of data is extracted by the compiler through data dependencies of variables. We believe that separating the design of an application and its program source code is a good practice to facilitate analyses, certifications and increase reusability. We aim at expressing the flow of data with a much simpler and more explicit approach. We also act at a higher level by focusing on the interaction of components considered as black boxes. Hence, our approach is orthogonal to Lustre which can be used to program the inner side of each component.

In [1], Lustre is extended with meta-operators to integrate, as an intermediate representation (called Lustre++), a complete Model-Based Design tools from a high-level Simulink model to a low-level implementation. Still, this extension does not separate the design of the program structure and its actual code source. And Lustre++ remains at a too low level to only represent application structure as we intend to do.

In general, dataflow programming language include both application structure and code. Some are asynchronous [] and therefore unusable for time-sensitive applications, while synchronous one (Lustre, Esterel) [] aim at bringing timing constraint into programming paradigm.

2.3 Data-flow programming and graph description

On the other hand, we were also not able to find a graph description language from the broad world of dataflow paradigm, that allows us to fulfil our needs in term of time and security. A promising approach was the Dataflow Interchange Format (DIF) [14] which captures essential structure information while hiding implementation details of computation blocks. But timing and security are absent of its definition. StreamIT [23] language also describes graph-based streaming applications but it is restricted to fork-join graphs while we need to support arbitrary graphs with possible multiple sources/sinks.

From all existing coordination languages or dataflow description language, TCL gives the possibility :(1) to describe multiple versions of the same component to chose the right level of security according to the mission state, (2) to define stateful and stateless actors, such as in StreamIT [23], (3) to select the dataflow path depending on environmental or scheduling decisions.

2.4 Scheduling

2.4.1 Multi-version scheduling. Lastly, a multi-version scheduling technique is described in [17]. They employ multiple

version of tasks where each version different in term of energy usage. However they do not detail how to model such application. In addition, we target security problem, and we envision, in a near future, to join our techniques to provide full energy/time/security-aware scheduling policies.

2.4.2 Multi-mode scheduling.

2.5 Model-based paradigm

Model-Based Design (MBD) decorrelates source code programming and application structure with a high-level model design. One of the most known model-based programming language is the Unified Modelling Language (UML) [1] and a massive amount of UML profiles exists on embedded systems, with or without real-time capabilities.

SysML is one of the profile that target embedded systems, but it lakes from real-time capabilities.

Simulink does not have a denotational semantic, thus it is impossible to prove its faithfulness/correctness, (faithful between the generated code and the simulink model).

RW of modelling languages in section 2.3 of https://tel.archives-ouvertes.fr/tel-01773786/document (french thesis)

2.5.1 AUTOSAR. The objective of AUTomotive Open System ARchitecture (AUTOSAR) [10] is to establish an open industry standard for the automotive software architecture between suppliers and manufacturers. The standard comprises a set of specifications describing software architecture components and defining their interfaces. Software components are linked to a common interface that is recognized by an AUTOSAR runtime environment. AUTOSAR model is based on UML. The task model behind AUTOSAR includes periodic, sporadic, burst, concrete and arbitrary arrival patterns. With its event chains, AUTOSAR seems able to provide dependent tasks.

In [2], they studied the system model and what kind of information is available and so what kind of scheduling is possible with it. They identified the four groups of information required to perform a schedulability test. Then on a case study, they map what is provided by AUTOSAR to these groups. They finally apply the schedulability test for distributed systems with fixed priorities among dependent periodic tasks.

Even with its event chain, AUTOSAR fails to represent dataflow application.

2.5.2 AADL. Architecture Analysis & Design Language (AADL) [8] targets real-time system design. It provides formal modeling concepts for the description and analysis of application systems architecture in terms of distinct components and their interactions. It supports early predictions and analyses including performance, schedulability and reliability.

All common features to represent real-time systems task models are presents including (non-exhaustive): tasks, dependencies, period, deadline, etc .... But, there is not possibility to represent SDFs, CSDFs, or other multi-rates data transfers between components.

2.5.3 MoSaRT.

2.5.4 MARTE. MARTE [20] has become the standard for embedded real-time system engineering. It provides a common way to describe hardware and software aspects and how components interact with each other. However there is no mean to express different versions of a component and leave the scheduler decides which one to use.

2.5.5 Time4Sys. Time4Sys seems abandoned in 2018.

2.6 Unclassified yet

[6] aims at increasing parallelism while guaranteeing safety, their *HEPTAGON* language also includes source code of the application which require an additional step to get the binary. Their execution model separate communication from computation, they implement communication channels on a FPGA while computation occur on ARM cores. Mixed-criticality

3 COORDINATION WORKFLOW

An application organised according to the coordination paradigm consists of a collection of interacting components. Components are meaningful, sequential application building blocks, implemented in a general-purpose programming language [11]. Given the focus of critical embedded systems domain, we exclusively work with the system-level programming language *C*. Hence, a component is technically a callable *C* function with certain restrictions on its functional behaviour, together with a set of non-functional properties i.e. timing, energy and security.

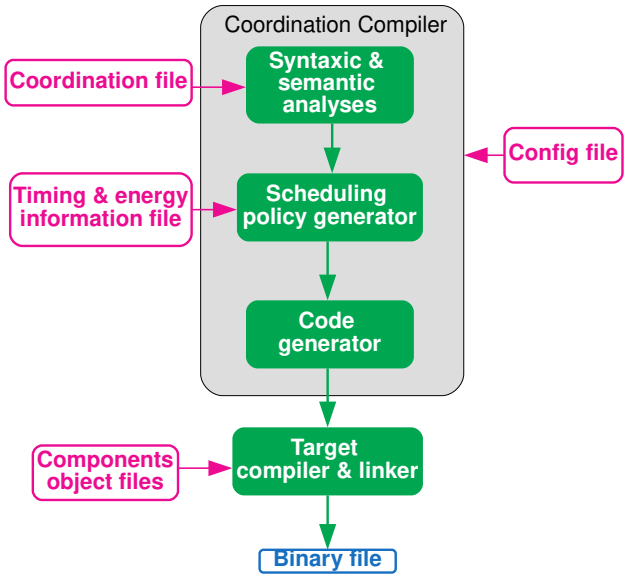


Figure 1: Coordination workflow

[1] https://www.uml.org

Figure 1 summarizes the workflow to build a coordinated application. The three main inputs of the workflow are:

(1) a *coordination file*: provided by the end-user to describe the application structure as well as non-functional properties, see details in Section 4,
(2) *timing and energy information*: provided by timing/energy harvesting tools such as AbsInt aiT [9],
(3) *object files*: provided by a C-compiler such as WCC [7], they contain compiled C-code for each component.

The fourth input file includes practical configurations i.e. target hardware description and security-level mission specifications, compiler passes to apply, ...

First the parser evaluates the *coordination file* and provides a graph representation of the application. This representation is then analysed and transformed according to the enabled compiler passes, e.g. deadlock-free check, security requirements. From the graph and timing & energy information, the schedule policy generator computes different schedules according to hardware specification and security levels configuration when an *off-line* schedule is asked, or performs schedulability analyses in case of *on-line* requirements. Generated schedules are passed to the code generation, e.g. dispatching schedule tables if off-line schedule applied. Finally, schedules and compiled object-files of the whole application are compiled and linked with the target toolchain to generate the final binary executable which can then be downloaded to the platform.

4 TEAMPLAY COORDINATION LANGUAGE

Our coordination language focuses on the design of arbitrary synchronous data-flow-oriented applications. It describes the graph structure by modelling the dependencies between components. Such applications are represented by a graph where vertices are component (a.k.a actors, tasks), while edges correspond to dependencies between components. A dependency defines a data exchange between a source and a sink through a FIFO channel, such data is often called token. A token can have different types according to the application, from primitive types to more evolve structures.

Similarly to periodic task models [16], a data-flow graph instance is called an iteration and a job is a task instance inside an iteration. Then, the DAG may iteratively executes until the end of time (or platform is shutdown). Hence, jobs execution order follows aforementioned constraint, job i finished before job i+1. But, the iteration j +1 can start before the completion of iteration j as long as jobs dependencies are satisfied. This allows to exploit job parallelism, i.e. pipelining [22].

Figure 2 presents the grammar written in pseudo-Xtext language which is given to ANTLR[2] in order to automatically create a lexer and a parser. Following is the description of each rule.

Rule *Application*, line 1, describes the root element of our application. It is composed by an identifier, a deadline and a period. These timing information refer to an iteration of the graph.

[2]https://www.antlr.org

```
1  Application: 'app' ID '{'
2          'deadline' TIME
3          'period' TIME
4          'datatypes' '{' (Datatype)* '}'
5          'components' '{' Component+ '}'
6          'edges' '{' Edge* '}'
7  '}';
8  Datatype: '(' ID ',' STRING
9      (',' DEFAULT (',' INT)?)? ')' ;
10 Component: ID '{'
11         ('inputs' ':' '[' (Connector)* ']')?
12         ('outputs' ':' '[' (Connector)* ']')?
13         Version*
14 '}';
15 Connector: '(' ID ',' INT ',' Datatype ')';
16 Version: 'version' ID '{'
17         ('deadline' INT )?
18         ('period' INT)?
19         ('targetArch' STRING)*
20         ('security' '[' INT ',' INT ']')?
21 '}';
22 Edge: (SingleEdge | DuplicateEdge | DataOrEdge |
       SchedOrEdge | EnvOrEdge);
23 SingleEdge: CompRef '->' CompRef;
24 DuplicateEdge:
25     CompRef '->' CompRef  ( '&' CompRef )+;
26 DataOrEdge: 'DATA' Component ('/' Version)? ':('
27     '.' Connector '->' ComponentRef
28   ('|'  '.' Connector '->' ComponentRef)+ ')';
29 SchedOrEdge:
30     'SCHED' ComponentRef '->' ComponentRef ('|'
           ComponentRef)+;
31 EnvOrEdge:
32         'ENV' ComponentRef '->'
33         '(' STRING '.' ComponentRef
34         ('|' STRING '.' ComponentRef)+ ')';
35 CompRef:
36      Component ('/' Version)? '.' Connector ;
```

Figure 2: Pseudo-Xtext grammar for our coordination language

Rules *Datatype* (line 8), *Component* (line 10) and *Edge* (lines 22-31) are children of an application. They respectively correspond application-wide used data types, components and dependencies between components, more details are following in respective Sections 4.3 4.1 and 4.2 for components and edges.

4.1 Component description

The coordination grammar, Figure 2 line 10, defines a component (as known as task or actor) by two set of connectors, for respectively input and output edges, and a set of versions with a minimum of 1 version.

Connectors represent the interface of the components. What do they consume and require to proceed with input connectors, and what will they produce with output connectors. Each connector includes a name, an amount of tokens, and a data type identifier. They are useful to perform a type checking analyses and guaranty that tokens produced along an edge are of the same type (or compatible one according

to casting capabilities of the parser) as tokens consumed on that same edge.

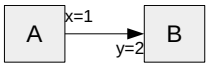
If no input connectors (resp. output connectors) are given, then the component is a source (resp. sink) component.

To enable different scheduling strategies depending on time, energy and security, each component can be characterized by multiple versions. Each version can have it own timing, energy and security information at the coordination specification level as well as extracted from the *Timing & energy information file* provided to the coordination toolchain. In order to switch between different security level, Figure 2 line 20 allows to specify security range levels at which this version can be used. In addition, our coordination specification allows to specify a specific architecture on which the version can run, this will enable further scheduling strategies to account for heterogeneous processor.

It must be noted that when no version is given by the user, a default version is created by the compiler with widest range for the security value.

4.2 Dependencies representation

With dataflow-oriented application, tasks exchange data known as tokens. The token type depends on the application, e.g. a surveillance camera system would include tokens of type *image* while wireless devices would use integers. Dependencies therefore represent the flow of tokens in the graph. Following are constructions we allow with our representation. Each construction are presented with both a graphical sketch, and a textual representation with respect to the grammar defined in Figure 2. For the graphical representation, components are grey boxes while dependencies are arrows.



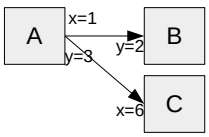
(a) Graphical sketch

```
A.x -> B.y
```

(b) Textual declaration

Figure 3: Simple edge

Figure 3 presents a simple edge between a source component A producing 1 token and a sink component B consuming 2 tokens, defined by Figure 2 line 23.



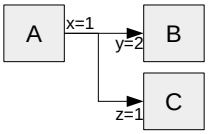
(a) Graphical sketch

```
A.x -> B.y
A.y -> C.x
```

(b) Textual declaration

Figure 4: Multiple sinks edge

Figure 4 is an extrapolation of the previous dependency construction where source A produces 4 tokens and the first one goes to component B while the 3 others go to component C. Identically to [15] we impose the *top-down* ordering to determine the order of produced/consumed tokens. This allows to use scheduling policy from [18] where the order of produced/consumed tokens might have an impact on the scheduler decision. Our coordination language also allows to have more than one source for a component which is enabled by trivially deriving the two previous constructions.



(a) Graphical sketch

```
A.x -> B.y & C.z
```

(b) Textual declaration

Figure 5: Tokens duplication edge

Figure 5 presents a duplicating edge between a source component A producing 1 token and two sink components B and C consuming respectively 2 tokens and 1 token, as defined by Figure 2 line 25. The *duplication* dependency imposes to duplicate the token produced by component A on parts of the edge going to the two sinks. Hence, components B and C will work on the same copy of the same token.

(a) Graphical sketch

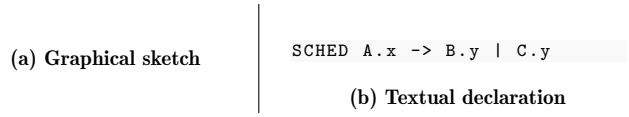
```
DATA A -> (
    .x -> B.x |
    .y -> C.x
)
```

(b) Textual declaration

Figure 6: Path selection edge depending on data

Figure 6 allows conditional dependencies driven by the data, as defined by Figure 2 line 26. In this case, component B and component C are dependent of component A but only one is allowed to actually execute depending on which output connector of A throw data out. If at the end of the execution of A, data are present on the connector x then component B is fired, or if data are present on the connector y then component C is fired. If no data are present on any of the two connectors x and y at the end of the execution of A, then neither B nor C are fired. This enables a powerful mechanism that can be used in control program where the presence of a stimuli enable part of the application. For example, in a face recognition system, a first component could detect if there are people on the image, if there is nobody there is no need to transmit the image to the face recognition sub-algorithms.

Figure **??** allows conditional dependencies driven by the scheduler, as defined by Figure 2 line 29. Similarly to the previous case, component B and component C are dependent



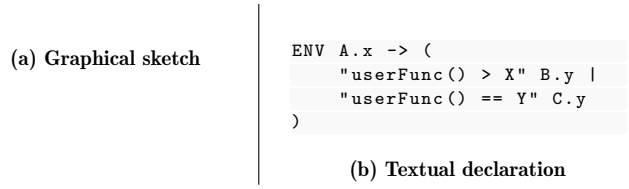
(a) Graphical sketch

```
SCHED A.x -> B.y | C.y
```

(b) Textual declaration

Figure 7: Path selection edge depending on scheduler decision

of component A but only one is allowed to actually execute depending on a decision from the scheduler. For example, if the time budget requested by component B is lower that component C request, the scheduler can chose to fire component B instead of C. Such decision could be motivated to avoid a deadline miss with a bit of loss of accuracy.



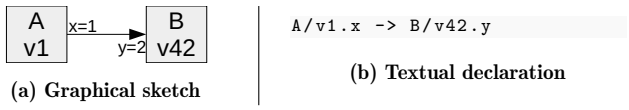
(a) Graphical sketch

```
ENV A.x -> (
    "userFunc() > X" B.y |
    "userFunc() == Y" C.y
)
```

(b) Textual declaration

Figure 8: Path selection edge depending on scheduler decision

Figure **??** allows conditional dependencies driven by the user, as defined by Figure 2 line 31. In this case, component B, C and D are dependent of component A and this dependency is preceded by a condition. If this condition returns true, then the corresponding component is fired with a copy of the data. All simultaneouly fired component will work on the same input data. If no condition returns *true*, then no component are actually fired. For example, if the *userFunc* is *get_battery*, the user might choose to fire all following component when the battery is full, and some of them when the battery is half empty, or none if the battery reach a specific level.



(a) Graphical sketch

```
A/v1.x -> B/v42.y
```

(b) Textual declaration

Figure 9: Simple edge with specific versions

Finally, our coordination language allows to constrain dependencies with component versions as in Figure 9 where a simple edge links the version x of component A and the version y of component B. This allows, for example, to guarantee that for component versions with overlapping security level, the two versions with the highest minimum security level are defined as dependent.

4.3 Other rules

Rule *Datatype*, line 8, refers to data types exchanged by components. First is an identifier used to refer to this particular data type inside the coordination file. Followed two by a string representing its implementation in user code with a default value, and finally a size in bits allowing further analyses, such as memory usage.

Rule *ComponentRef*, line 35, is used to reference a component connector in an edge and refers to an identifier of this already defined *Component*. To allow dependencies on component version (see Section 4.1 for details on version), a *ComponentRef* can include a version identifier. Then, the last *ID* of rule *ComponentRef* states the connector used to connect the component.

Terminals *ID*, *INT*, *STRING*, *DEFAULT* and *TIME* are not described for space consideration. But they respectively catch an identifier (start with a letter followed with any alpha-numeric characters and some special characters, e.g. '_'), an integer, a string, any string representing a default value, a time duration in e.g. hours, milliseconds, hertz or cycles.

5 COORDINATION COMPILER TOOLCHAIN

5.1 Syntax & semantic validation

parser generation with antlr to C++ parser to create an AST. This parser validates the syntaxe. From the AST, we visit each element to create an IR and validates the semantic rules:

- connector include a defined datatype
- edges connect existing component
- edges connect existing connectors,
- edges connect connectors that are respectively in input list for source component and output for sink one.
- if version number are provided at edge level, check that edges connect existing version for considered component
- version are targeting available architecture from the configuration

Available higher level static validation:

- Type check: source and sink connector refer to the same data type
- Cycle check: detect if a cycle is present
- Deadlock check: detect if consumption/production is not stable

Graph transformation:

- Transitive closure computation
- Qvector computation
- Qvector flattening
- Security level flattening
- Conditional edge flattening

5.2 Type checking

$$\forall src, sink \in E, \quad typesrc = typesink \tag{1}$$

5.3 Deadlock checking

$$\forall v \in V, prodv = \underset{p \in succV}{} consp \tag{2}$$

$$\forall v \in V, consv = \underset{p \in predV}{} prodp \tag{3}$$

5.4 Scheduling policy generator

5.4.1 Towards scheduling. Prior to scheduling a graph unfolding is required to compute the repetition vector, i.e. number of times each actor needs to be fired, in order to consume all produced tokens as well as to produce as much as consumed tokens. A scheduler then generates a schedule of the graph which will be repeatedly executed on the board. Hence, the scheduler acts on one iteration of the graph [23].

6 CONCLUSION

In this paper we presented Teamplay Coordination Language, a simple DSL to coordinate a critical streaming applications while enforcing safety and security. We presented a grammar to describe the language and applied it to implement a Drone use-case. It is also worth mentioning that we implemented our grammar using Xtext-Eclipse[3] to provide a graphical editor. Hence, Xtext uses ANTLR inwards which allows us to generate a lexer/parser for any scheduler software a end-user might chose.

As future work, we intend to fill all boxes from the coordination workflow including an off-line and an on-line scheduler. Our future scheduling algorithms will guarantee an efficient, safe, secure and energy optimized application that we intend to effectively run on a drone after a code generation phase.

ACKNOWLEDGEMENT

This work is funded by the European Union Horizon2020 research and innovation programme under grant agreement No. 779882 (TeamPlay).

REFERENCES

[1] Mouaiad Alras, Paul Caspi, Alain Girault, and Pascal Raymond. 2009. Model-based design of embedded control systems by means of a synchronous intermediate model. In *2009 International Conference on Embedded Software and Systems*. IEEE, 3–10.

[2] Saoussen Anssi, Sara Tucci-Piergiovanni, Stefan Kuntz, Sébastien Gérard, and François Terrier. 2011. Enabling scheduling analysis for AUTOSAR systems. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 152–159.

[3] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[4] Andrea Bondavalli, Lorenzo Strigini, and Luca Simoncini. 1992. Dataflow-like languages for real-time systems: issues of computational models and notation. In *[1992] Proceedings 11th Symposium on Reliable Distributed Systems*. IEEE, 214–221.

[5] Giovanni Ciatto, Stefano Mariani, Maxime Louvel, Andrea Omicini, and Franco Zambonelli. 2018. Twenty years of coordination technologies: State-of-the-art and perspectives. In *International Conference on Coordination Languages and Models*. Springer, 51–80.

[6] Albert Cohen, Valentin Perrelle, Dumitru Potop-Butucaru, Marc Pouzet, Elie Soubiran, and Zhen Zhang. 2016. Hard Real Time and Mixed Time Criticality on Off-The-Shelf Embedded Multi-Cores. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*.

[7] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. 2006. Design of a wcet-aware c compiler. In *2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*. IEEE, 121–126.

[8] Peter H Feiler, David P Gluch, and John J Hudak. 2006. *The architecture analysis & design language (AADL): An introduction*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

[9] Christian Ferdinand and Reinhold Heckmann. 2004. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*. Springer, 377–383.

[10] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. 2009. AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, Vol. 62. 5.

[11] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. 2010. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming* 38, 1 (2010), 38–67.

[12] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[13] Kevin Hammond and Greg Michaelson. 2003. Hume: a domain-specific language for real-time embedded systems. In *International Conference on Generative Programming and Component Engineering*. Springer, 37–56.

[14] Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra S Bhattacharyya. 2004. DIF: An interchange format for dataflow-based design tools. In *International Workshop on Embedded Computer Systems*. Springer, 423–432.

[15] Edward Ashford Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers* 100, 1 (1987), 24–35.

[16] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[17] Riad Nassiffe, Eduardo Camponogara, and George Lima. 2011. A Model for Reconfiguration of Multi-Modal Real-Time Systems under Energy Constraints. In *2011 Brazilian Symposium on Computing System Engineering*. IEEE, 127–132.

[18] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. 2019. Hiding communication delays in contention-free execution for SPM-based multi-core architectures. In *31th Euromicro Conference on Real-Time Systems (ECRTS19)*.

[19] Cosmin Rusu, Rami Melhem, and Daniel Moss'e. 2005. Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing* 1, 2 (2005), 271–283.

[20] Bran Selic and Sébastien Gérard. 2013. *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*. Elsevier.

[21] Abhishek Singh, Pontus Ekberg, and Sanjoy Baruah. 2017. Applying real-time scheduling theory to the synchronous data flow model of computation. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[22] Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. 2014. Many-core scheduling of data parallel applications using SMT solvers. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 615–622.

[23] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 179–196.

[3] https://www.eclipse.org/Xtext/