Towards Energy-, Time- and Security-Aware Multi-core Coordination

Julius Roeder[1], Benjamin Rouxel[1], Sebastian Altmeyer[2], and Clemens Grelck[1(✉)]

[1] University of Amsterdam, Science Park 904, 1098XH Amsterdam, Netherlands
{j.roeder,b.rouxel,c.grelck}@uva.nl
[2] University of Augsburg, Universitätsstr. 2, 86159 Augsburg, Germany
altmeyer@informatik.uni-augsburg.de

Abstract. Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches. Yet, we are not aware of any adoption of the principle of coordination in the broad domain of cyber-physical systems, where non-functional properties, such as execution/response time, energy consumption and security are as crucial as functional correctness.

We propose a coordination approach, including a functional coordination language and its associated tool flow, that considers time, energy and security as first-class citizens in application design and development. We primarily target cyber-physical systems running on off-the-shelf heterogeneous multi-core platforms. We illustrate our approach by means of a real-world use case, an unmanned aerial vehicle for autonomous reconnaissance mission, which we develop in close collaboration with industry.

Keywords: Cyber-physical systems · Non-functional properties · Real-time · Energy · Security

1 Introduction

Cyber-physical systems (CPS) deeply intertwine software with physical components, such as sensors and actuators that impact the physical world. Broadly speaking the software controls the actuators of a physical system based on input from the sensors and specified policies. Our world is full of cyber-physical systems, ranging from washing machines to airplanes. Designing secure, safe and correct cyber-physical systems requires a tremendous amount of verification, validation and certification.

A common characteristic of cyber-physical systems is that non-functional properties of the software, such as time, energy and security, are as important for correct behaviour as purely functional correctness. Actuators must react on

This work is supported and partly funded by the European Union Horizon-2020 research and innovation programme under grant agreement No. 779882 (TeamPlay).

Published by Springer Nature Switzerland AG 2020
S. Bliudze and L. Bocchi (Eds.): COORDINATION 2020, LNCS 12134, pp. 57–74, 2020.
https://doi.org/10.1007/978-3-030-50029-0_4

sensor input within a certain time limit, or the reaction might in the worst case become useless. In addition to general environmental concerns, energy consumption of computing devices becomes crucial in battery-powered cyber-physical systems. Security concerns are paramount in many cyber-physical systems due to their potentially harmful impact on the real world. However, more security typically requires more computing effort. More computing effort takes more time and consumes more energy. Thus, time, energy and security are connected in the triangle of non-functional properties.

The multi-core revolution has meanwhile also reached the domain of cyber-physical systems. A typical example is the ARM big. LITTLE CPU architecture that features four energy-efficient Cortex A7 cores and four energy-hungry, but computationally more powerful, Cortex A15 cores. Many platforms complement this heterogeneous CPU architecture with an on-chip GPU. Such architectures create previously unknown degrees of freedom regarding the internal organisation of an application: what to compute where and when. This induces a global optimisation problem, for instance minimising energy consumption, under budget constraints, for instance in terms of time and security.

We propose the domain-specific functional coordination language TeamPlay and the associated tool chain that consider the aforementioned non-functional properties as first-class citizens in the application design and development process. Our tool chain compiles coordination code to a final executable linked with separately compiled component implementations. We combine a range of analysis and scheduling techniques for the user to choose from like in a tool box. The generated code either implements a static (offline) schedule or a dynamic (online) schedule. With static/offline scheduling all placements and activation times are pre-computed; with dynamic/online scheduling certain decisions are postponed until runtime.

Both options are driven by application-specific global objectives. The most common objective is to minimise energy consumption while meeting both time and security constraints. A variation of the theme would be to maximise security while meeting time and energy constraints. Less popular, but possible in principle, would be the third combination: minimising time under energy and security constraints.

Both offline and online scheduling share the concept of making conscious and application-specific decisions as to what compute where and when. Our work distinguishes itself from, say, operating system level scheduling by the clear insight into both the inner workings of an application and into the available computing resources.

The specific contribution of this paper lies in the design of the energy-, time- and security-aware coordination language and the overall approach. Due to space limitations we can only sketch out the various elements of our tool chain and must refer the interested reader to future publications to some degree.

The remainder of the paper is organised as follows: In Sect. 2 we explain our view on coordination followed by a detailed account of our (domain-specific) coordination language in Sect. 3. In Sect. 4 we illustrate our approach by means of

a real-world use-case, and in Sect. 5 we sketch out our tool chain implementation. We discuss related work in Sect. 6 and conclude in Sect. 7.

2 Coordination Model

The term *coordination* goes back to the seminal work of Gelernter and Carriero [13] and their coordination language Linda. Coordination languages can be classified as either *endogenous* or *exogenous* [5]. Endogenous approaches provide coordination primitives within application code. The original work on Linda falls into this category. Exogenous approaches fully separate the concerns of coordination programming and application programming

We pursue an exogenous approach and foster 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.

2.1 Components

We illustrate our component model in Fig. 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.

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 Fig. 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 components and sink components form the quintessential interfaces between the physical world and the cyber-world representing sensors and actuators in the broadest sense. We adopt the firing rule of Petri-nets, i.e. a component is activated as soon as data (tokens) are available on each of its input ports.

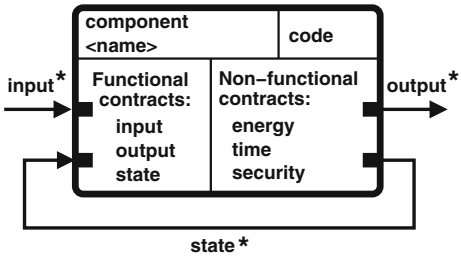


Fig. 1. Illustration of component model

Technically, a component implementation is a function adhering to the C calling and linking conventions [21]. Name and signature of this function can be derived from the component specification in a defined way. This function may call other functions using the regular C calling convention. The execution of a component (function), including execution of all subsidiary regular functions, must be free of side-effects. In other words, input tokens must map to output tokens in a purely functional way. Exceptions are source and sink components that are supposed to control sensors and actuators, respectively.

2.2 Stateful Components

Our components are conceptually stateless. However, some sort of state is very common in cyber-physical systems. We model such state in a functionally transparent way as illustrated in Fig. 1, namely by so-called state ports that are short-circuited from output to input. In analogy to input ports and output ports, a component may well have no state ports. We call such a component a (practically) *stateless* component.

Our approach to state is 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 [28] in Haskell or uniqueness types [1] in Clean. Making state explicit is key to properly deal 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 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 ETS-aware Components

We are particularly interested in the non-functional properties of code execution. Hence, any component not only comes with functional contracts, as sketched out before, but additionally with non-functional contracts concerning energy, time and security (and potentially more in the future).

These three non-functional properties are inherently different in nature. Execution time and energy consumption can be measured, depend on a concrete execution machinery and vary between different hardware scenarios. In contrast, security, more precisely algorithmic security, depends on the concrete implementation of a component, for example using different levels of encryption, etc. However, different security levels almost inevitably incur different computational demands and, thus, are likely to expose different runtime behaviour in terms of time and energy consumption as well.

Knowledge about non-functional properties of components is at the heart of our approach. It is this information that drives our scheduling and mapping decisions to solve the given optimisation problem (e.g. minimising energy consumption) under constraints (e.g. execution deadlines and minimum security requirements).

2.4 Multi-version Components

As illustrated in Fig. 2, a component may have multiple versions with identical functional behaviour, but with different implementations and, thus, different energy, time and (possibly) security contracts. Multi-version components add another degree of freedom to the scheduling and mapping problem that we address: selecting the best fitting variant of a component under given optimisation objectives and constraints.

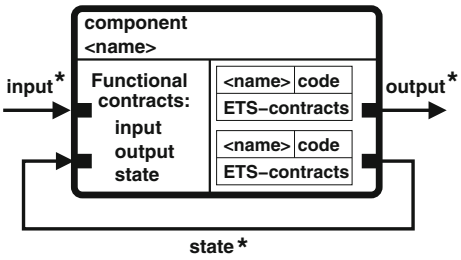


Fig. 2. Multi-version component with individual energy, time and security contracts

Take as an example our reconnaissance drone use case, that we will explore in more detail in Sect. 4. A drone could adapt its security protocol for communication with the base station in accordance with changing mission state: low security while taking off or landing, medium security while navigating to/from mission area, high security during mission. Continuous adaptation of security levels results in less computing and, thus, in energy savings that could be exploited for longer flight times.

Our solution is to provide different versions of the same component (similar to [24]) and to select the best version regarding mission state and objectives based on the scheduling strategy. For the time being, we only support off-line version selection, but scenarios with online version control, as sketched out above, are on our agenda.

2.5 Component Interplay

Components are connected via FIFO channels to exchange data, as illustrated in Fig. 3. Depending on application requirements, components may start computing at statically determined time slots (when all input data is guaranteed to be present) or may be activated dynamically by the presence of all required input data. Components may produce output data on all or on selected output ports.

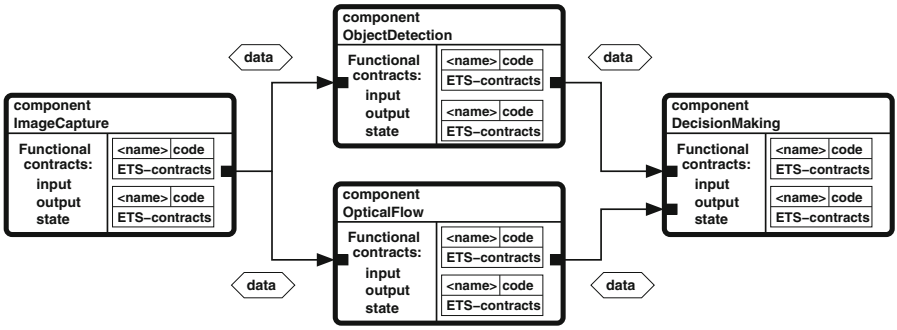


Fig. 3. Illustration of data-driven component interplay via FIFO channels

3 Coordination Language

Our coordination language focuses on the design of arbitrary synchronous data-flow-oriented applications. It describes a graph structure where vertices are components (*actors*, *tasks*) while edges represent dependencies between components. A dependency/edge defines a data exchange between a source and a sink through a FIFO channel. Such a data item, called *token*, can have different types, from primitive types to more elaborate structures.

Similar to periodic task models [18] a data-flow graph instance is called an *iteration*. A *job* is a component instance inside an iteration. As usual we require graphs to be acyclic (i.e. DAGs). The DAG iteratively executes until the end of time (or platform shutdown). The job execution order follows the (aforementioned) constraint that job i must finish before job i+1. However, iteration j+1 can start before the completion of iteration j as long as dependencies are satisfied. This allows us to exploit job parallelism, e.g. pipelining [26].

Figure 4 presents the grammar of our coordination language written in pseudo-Xtext style. In the following we describe each production rule in more detail.

3.1 Program Header

Rule *Application* (Fig. 4, line 1) describes the root element of our application. It is composed of the application name, a deadline and a period. All times refer to one iteration of the graph; they can be given in, for instance, hours, milliseconds, hertz or clock cycles.

Rule *Datatype* (Fig. 4, line 9) declares the data types used throughout the coordination specification. One data type declaration consists of the type's name, followed by a string representation of its implementation in user code (i.e. the actual C type like *int* or *struct FooBar*) and, optionally, by the size in bytes. The size information allows for further analysis, e.g. regarding memory footprint. The string representation of the type's implementation is needed for code generation.

3.2 (Multi-version) Components

A component in our coordination DSL (Fig. 4, line 11) consists of a unique name, three sets of *ports* and a number of additional properties. Multi-version components (see Sect. 2.4) feature a number of *versions*, where each version consists of a unique name and the additional properties, now specific to each version. The simplified syntax for single-version components is motivated by their abundance in practice.

Ports represent the interface of a component. The *inports* specify the data items (or tokens) consumed by a component while the *outports* specify the data items (or tokens) that a component (potentially) produces. The third set of ports, introduced by the keyword `state`, are both input ports and output ports at the same time, where the output ports are short-circuited to the corresponding input ports, as explained in Sect. 2.2.

```
Application: 'app' ID '{'
               'deadline' TIME
               'period' TIME
               'datatypes' '{'  Datatype+ '}'
               'components' '{' Component+ '}'
               'edges' '{' Edge+ '}'
             '}';

Datatype: '(' ID ',' STRING (',' UINT)?;

Component: ID '{'
             ('inports' ':' '[' Port* ']')?
             ('outports' ':' '[' Port* ']')?
             ('state' ':' '[' Port* ']')?
             (Properties | Version+)
           '}';

Port: '(' ID (',' INT)? ',' DatatypeRef ')';

Properties: ('deadline' TIME)?
            ('period' TIME)?
            ('arch' STRING)*
            ('security' UINT)?

Version: 'version' ID '{' Properties '}';

Edge:  SimpleEdge | BroadcastEdge | DataOrEdge
       | SchedOrEdge | EnvOrEdge;

SimpleEdge: OutPort '->' InPort;

BroadcastEdge: OutPort '->' InPort ('&' InPort)+;

DataOrEdge: OutPort '->' InPort
            ('|' OutPort '->' InPort)+;

SchedOrEdge: OutPort '->' InPort ('|' InPort)+;

EnvOrEdge: OutPort '->' InPort 'where' STRING
           ('|' InPort 'where' STRING)+;

OutPort: CompRef ('.' OutPortRef)?

InPort: CompRef ('.' InPortRef)?
```

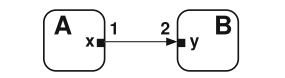
Fig. 4. Coordination language pseudo-Xtext grammar

A port specification includes a unique name, the token multiplicity and a data type identifier. Token multiplicities are optional and default to one. They allow components to consume a fixed number of tokens on an input port at once, to produce a fixed number of tokens on an output port at once or to keep multiple tokens of the same type as internal (pseudo) state. The firing rule for components is amended accordingly and requires (at least) the right number of tokens on each input port. Typing ports is useful to perform static type checking and to guarantee that tokens produced by one component are expected by a subsequent component connected by an edge. To start with we require type equality, but we intend to introduce some form of subtyping at a later stage,

Our three non-functional properties behave differently. While the security level is an algorithmic property of a component (version), energy and time critically depend on the execution platform. Therefore, we encode the (application-specific) security (level) as an integer number in the code, but not energy and time information. We keep the coordination code platform-independent and obtain energy and time information from a separate data base (to be elaborated on in Sect. 5).

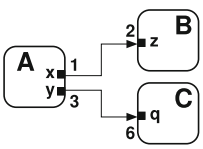
3.3 Dependencies

Dependencies (or *edges*) represent the flow of tokens in the graph. Their specification is crucial for the overall expressiveness of the coordination language. We support a number of constructions to connect output ports to input ports (Fig. 4, line 27). In the following we illustrate each such construction with both a graphical sketch and the corresponding textual representation.



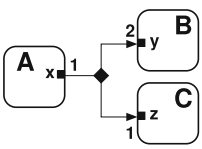

```
A.x -> B.y
```

(a) Simple edge




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

(b) Multiple edges




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

(c) Broadcast edge

Fig. 5. Various edge construction examples

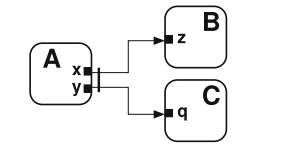
Figure 5a presents a simple edge between the output port `x` of component *A* and the input port `y` of component `B`. In our example the output port has a multiplicity of one token while the input port has a multiplicity of two tokens. We show token multiplicities in Fig. 5a for illustration only. In the coordination program token multiplicities are part of the port specification (Fig. 4, line 18),

not the edge specification (line 30). Coming back to the example of Fig. 5a, component `A` produces one output token per activation, but component `B` only becomes activated once (at least) two tokens are available on its input port. Thus, component `A` must fire twice before component `B` becomes activated.

Figure 5b shows an extension of the previous dependency construction where component A produces a total of four tokens: one on port `x` and three on port `y`. Component `B` expects two tokens on input port `z` while sink component `C` expects a total of six tokens on input port `q`. These examples can be extended to fairly complex dependency graphs.

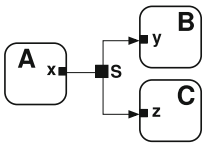
Figure 5c shows a so-called *broadcast edge* between a source component A producing one token and two sink components B and C consuming two tokens and one token, respectively (corresponding to Fig. 4, line 32). This form of component dependency duplicates the token produced on the output port of the source component and sends it to the corresponding input ports of all sink components. Token multiplicities work in the very same way as before: any tokens produced by a source component go to each sink component, but sink components only become activated as soon as the necessary number of tokens accumulate on their input ports. A broadcast edge does not copy the data associated with a token, only the token itself. Hence, components B and C in the above example will operate on the same data and, thus, are restricted to read access.

Components with a single input port or a single output port are very common. In these cases port names in edge specifications can be omitted, as they are not needed for disambiguation.



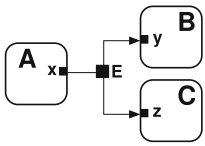

```
  A.x -> B.z
| A.y -> C.q
```

(a) Data-dependent




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

(b) Scheduler-dependent




```
A.x -> B.y
 where "Cexpr"
   | C.z
 where "Cexpr"
```

(c) Environment-dependent

Fig. 6. Data-, scheduler- and environment-dependent edges

Figure 6a illustrates a data-driven conditional dependency (corresponding to Fig. 4, line 34). In this case, component B and component C are dependent on component A, but only one is allowed to actually execute depending on which output port component A makes use of. If at the end of the execution of A a token is present on port x then component B is fired; if a token is present on

port y then component C is fired. If no tokens are present on either port 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 programs where the presence of a stimulus enables part of the application. For example, in a face recognition system an initial component in a processing pipeline could detect if there are any person on an image. If so, the image is forwarded to the subsequent face recognition sub-algorithms; otherwise, it is discarded.

Figure 6b allows conditional dependencies driven by the scheduler (corresponding to Fig. 4, line 37. Similar to the previous case, component B and component C depend on component A, but only one is allowed to actually execute depending on a decision by the scheduler. For example, if the time budget requested by component B is lower than that requested by component C, the scheduler can choose to fire component B instead of C. Such a decision could be motivated by the need to avoid a deadline miss at the expense of some loss of accuracy.

Figure 6c allows conditional dependencies driven by the user (corresponding to Fig. 4, line 39). In this case components B and C again depend on component A, but this time the dependency is guarded by a condition. If the condition evaluates to true then the token is sent to the corresponding route. There is no particular evaluation order for conditions, and tokens are simultaneously sent to all sink components whose guards evaluate to true. Like in the case of the broadcast edge all fired components receive the very same input data. If no guard returns *true*, the token is discarded.

The guards come in the form of strings as inline C code. The code generator will literally place this code into condition positions in the generated code. The user is responsible for the syntactic and semantic correctness of these C code snippets. This is not ideal with respect to static validation of coordination code, but similar to, for instance, the if-clause in OpenMP. On the positive side, this feature ensures maximum flexibility in application design without the need for a fully-fledged C compiler frontend, which would be far beyond our means.

For example, the *Cexpr* could contain a call to a function `get_battery` that enquires about the battery charge status. The coordination program may choose to fire all subsequent components as long as the battery is well charged, but only some as the battery power drains. Or, it may fire different components altogether, changing the system behaviour under different battery conditions.

4 Example Use Case Reconnaissance Drone

We illustrate our coordination approach by means of a use case that we develop jointly with our project partners University of Southern Denmark and SkyWatch A/S [25]. Fixed-wing drones can stay several hours in the air, making them ideal equipment for surveillance and reconnaissance missions. In addition to the flight control system keeping the drone up in the air, our drone is equipped with a camera and a payload computing system. Since fixed-wing drones are highly energy-efficient, computing on the payload system does have a noticeable

impact on overall energy consumption and, thus, on mission length. We illustrate our coordination approach in Fig. 7; the corresponding coordination code is shown in Fig. 8. We re-use the original application building blocks developed and used by Sky-Watch A/S.

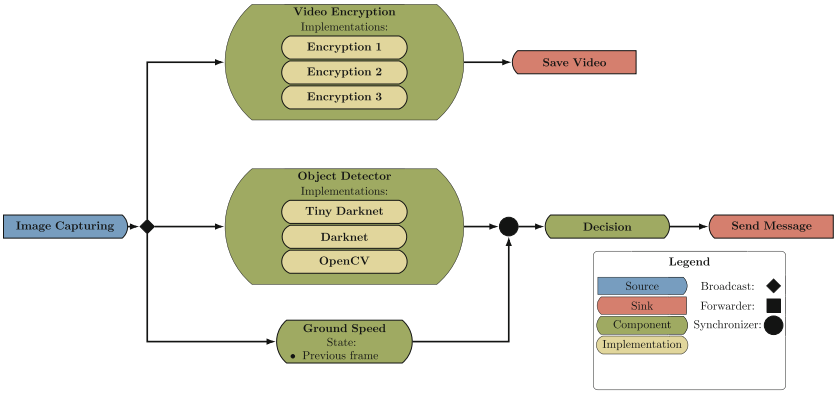


Fig. 7. Reconnaissance drone use case coordination model

The drone's camera system takes pictures in predefined intervals. Our *ImageCapture* component represents this interface to the physical world. Global period and deadline specifications correspond to the capture frequency of the camera. The non-standard data types declared in the `datatypes` section of the coordination program are adopted from the original application code. We use the C types in string form for code generation and require that corresponding C type definitions are made available to the backend C compiler via header files.

Images are broadcast to three subsequent components. The *VideoEncryption* component encrypts the images of the video stream and forwards the encrypted images to follow-up component *SaveVideo* that stores the video in persistent memory for post-mission analysis and archival. Video encryption comes with three different security levels. For simplicity we just call them *Encryption1*, *Encryption2* and *Encryption3*. Different encryption levels could be used, for instance, for different mission environments, from friendly to hostile.

The drone also performs on-board analyses of the images taken. These are represented by our components *ObjectDetector* and *GroundSpeed*. Object detection can choose between three algorithms with different accuracy, time and energy properties: Darknet[1], Tiny Darknet[2], OpenCV. The ground speed estimator works by comparing two subsequent images from the video stream. This is the only stateful component in our model. The results of object detection and ground speed estimation are synchronised and fed into the follow-up component

[1] https://pjreddie.com/darknet/.
[2] https://pjreddie.com/darknet/tiny-darknet/.

```
app drone {
  deadline 50Hz
  period 50Hz
  datatypes {
    (frame, "jpegFrame*")
    (num, "uint32_t")
    (enc, "encryptedData*")
    (string, "char*")
  }
  components {
    ImageCapture { outports [ (out, frame) ] }
    Encryption {
      inports [ (in, frame) ]
      outports [ (out, enc) ]
      version Encryption1 {security 4}
      version Encryption2 {security 6}
      version Encryption3 {security 9}
    }
    ObjectDetector {
      inports [ (in, frame) ]
      outports [ (obj, num) (frame, frame) ]
      version TinyDarknet {arch "cpu/big"}
      version Darknet {arch "cpugpu"}
      version OpenCV {arch "cpugpu"}
    }
    GroundSpeed {
      inports [ (in, frame) ]
      outports [ (speed, num) ]
      state [ (s, frame) ]
    }
    Decision {
      inports [(obj, num) (frame, frame) (speed, num)]
      outports [ (msg, string) ]
    }
    SaveVideo { inports [ (in, enc) ] }
    SendMessage {  inports [ (msg, string) ] }
  } edges {
    ImageCapture -> Encryption & ObjectDetector & GroundSpeed
    Encryption -> SaveVideo
    ObjectDetector.obj -> Decision.obj
    ObjectDetector.frame -> Decision.frame
    GroundSpeed -> Decision.speed
    Decision -> SendMessage
} }
```

Fig. 8. Coordination program for drone use case

Decision that combines all information and decides whether or not to notify the base station about a potentially relevant object detected.

Transmission of the message is modelled by the sink component *SendMessage*, where the action returns to the physical world. To implement dynamic adaptation to dynamically changing mission phases, as sketched out in Sect. 2.4, we would need multiple versions of this component with different security levels as well. However, we leave dynamic adaptation to future work for now.

As Fig. 8 demonstrates, our coordination language allows users to specify non-trivial cyber-physical applications in a concise and comprehensible way. The entire wiring of components only takes a few lines of code. Our approach facilitates playing with implementation variations and, thus, enables system engineers to explore the consequences of design choices on non-functional properties at an early stage. Note that all ports in our example have a token multiplicity of one, and we consistently make use of default ports where components only feature a single input port or a single output port.

5 Coordination Tool Chain

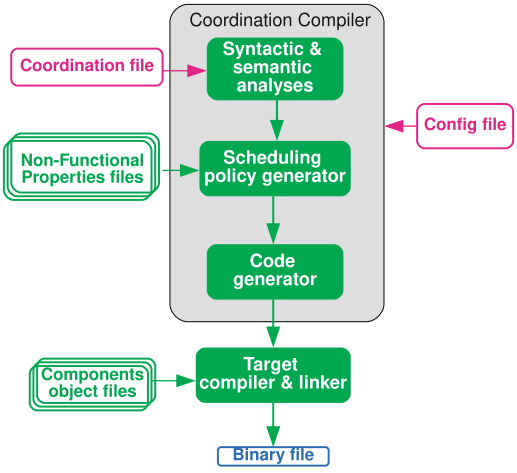


Fig. 9. Coordination workflow

Figure 9 illustrates our coordination tool chain; its four main inputs are:

1. the *coordination program*, as described in Sect. 3;
2. *timing and energy information per component*: provided by timing/energy harvesting tools such as AbsInt aiT [12] for a specific architecture;
3. *object files*: provided by a C-compiler such as WCC [10], containing binary code for each component (version).
4. a *config file* with configuration information, e.g. target hardware, security-level mission specifications, compiler passes to apply, etc.

For syntactic and semantic analysis, we use the parser generator ANTLR to derive a C++ parser from an Xtext grammar specification that is very similar to the one shown in Fig. 4. This implementation choice provides us with a graphical editor plug-in for the Eclipse IDE for free[3]. The resulting parser validates the syntax and creates an abstract syntax tree (AST), on which we validate a number of semantic rules:

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

– ports refer to well defined data types;
– edges connect existing components;
– edges connect output ports with input ports;
– versions target available architectures.

Type checking entails validating that output and input ports connected by an edge use equivalent types. Using standard graph terminology this can be formalised as

$$\forall src, sink \in E : src_{type} = sink_{type} \tag{1}$$

Deadlock checking in our context entails static detection of stable token consumption/production rates. Formally, the number of tokens produced by a component (vertex) must coincide with the sum of tokens expected by all successor components:

$$\forall v \in V : v_{prod} = \sum_{p \in V_{succ}} p_{cons} \tag{2}$$

Likewise, the number of tokens consumed by a component must match the sum of tokens produced by all predecessor components:

$$\forall v \in V : v_{cons} = \sum_{p \in v_{pred}} p_{prod} \tag{3}$$

The second block of our coordination tool chain in Fig. 9 is the *scheduling policy generator*, which depends on configuration parameters provided by the user. In the case of static offline scheduling, the scheduling policy generator generates a schedule table with locations and release times for each component [22,23]. In the case of dynamic online scheduling it performs a schedulability analysis for which we have adapted the techniques of Melani et al. [19] or, alternatively, those of Casini et al. [8].

Offline and online schedulers both have their specific benefits and drawbacks: offline schedulers are easy to implement (e.g. with alarms) and, as all release times are decided a-priori, scheduling overhead is minimal. However, offline schedulers are not work-conserving. Should a component finish quicker than suggested by its worst-case execution time, the corresponding core stays idle until the subsequent release time of some component. In contrast, online schedulers are work-conserving and, thus, more efficient in practice. However, this efficiency comes at the cost of higher runtime overhead and implementation difficulty since we need a mechanism that decides at runtime which component to execute next.

Whether to opt for offline or online scheduling depends on the application scenario at hand. Our tool chain merely facilitates users to make this choice. For offline scheduling we provide both an ILP-based solution [22] and a heuristic for larger use cases, where the solving an ILP proves to be too time-consuming.

Code generation is the final step in our tool flow. For the coordination part of an application, we generate C-code that manages components and their interaction through threads and processes according to the configured scheduling policy, including releasing, synchronisation, and communication of components.

In a final step the generated C-code is compiled by a platform-specific C compiler and linked with the likewise compiled component implementations into an executable binary, ready to be deployed to the platform of choice.

We successfully applied our tool chain to the drone use-case presented in Sect. 4. At the time of writing we are able to generate a static schedule (both ILP- and heuristics-based) that optimises the overall energy consumption while meeting all time and security constraints. Our project partner Sky-Watch A/S successfully tested this code on an actually flying drone. We are still in the process of evaluating the outcome of these experiments compared to the original hand-coded software of Sky-Watch A/S. We envision in the very near future to have our code generator ready to produce dynamically scheduled applications.

6 Related Work

Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches, surveyed in [9]. 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-, time- or security-aware approaches to coordination similar to our approach.

In the area of exogenous coordination languages we mention the work on Reo [4]. 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 labeled directed hypergraph. Reo further has a (or rather many) formal semantics [17]. 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) DSL to create executable energy-, time- and security-aware programs running on concrete machinery.

Another example of an exogenous coordination language is S-Net [14], from which we draw inspiration and experience for our proposed design. However, S-Net merely addresses the functional aspects of coordination programming and has left out any non-functional requirements, not to mention energy, time and security, in particular.

A notable exception in the otherwise fairly uncharted territory of resource-aware (functional) languages is Hume [16]. 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.

Bondavalli et al. [7] present a simple in-the-large programming language to describe the structure of a graph-based application. However, they only model what we call components and simple edges, whereas their simple language neither accounts for multi-version components nor for complex communication structures, not to mention any notion of non-functional properties.

A term related to coordination is *algorithmic skeletons*. Merely as examples we mention FastFlow [2] and Musket [20]. Again, all work in this area that we

are aware of in one way or another focuses on the trade-off between programming efficiency and execution performance, whereas our focus is on energy, time and security as non-functional properties.

Lustre [6,15] was designed to program reactive system, such as automatic control and monitoring systems. In contrast to general-purpose programming language, Lustre models the flow of data. The idea is to represent actions done on data at each time tick, like in an electronic 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).

Lustre is synchronous which seems necessary for time-sensitive applications. However, Lustre does not decouple the program source code from its structure. The flow of data is extracted by the compiler through data dependencies of variables. 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.

In [3] Lustre is extended by meta-operators to integrate a complete model-based design tool from a high-level Simulink model to a low-level implementation. Still, this extension, called Lustre++, does not separate the design of the program structure from actual feature implementation and remains at a too low level to only represent application structure as we intend to do.

The StreamIT [27] language also describes graph-based streaming applications, but it is restricted to fork-join graphs while we need to support arbitrary graphs, possibly with multiple sources and/or sinks.

The Architecture Analysis & Design Language (AADL) [11] targets real-time system design. It provides formal modeling concepts for the description and analysis of application architectures in terms of distinct components and their interactions. AADL supports early prediction and analysis with respect to performance, schedulability and reliability.

7 Conclusion

We propose the TeamPlay coordination language and component technology for the high-level design and development of cyber-physical systems. Our coordination DSL allows users to specify non-trivial streaming applications in a few lines of code while treating crucial non-functional properties such energy, time and security as first-class citizens throughout the application design process.

We describe a complete tool flow from syntactic and semantic validation of coordination programs to code generation for typical off-the-shelf heterogeneous multi-core hardware for cyber-physical systems. Our tool flow includes a variety of offline and online scheduling and mapping techniques that form a tool box, from which the user can choose the most appropriate combination with respect to application needs.

We apply our approach to a real-world use case: a mission-critical reconnaissance drone. We demonstrate the merits of our approach in terms of specification conciseness. An initial version of our tool chain is functional, and we have run

preliminary experiments on an actually flying drone. However, the outcome of these experiments is still under analysis and beyond the scope of this paper.

Our work continues in multiple directions. We currently work on a number of further application use cases, among others a car park monitoring system, a satellite communication system and a camera pill application from the medical domain. Further experience with these additional use cases will most likely motivate us to refine the design of our coordination DSL.

Implementation-wise we plan to extend and refine the various scheduling and mapping options. Our code generator currently expects a Linux-like environment with a certain level of operating system support. This is a realistic assumption for many cyber-physical systems, but others run in more bare-metal environments, e.g. where the form factor requires minimal computing hardware. Our more long-term vision is to adapt our coordination technology for safety-critical applications that must be secured against component failure or cyber attacks.

References

1. Achten, P., Plasmeijer, M.: The ins and outs of Clean I/O. J. Funct. Program. **5**(1), 81–110 (1995)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multicore. In: Programming Multi-core and Many-core Computing Systems. Wiley (2017)
3. Alras, M., Caspi, P., Girault, A., Raymond, P.: Model-based design of embedded control systems by means of a synchronous intermediate model. In: International Conference on Embedded Software and Systems, pp. 3–10. IEEE (2009)
4. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
5. Arbab, F.: Composition of interacting computations. In: Goldin, D., Smolka, S., Wegner, P. (eds.) Interactive Computation, pp. 277–321. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-34874-3_12
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages 12 years later. Proc. IEEE **91**(1), 64–83 (2003)
7. Bondavalli, A., Strigini, L., Simoncini, L.: Dataflow-like languages for real-time systems: issues of computational models and notation. In: 11th Symposium on Reliable Distributed Systems (SRDS 1992), pp. 214–221. IEEE (1992)
8. Casini, D., Biondi, A., Nelissen, G., Buttazzo, G.: Partitioned fixed-priority scheduling of parallel tasks without preemptions. In: 2018 IEEE Real-Time Systems Symposium (RTSS 2018), pp. 421–433. IEEE (2018)
9. Ciatto, G., Mariani, S., Louvel, M., Omicini, A., Zambonelli, F.: Twenty years of coordination technologies: state-of-the-art and perspectives. In: Di Marzo Serugendo, G., Loreti, M. (eds.) (COORDINATION'18). LNCS, vol. 10852, pp. 51–80. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-92408-3_3
10. Falk, H., Lokuciejewski, P., Theiling, H.: Design of a WCET-aware C compiler. In: 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTIMedia 2006), pp. 121–126. IEEE (2006)
11. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis and design language (AADL): an introduction. Technical report, Carnegie-Mellon University, Pittsburgh, USA, Software Engineering Institute (2006)

12. Ferdinand, C., Heckmann, R.: aiT: worst-case execution time prediction by static program analysis. In: Jacquart, R. (ed.) Building the Information Society. IIFIP, vol. 156, pp. 377–383. Springer, Boston (2004). https://doi.org/10.1007/978-1-4020-8157-6_29
13. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35**(2), 97–107 (1992)
14. Grelck, C., Scholz, S.B., Shafarenko, A.: Asynchronous stream processing with S-Net. Int. J. Parallel Prog. **38**(1), 38–67 (2010). https://doi.org/10.1007/s10766-009-0121-x
15. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)
16. Hammond, K., Michaelson, G.: Hume: a domain-specific language for real-time embedded systems. In: Pfenning, F., Smaragdakis, Y. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 37–56. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39815-8_3
17. Jongmans, S.S., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
18. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM (JACM) **20**(1), 46–61 (1973)
19. Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., Buttazzo, G.C.: Response-time analysis of conditional dag tasks in multiprocessor systems. In: 27th Euromicro Conference on Real-Time Systems (RTS 2015), pp. 211–221. IEEE (2015)
20. Rieger, C., Wrede, F., Kuchen, H.: Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In: 34th ACM Symposium on Applied Computing (SAC 2019), pp. 1534–1543. ACM, New York (2019)
21. Ritchie, D.M., Kernighan, B.W., Lesk, M.E.: The C Programming Language. Prentice Hall, Englewood Cliffs (1988)
22. Roeder, J., Rouxel, B., Altmeyer, S., Grelck, C.: Interdependent multi-version scheduling in heterogeneous energy-aware embedded systems. In: 13th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2019) of the 27th International Conference on Real-Time Networks and Systems (RTNS 2019) (2019)
23. Rouxel, B., Skalistis, S., Derrien, S., Puaut, I.: Hiding communication delays in contention-free execution for SPM-based multi-core architectures. In: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (2019)
24. Rusu, C., Melhem, R., Mossé, D.: Multi-version scheduling in rechargeable energy-aware real-time systems. J. Embed. Comput. **1**(2), 271–283 (2005)
25. Seewald, A., Schultz, U.P., Roeder, J., Rouxel, B., Grelck, C.: Component-based computation-energy modeling for embedded systems. In: Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. SPLASH Companion 2019. ACM, New York (2019)
26. Tendulkar, P., Poplavko, P., Galanommatis, I., Maler, O.: Many-core scheduling of data parallel applications using SMT solvers. In: 17th Euromicro Conference on Digital System Design (DSD 2014), pp. 615–622. IEEE (2014)
27. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: a language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_14
28. Wadler, P.: The Essence of Functional Programming. In: 19th ACM Symposium on Principles of Programming Languages (POPL 1992), pp. 1–14. ACM Press (1992)