Systems Engineering for Industrial Cyber-Physical Systems using Aspects

Ilge Akkaya, *Member IEEE*, Patricia Derler, Shuhei Emoto, Edward A. Lee, *Fellow IEEE*

***Abstract*—One of the biggest challenges in cyber-physical system (CPS) design is their intrinsic complexity, heterogeneity, and multidisciplinary nature. Emerging distributed CPS integrate a wide range of heterogeneous aspects such as physical dynamics, control, machine learning, and error handling. Furthermore, system components are often distributed over multiple physical locations, hardware platforms and communication networks. While model-based design (MBD) has tremendously improved the design process, CPS design remains a difficult task. Models are meant to improve understanding of a system, yet this quality is often lost when models become too complicated. In this paper, we show how to use aspect-oriented (AO) modeling techniques in MBD as a systematic way to segregate domains of expertise and cross-cutting concerns within the model. We demonstrate these concepts on actor-oriented models of an industrial robotic swarm application and illustrate the use of AO modeling techniques to manage the complexity. We also show how to use AO modeling for design-space exploration.**

***Index Terms*—Actor-oriented modeling, aspect-oriented modeling, model-based design, cyber-physical systems, robotic swarms**

I. Introduction

Cyber-physical system design integrates a wide variety of heterogeneous disciplines, including control engineering, mechanics, thermodynamics, sensors, electronics, networking, and software engineering [1]. Engineers use domain-specific tools and techniques in each of these disciplines, but integration of the diverse tools and techniques remains challenging [2]. MBD [3] has proven successful in several of these domains, including, for example, modeling and simulation of physical dynamics using Modelica [4], design, simulation, and code generation of control systems using Simulink® (by MathWorks), design of instrumentation systems using LabVIEW® (from National Instruments), modeling and simulation of communication networks using OPNET Modeler® (by Riverbed) and ns-3 (http://www.nsnam.org/), and modeling of software architecture using the unified modeling language (UML) and the architecture analysis and design language (AADL). Integrating these tools and techniques, however, remains a daunting challenge [5], [6]. Accidental complexities such as incompatible design data representations, unpublished APIs, and vague or unspecified semantics dominate, so that it is rare to achieve an effective integration. System integration ends up happening late in the design process, when prototypes of all components are available, and problems that emerge at that stage can be very expensive to fix.

In this paper, we describe an approach that focuses not on integration of tools, but rather on integration of modeling methods. Models improve understanding, analysis, and certifiability of systems. They can reduce the time it takes to introduce new products, as has been very effectively demonstrated in the automotive industry. Although CPS are intrinsically complex, simple models are more useful than complex models, because they are more understandable, analyzable, and certifiable. The goal in this paper is to improve the degree to which a model can accurately reflect the complexity in system design without making the models themselves so complex as to degrade their utility.

A key technique for keeping models simple is *abstraction*. In the 1960s, object-oriented (OO) design methodologies were introduced with the language Simula. The main idea behind OO programming is the separation of concerns into objects with well-defined interfaces. Data and access to data are encapsulated within the object, and the internal implementation is hidden. Code duplication is reduced by inheritance, which enables common code reuse in related objects. OO concepts enable developers to think of a problem in terms of abstract data types and relationships between them. The technique breaks down complex problems into simpler subproblems. OO programming is widely successful and enables more effective development of complex software systems.

Software engineers, however, frequently discover that inheritance mechanisms are inadequate, and that leveraging pre-existing code often requires modifications to the existing code. Consider, for example, a new requirement in a software engineering project that all operations of a certain kind must be logged. It is difficult in existing OO languages to insert such capability without making considerable changes to the existing code base. Moreover, after making those changes, the software design has tangled distinct, cross-cutting concerns, the original functionality, and the logging.

In the late 1990s, Kiczales et al. introduced aspect-oriented programming [7] to deal with such cross-cutting functionality. For the logging problem, instead of inserting logging code in various places of the original code base, a logging "aspect" is automatically "woven" with the code base as part of the

Manuscript received May 30, 2015; revised October 29, 2015; accepted December 16, 2015. This work was supported in part by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

I. Akkaya and E.A. Lee are with the Electrical and Computer Sciences Department, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail:ilgea@eecs.berkeley.edu; eal@eecs.berkeley.edu).

P. Derler is with National Instruments, Berkeley, USA (e-mail: patricia.derler@ni.com).

S. Emoto is with IHI Corporation, Yokohama, Japan (e-mail: syuhei_emoto@ihi.co.jp).

compilation process. This allows separate development and maintenance of the original functionality and the logging capability. Aspects have been embedded in programming frameworks such as AspectJ [8] and Spring [9], which extend Java with specific syntax supporting aspects.

AO programming has some clear benefits. Code tangling, code duplication, and scattered code can be reduced through abstraction and modularization. However, the adoption of AO programming has been slow. Studies on fault-proneness of aspect-oriented programs [10] show that the *obliviousness* property in AO programming causes faults due to lack of awareness among base code and aspects. Moreover, debugging aspects can be difficult [11].

Although OO design provides a measure of modularity, it contributes nothing to dealing with concurrency. A modularization technique that complements objects is actor-oriented design [12]–[14]. Actors are components (that, in fact, can also be objects), which execute concurrently and communicate via messages, vs. the procedure calls commonly used in object-oriented languages. Many of the model-based design tools discussed above, such as Modelica, Simulink, LabVIEW, and ns-3, support actor-oriented design.

The semantics of actors and the communication between them can vary widely across domains. A concurrency model and communication strategy for actors form a concurrent *Model of Computation* (MoC), and distinct MoCs can be combined to create heterogeneous models [15].

This paper shows how the ideas introduced in AO programming can be applied not only to OO design, but also to actor-oriented models. We focus on models in CPS domain that are highly heterogeneous in nature, causing them to become hard to validate and verify due to the inherent complexity of CPS.

It is common to use the term "functional model" for a model that describes the core intended behavior of a system. For example, a functional model of a cooperative cruise control system may describe the feedback control laws and the physical dynamics of a car. So called "non-functional" aspects might include properties of an implementation such as communication latencies, faults, and energy consumption. Of course, what is viewed as "core intended behavior" will depend on who is building the model. But generally, when an engineer builds a model, that engineer has some view of a "core intended behavior" and some notion of other concerns that may be important, but are distinct from the core functionality. Our goal is to treat these "non-functional" concerns as aspects, specifically to be able to model these without entangling them with the core intended behavior. Examples of such concerns that arise in CPS are as follows.

Communication. CPS applications are often distributed, sometimes widely. Communication infrastructure is important, and communication artifacts such as delay and losses must be taken into account during system validation. Resource contention and communication delays can lead to sub-optimal system behavior that is hard to predict and is not reflected in a purely functional model. Modeling of networks, however, is itself a sophisticated domain, therefore, entangling network models with core models is undesirable. Aspect-oriented modeling of networks in cyber-physical energy systems is discussed in [16].

Execution time. A model of "core intended behavior" will typically not include execution time of software components. At early stages of a design, this cannot be known, as it requires considerable implementation detail. Even at later design stages, complexity of the underlying architecture can make it difficult to precisely account for execution time [17]. Modeling execution time is again a sophisticated domain, and such models should not be entangled with core intended behavior.

Architecture. Design choices such as a hardware architecture (multicore computers, centralized triple-redundant machines, distributed microcontrollers, etc.), scheduling strategies (time triggered, earliest deadline first (EDF), etc.), and mapping of software functions onto hardware resources can profoundly affect system cost and behavior. If we can avoid entangling architecture models with core intended behavior, then we can facilitate design space exploration at different levels of abstraction [18].

Error handling. Error handling is necessary in real systems, but the additional logic can clutter the core model. Aspect-oriented modeling helps in separating error handling logic from the core model. In addition, proper error handling requires fault models, and testing the system under fault conditions can also be handled through aspects.

Logging and debugging. Logging and debugging features are often heavily used during development and less in a deployed system. Factoring out the logging and debugging infrastructure makes it easier to include or exclude.

Verification. Recent work on design contracts [19] formalizes high-level system requirements. Aspects can be used to automatically check compliance with these requirements during system development, and to detect fault conditions that result in contract violations in a deployed system.

Fault modeling and anomaly detection System modeling primarily aims at representing the nominal behavior of a system. In many CPS domains such as automotive, aerospace, and manufacturing, fault models are an essential part of system development. Faults and anomalies that could occur in a system can be factored as aspects. This naturally segregates faulty behavior from the nominal system workflow, and enables efficient modeling of cascading faults.

Security concerns Wasicek et. al. [20] present aspect-oriented modeling as a model-based design technique to assess the security of CPS. By associating attack models with the CPS in an aspect-oriented manner, the designer can gain insights into the behavior of the CPS under attack.

Dynamics and sensing models The foundation of many CPS aspects such as control, machine learning, and optimization build upon mathematical models of a physical process. Consistency of the common mathematical models across the system becomes prone to errors due to the composition of components from different areas of expertise. Using aspects enables these shared mathematical models to be explicitly factored out, and reused by multiple parts of the CPS model.

In this paper, we describe in detail an infrastructure for aspect-oriented modeling (AOM) that we have realized in the Ptolemy II [15] framework. The techniques presented in this paper should be easily implementable in many commonly used design tools. They can thereby provide the foundation for effective systems engineering of complex industrial cyber-physical systems.

II. Related Work

Since AOM is often considered an extension to object-oriented modeling, various AOM extensions to OO languages have been proposed. Naturally, UML-based AOM design approaches have been developed [21]. While some approaches provide general-purpose AOM languages, others only focus on specific aspects. Our approach is a general purpose AOM language in that it allows the definition of arbitrary aspects.

Some work on UML AOM extensions is related in that they focus on CPS aspects that we also explore here. For instance, Liu and Zhang [22] present an aspect-oriented framework that combines UML profiles and real-time logic (RTL) for specifying QoS properties such as timing, reliability, and safety. Recent work on using AO modeling for automation systems is presented by Wehrmeister et al. in [23]. Espinoza et al. [24] describe annotations of schedulability and performance analysis data in UML models.

The main differentiating factor between our work and UML based AOM extensions is the purpose of the models. We design executable models, i.e. models have a clear, deterministic, and concurrent semantics, whereas UML models usually do not have execution semantics.

Mechanisms similar to the ones we use to weave aspects in model-based design environments are introduced in Gray et al. [25], where AOM is used for domain specific, cross-cutting constraints.

Our work is influenced by Metropolis [26]. Here, so-called "quantity managers" assign quantities to events, which in turn are scheduled by the framework. The concept is similar to ours in that common aspects are abstracted away from the functional model and added later via quantity managers and schedulers. The Metropolis project facilitates platform-based design [18], which allows for co-design of functional and architecture models that are evaluated as a whole.

AADL, the architecture analysis and design language (formerly known as avionics architecture description language) [27] aims at modeling and analyzing complex architecture models. Architectural descriptions can be extended with annexes, such as the behavior annex that allows a state machine description of component implementations, or the error annex, which supports fault modeling. De Niz et al. discuss different aspects and the separation of concerns, in particular nonfunctional concerns, in AADL [28]. AO4AADL [29] is an aspect-oriented extension to AADL to master complexity and ensure scalability.

III. Aspect-oriented modeling on an example

Industrial applications, including factory automation, assembly, monitoring, and disaster response rely on cooperative behavior of humans and machines. Robotic swarms, which extend the concept of cooperative machine intelligence, fit well into this framework. "Swarm intelligence" is defined as a system property that arises from the interaction of non-intelligent mechanical robots to collectively form an "intelligent" system [30].

One example in an industrial setting is a disaster response scenario, where the consequences of a disaster may have rendered a physical space hazardous for humans to explore. In addition, parts of the site may not have ground access at all, due to debris caused by the disaster. Collaborative mapping of earthquake-damaged buildings is an example of this, where ground and aerial robots need to work together to create a map of the hazardous site [31].

In addition to exploration and risk assessment, robots can also perform mechanical tasks in sites that are inaccessible to humans. The Fukushima Daiichi nuclear reactor cleanup and investigation effort Japan provides such a scenario. Robots are currently being deployed inside the reactors to collect critical information on radiation levels, as well as on safe paths to be taken in subsequent missions [32]. Larger robots are being employed to carry out mechanical tasks within the reactor, such as to deploy vacuum and filtration systems that reduce nuclear contamination and allow human operators to safely enter the site [33].

Motivated by such scenarios, we illustrate the concept of aspect-oriented modeling on the design of such robotic swarms. A simplified top-level model is shown in Figure 1, where two sets of robots are deployed: (i) a team of lightweight *observation robots* that are equipped with sensors whose task is to cooperate to assess an environment in order to locate a target in the presence of potential hazards, and (ii) a heavy-duty *main robot*, with limited maneuvering abilities, whose task is to retrieve a target from an unfamiliar environment.

In the model, a hierarchical control strategy is being deployed to fully enable swarm intelligence. First, an observation-optimizing controller receives sensor measurements from observation robots, estimates the position of the target based on the robot dynamics and sensor readings, and steers the observation robots, subject to dynamics and safety constraints, to reduce *uncertainty* in target position in the subsequent control steps. Meanwhile, a higher level controller receives the estimated target position, and steers the main robot towards the target. The sensors, of course, have limited sensing capability. For example, in our model, we assume that they can only make imperfect measurements of their distance from the target.

The modeling environment used here is Ptolemy II [15], which is a framework for building and simulating actor-oriented models of heterogeneous systems. A Ptolemy model consists of actors that communicate via ports. The semantics of the communication, also referred to as the model of computation, is given by so-called *directors*. Different MoCs can be combined hierarchically to represent heterogeneous systems.

The robots and controllers are composite components, where a single icon represents a potentially complex submodel. The components communicate via time-stamped events, using the discrete-event MoC, represented in the model by the `DE Director`.

Such a model, which captures component interactions alone, is often created by a system designer for evaluation of functional behavior and timing analysis of the application. Once the behavior has been evaluated and deemed suitable, the next step is deployment, during which the timing behavior of the application might change. For example, in deployment,

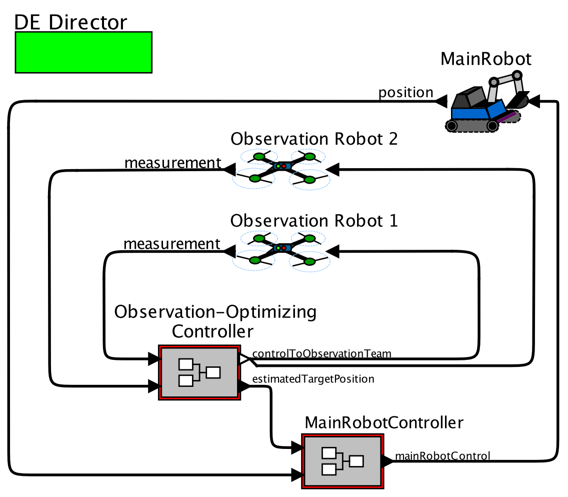

Fig. 1: A model of a robotic swarm, cooperating to carry out a target retrieval task within an unfamiliar environment

the communication between controller and the robots will be subject to context-dependent latency, packet losses, and re-ordering of messages.

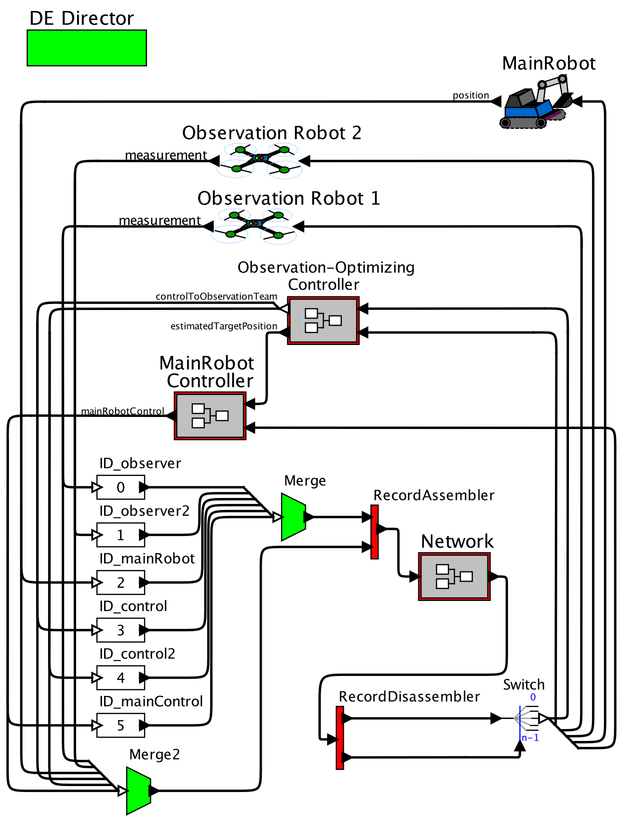

Fig. 2: A poor model of a robotic swarm with network specifications.

One possible model that includes the communication aspects is given in Figure 2. In this model, the `Network` actor represents the shared communication resource. It is a hierarchical component that can include delays, losses, and re-ordering. Such a hierarchical component could include sophisticated models of specific networking technologies, including, for example, WiFi interference and MAC (media access) protocols.

The model in Figure 2, however, is awkward. In order for the communication aspects to affect all relevant communication paths, the model builder is forced to model the construction of packets that include addressing information and to multiplex these packets through a single model of the communication fabric. The designer is often not interested in such low level modeling details but only in the high-level effect of the network on application behavior. Moreover, the logical communication paths of the original models have been lost, eliminating many of the advantages of a visual modeling syntax.

Nevertheless, a Monte Carlo simulation, yielding results such as that in Figure 3, can be used to study the behavior of the system with and without network models. These results illustrate the detrimental effects of packet losses and latency, and an engineer can determine thresholds on networking behavior that would put the target retrieval mission at risk.

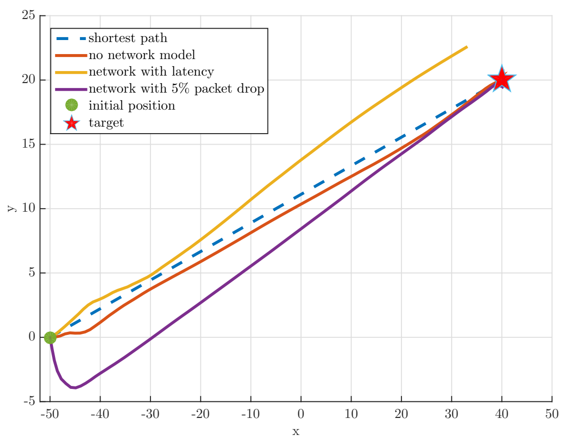

Fig. 3: Simulated effect of network aspects on main robot trajectory.

It is important to observe that the network fabric used to enable communication between components is not an intrinsic part of the system design; it is only an implementation choice. Therefore, the communication via a shared resource can be considered an *aspect* of the system and can be modeled as such. An aspect-oriented alternative that uses our Ptolemy II extensions is shown in Figure 4. The network aspect, represented by an icon at the top, models the delay and resource contention of communication; connections that use the network, namely, the channels between robots and controllers, are annotated with textual parameters that bind those communication links to the network model. This preserves the benefits of the visual syntax, in that, the new model still visually represents logical communication paths. But more importantly, it abstracts away the low-level details of the network implementation, such as packet structure and addressing. These were needed in Figure 2 as an accident of the modeling technique, whereas in the aspect-oriented alternative, the low-level routing functionality is handled internally by the aspect. The aspect illustrated here is a *Communication aspect*. There are many more aspects that can be modularly incorporated into a model in a similar way, as will be explained next.

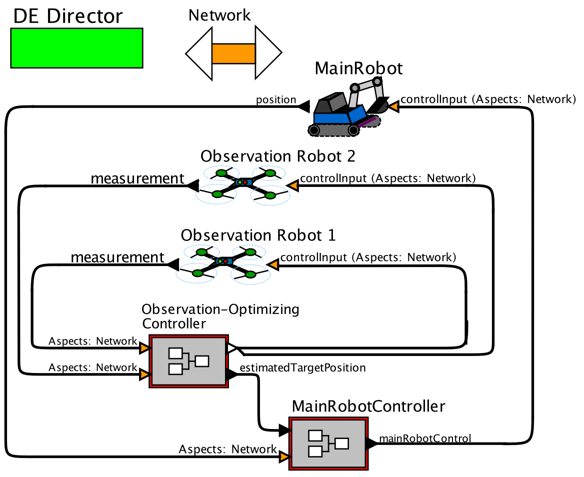

Fig. 4: A model of a robotic swarm with network specifications modeled as aspects

IV. Aspect-Oriented Actor-Oriented Modeling

First, let us review the basic ideas of actor-oriented programming. Actor-oriented programs compose concurrent components that communicate via messages. This complements object-oriented languages, where components are abstract data structures that interact via procedure calls. An actor has a set of input ports, a set of output ports, and state. Actors communicate by sending messages through ports. The semantics of this communication is defined by an MoC that is implemented by an execution engine.

An actor model can be annotated with additional information that is orthogonal to the information in the model. This information can be used to statically evaluate the model or to modify execution. An example would be evaluation of the cost of implementing a system. By annotating each model element with a price, the cost of the entire system can be computed in design time. The cost can be compared to a constraint, for instance an upper bound on cost.

With aspect-oriented modeling we go a step further and annotate models with information that is evaluated dynamically and can change the system behavior. An example is the communication aspect described above. In order to explain the concepts further, we introduce some terminology.

- *Join point*. A point during the execution where the program interacts with an aspect. In AO programming languages, this often refers to a method execution.
- *Advice*. Action taken by an aspect at a particular join point. An advice can be configured around, before, or after the join point.
- *Pointcut*. The point during the execution where an advice should be executed. Pointcuts might depend on runtime information.
- *Aspect*. The advice together with the pointcut.
- *Weaving*. Linking aspects with the application, which can happen at compile time or at run time.

An *advice*, in the scope of this paper, represents an actor that implements the cross-cutting concern. In the previous example, the advice is the `Network` actor.

A *join point* is the execution of an actor or the transmission of a token. While advice actors do not encode any information about the model they are used in, they know at which join points they can be used. For instance, the `Network` can only be associated with connections between actors. We generalize this to input ports on actors, meaning that if the `Network` is associated with an input port, the incoming connection is enhanced by the *advice*. The fact that *advice* actors can only be used on certain join points can be utilized to guide the model builder. In Ptolemy II, upon dropping an advice actor into a model, parameters are added to all the possible join points in the model, at which the use of the advice can either be enabled or disabled by the model builder.

A *pointcut* is a join point where the use of the advice is enabled. An *aspect* is then all enabled join points together with the advice.

Ptolemy II handles the *weaving* at run time. A simulation framework where code is generated before simulation would need to include the weaving in the generated code.

In many AO programming languages, pointcuts rely on naming conventions to find the places in the program where aspects should be executed. Thus changes in the program can break the AO program, a problem often referred to as the fragile pointcut problem [34] or "unintended consequences" in AOP. In our implementation, advices need not be aware of the model and pointcuts that are defined. Deleting an advice from the model automatically results in a deletion of all pointcuts.

In this work, we investigate aspects with two types of pointcuts: on input ports of actors and on actor executions. An aspect with a pointcut on an input port of an actor executes the advice whenever a token is sent to this input port. The advice can modify the token (e.g., probabilistically dropping it) or perform other operations. In the previous example, the `Network` advice implements resource contention and packet drops on tokens.

Figure 5 illustrates the concepts introduced on an abstract example. The model contains 5 actors, A_1,A_2,A_3,c and e, where c and e are advice actors. Actors A_1 and A_2 communicate as illustrated by their connections. The advice c can be enabled on communications between actors, thus the communication between A_1 and A_2 as well as the communication between A_1 and A_3 form join points. In the example, the advice is only enabled on the communication between A_1 and A_2, which describes a pointcut. Advice actor e can be enabled on actor executions, so its join points are actors. In this example, e is enabled on actor A_1 and actor A_3, but not on A_2. In the figure, enabled join points are illustrated by highlighting.

V. Aspect-oriented modeling in Ptolemy II

To implement join points and pointcuts in Ptolemy II, we use the decorator pattern [35], which allows adding behavior to an individual object without affecting the behavior of other objects of the same class. We implement advices as *decorators* that decorate join points with additional attributes.

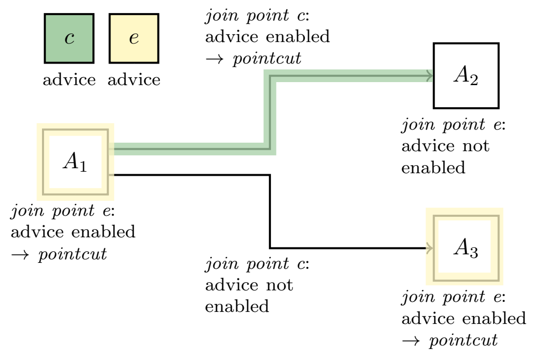
c
e
advice advice
join point c: advice enabled → pointcut
A2
join point e: advice not enabled
A1
join point e: advice enabled → pointcut
join point c: advice not enabled
A3
join point e: advice enabled → pointcut

Fig. 5: Aspects in actor-oriented models.

Figure 6 illustrates the implementation of advices that decorate communication between actors in Ptolemy. A communication advice decorates all ports with a boolean attribute *enable*. If this enable flag is true, the receiver in the port is wrapped by an intermediate receiver, which intervenes in the communication and coordinates with the aspect actor. Communication advices can be composed serially. In this case, the original receiver is wrapped by multiple intermediate receivers, providing associations with multiple advices. The order in which communication aspects are enabled can be controlled by the model builder.

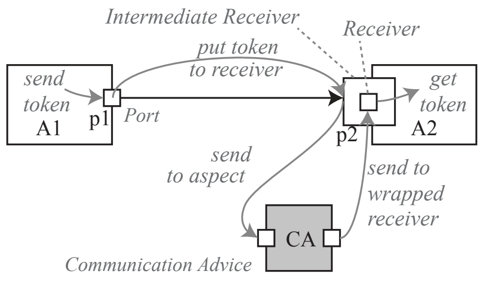

Fig. 6: Advice with join points on actor communication

The implementation of an *execution* advice (vs. a communication advice) is illustrated in Figure 7. Similarly to the communication aspect, an execution aspect decorates *actors* (vs. ports) in a model with an *enable* flag. If this flag is set to true, the advice actor will be consulted each time the actor is executed. This is implemented by modifying the director. The director (a) selects the next actor to be executed and (b) executes the actor by calling its fire function. Between (a) and (b), a call to the advice is inserted. In case the advice decides that the actor cannot be executed, the director can choose another actor to be fired. *Weaving*, the process of linking aspects with the application, is performed by the Ptolemy II runtime. Part of the weaving is implemented by the director that executes this model.

VI. Other application Areas of AOM

In section III we showed how to use aspects to model network fabrics. In this section, we will present a set of other essential modeling concerns addressed by aspects.

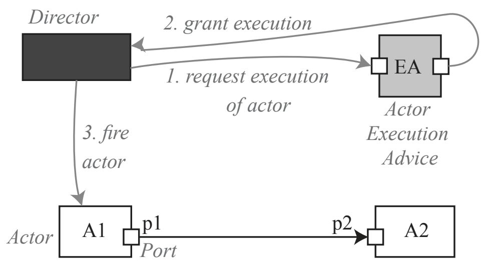

Fig. 7: Advice with join points on actor execution

A. Sensor Models and Robot Dynamics

In industrial settings, mathematical models of component dynamics, kinematics, as well as models of uncertainty and noise are key to building complex systems. Moreover, these mathematical models are often shared between different pieces of the functional model. Consider the observation team in the context of the robotics disaster response scenario. Here, robot dynamics, often given by a state-space representation, are used to model the behavior of observation robots in a physical environment. Also, noise characteristics of on-board sensors are required for accurate quantification of the information content of sensor measurements. Clearly, these dynamics models need to interact with one another for cooperative inference and control algorithms.

As an example, consider the *target estimation* problem carried out by the `Observation-optimizing Controller` in Figure 4. An implementation of this hierarchical component is given by the submodel in Figure 8. In this model, robot dynamics are modeled as aspects, which contain the state space representations and process noise of components. Also, the range sensors on board each robot are modeled as aspects, which specify sensor measurement models and noise. The component annotated as “mutual information based optimization” contains an internal workflow consisting of state estimation and optimization actors that estimate target position given noisy range measurements, and by taking the models of robot dynamics into account, optimize future robot trajectories. The objective function for such a control strategy often uses a mutual-information metric, such that the set of future trajectories is optimized and the collective set of robot observations provides as much information as possible about target location. Design and implementation of such control strategies are described as part of a case study in [36].

In general, the “mutual information based optimization” component would be designed by a specialist in machine learning and control, who would reuse dynamics and sensor models inherited from other system components to populate learning and optimization tasks. Using aspects to represent mathematical models of such system properties therefore increases modularity and simplifies the design of control strategies. It also ensures that a re-design of one of the robot dynamics, or replacement of a sensor does not require any changes to the inference and optimization components. Therefore, separation of concerns is ensured during design and development.

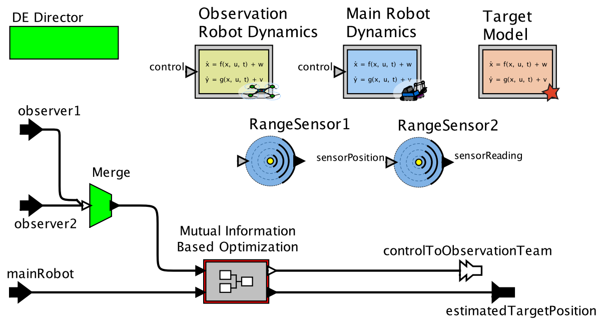

Fig. 8: The `Observation-Optimizing Controller` demonstrates AO modeling of sensors and robot dynamics

B. Execution

In many embedded control applications, execution time of software influences the behavior of the application. Design space exploration is necessary to evaluate the behavior of different implementations. To this day, tool support for this activity is very limited. In this next example, we illustrate how one could build simple models of CPUs as *advices*, and associate actor executions with these advices using AOM.

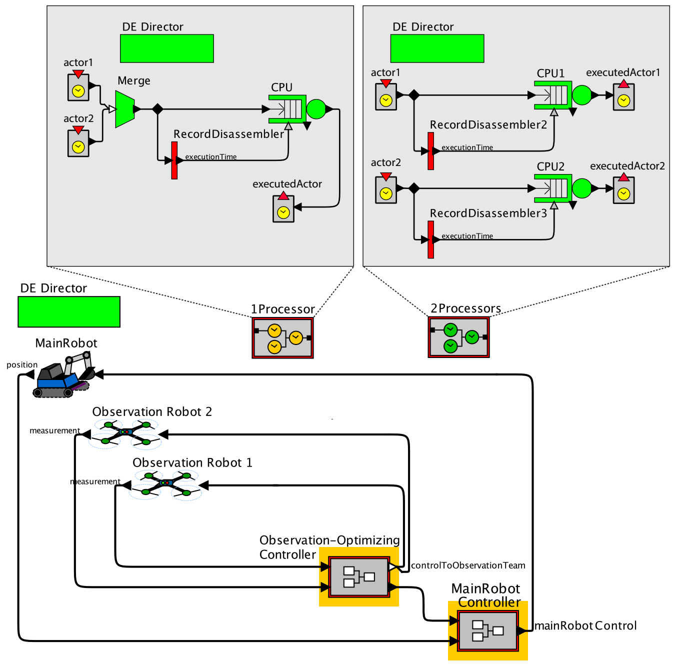

Fig. 9: Mapping of functional models onto architecture models using aspects

In cooperative control applications, an execution bottleneck is often caused by on-line control algorithms. In the running example, it would be of interest to model execution times for the two controller models to ensure that application requirements are met at runtime. As an example, suppose we want to evaluate two alternative architectural designs. In the model depicted in Figure 9, two execution advices are implemented as composite actors. In the figure, the two advices represent two alternative architectures, one with a single core and one with two cores. In the figure, the `1Processor` advice is enabled on the `Observation-Optimizing Controller` and the `Main Robot Control` actors. The 1-processor model merges incoming execution requests and schedules them on a server that delays the actors for a specified amount of time, emulating execution time. A more elaborate model could include a scheduling policy such as EDF. In addition to the *enable* flag, an *execution time* parameter is added to all join points by an execution advice. As a result, every actor can be simulated to have a different execution time. As this parameter is read every time the actor is executed, it is possible to provide execution times that differ in between iterations. The join points are also extended with an *execution request port*, which is a special actor inside the advice that receives *execution request tokens* when the actor is scheduled to be fired. In the example, these execution request ports are `actor1` and `actor2`. The execution request token contains an object reference of the actor to be fired and the execution time for the current firing. Before firing an actor that is decorated with a processor advice, the director triggers a firing of the advice and generates an execution request token for the actor in the appropriate execution request port.

Figure 10 shows an execution of the main robot controller and the observation-optimizing controller in the single and two processor cases, respectively. The illustrated execution monitor is part of the advice implementation. The execution simulations are obtained with a global sampling period of 250 ms, and worst-case execution times for Observation-OptimizingController and MainRobotController set to 250 and 50 ms, respectively. It is seen that this particular execution model is not schedulable on the single core architecture for the given execution time and sampling period parameters.

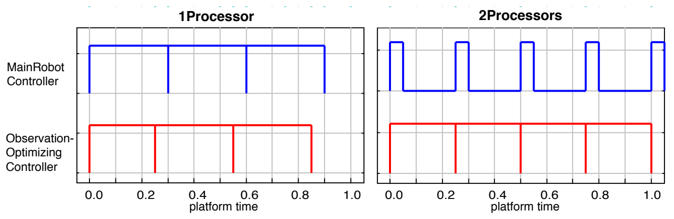

Fig. 10: Controller execution times on different architectures

To keep the illustration simple, the model shown here is naïve. Execution times of the components are difficult to know precisely, and may need to be modeled probabilistically or to rely on sophisticated program analysis tools. Similarly, models of scheduling policies can get quite sophisticated, and the effects of resource contention in the processor architecture can get complex. But because of the effective separation of concerns, an expert on modeling and simulation of real-time software systems can focus on the design of the aspect model, while the expert on machine learning and optimization focuses on the control design. Very little coordination is required between the two.

To further demonstrate the interaction of an enabled execution aspect with the functional model, consider Figure 11, which illustrates the impact of a slow processor on the

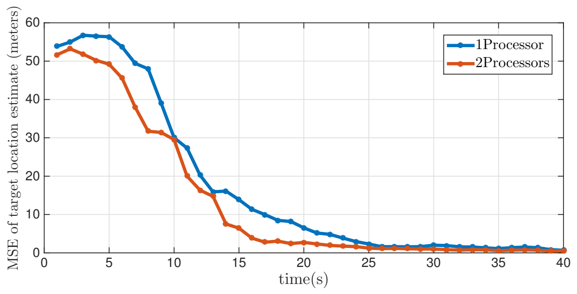

Fig. 11: Effect of processor architecture on target estimation timing and functionality

target-estimation accuracy, as obtained by the Observation-Optimizing Controller. Although information from the observation team is still processed in order, the processing delay causes a slower convergence to the true target state, impacting the real-time performance, which, by an execution aspect, has been detected before deployment.

C. Fault Modeling

Fault models are among the most natural aspects in a multi-view system. By definition, faults are not part of the system specification itself. An orthogonal modeling paradigm helps fault models to be integrated with the system design in a way that preserves separation of concerns. In many standardized architecture description languages, error models are part of the native system specification. For instance, AADL features an error annex describing numerous fault models [37].

Faults are orthogonal concerns that can be modeled as aspects. Figure 12 demonstrates the top-level model of our running example, which has been decorated by two fault models: (i) a stuck-at component fault model that affects the range sensor readings of one of the observation robots, (ii) a packet drop fault model that affects the controller-to-robot communication for one of the observation robots.

Figure 13 illustrates the implementation of the `StuckAtFault` aspect shown to affect the output values of the `RangeSensor` component. A stuck-at fault occurs when the individual signals and pins are stuck at a fixed value on an integrated circuit. As an abstraction, such faults can be modeled to occur on a single input or output of a component. With the addition of this aspect to the functional model of `Observation Robot 2`, the sensor output values produced on board this robot will get stuck at a fixed value with some probability during execution.

A packet-drop fault, as the name suggests, models a dropped packet over a network link, often mimicking a packet erasure channel with an assigned packet erasure probability, applying independently to each transmitted packet. The `PacketDropFault` aspect that affects transmission of control inputs to `Observation Robot 2` is shown in Figure 14.

D. Fault and error handling

Errors can occur due to software and hardware failures. While most software errors are eliminated through validation

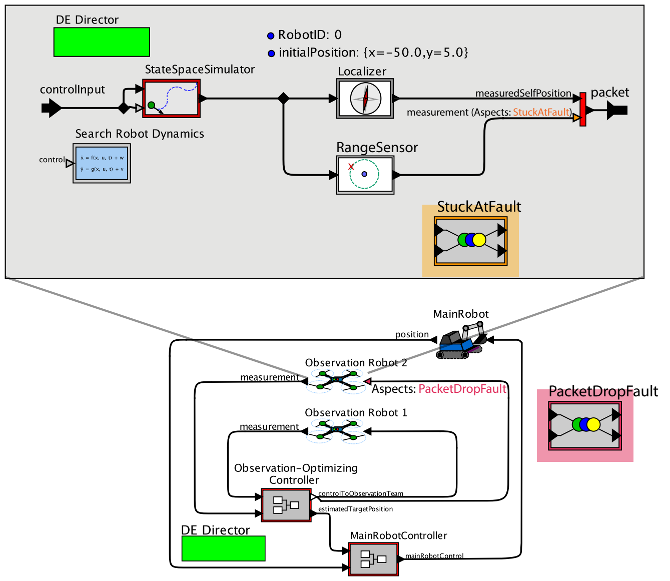

Fig. 12: Aspect-oriented component and communication fault models

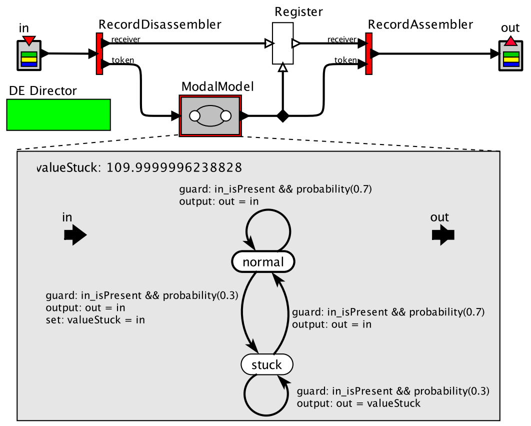

Fig. 13: Fault model of a stuck signal.

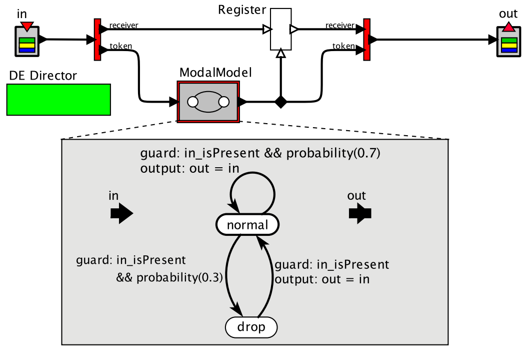

Fig. 14: Packet Drop Fault model that drops packets with a probability.

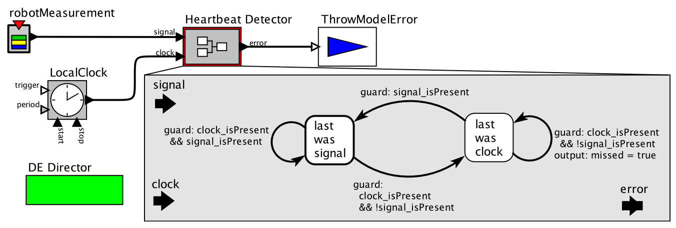

Fig. 15: Aspect-oriented heartbeat detection

and verification, hardware errors cannot be eliminated easily, thus precautions have to be taken. Error handling code must be inserted to catch possible failure situations. Because errors can occur in many different places, mixing error handling functionality with the system functionality can make the model difficult to understand. Also, error handling strategies might depend on the operating conditions, and therefore might need to exhibit modal behavior.

Figure 15 shows a model of a heartbeat detector which implements a mechanism for detecting a missed sensor reading. It uses a state machine that expects time stamped sensor readings at its input. In our running example, this input is connected to the `robotMeasurements` port. The *clock* input reads time stamped inputs from a local clock. The state machine keeps track of whether the most recent event was a sensor message or a message from the local clock. It issues a missed event if two consecutive local clock messages were received. See also [38] for a use of the HeartBeatDetector in a power plant control system. This heartbeat detector can be used on signals that come from unreliable sources.

E. Contract Modeling

Formal contracts [39] can be useful in CPS design to clarify interfaces and enforce documented interaction behavior between components. In such contracts, high-level system specifications are formalized as assume-guarantee formulas. Some work has been dedicated to correct-by-construction synthesis of control protocols based on temporal logic formalization of these contracts [40]. In cases where such a controller design is not possible due to complexity, when dealing with legacy systems, or to detect runtime faults that cause contract violations, one might want runtime components that monitor contract compliance. Aspect-oriented modeling enables addition of design contracts to an existing system model. If design requirements are violated in runtime, this can be detected by an aspect-oriented contract monitor.

Consider, as part of our running example, a *collision avoidance* contract defined on the main robot to ensure that the robot does not collide with a detected target or obstacle on the map. The informal contract specification is that "whenever a *proximity alert* flag has been raised, the robot must come to a full stop within X seconds, and wait for the *resolved* signal from the operator before proceeding with its mission."[1] This contract needs to be monitored on the the main robot controller. Figure 16 implements a contract monitor that, upon detecting a violation, raises a flag. This aspect can be easily and unobtrusively woven with a model of this robot system.

[1]This contract can formally be represented by the Signal Temporal Logic (STL) formula: $G(\texttt{proximityAlert} \rightarrow F_{[0,Xs]} G(\texttt{idle}\ U\ \texttt{resolved}))$.

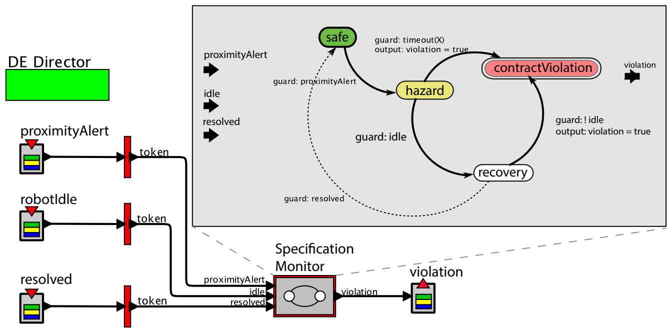

Fig. 16: A collision avoidance contract defined as an aspect on the main robot controller

F. Logging and Debugging

Logging is a common example for aspect-oriented programming. Adding logging code to functional code or models can unnecessarily make the model more complex. Also, at different stages of the design, different information is logged. For deployment, logging is usually completely removed or disabled. Using aspects to perform logging and plotting signals is a much cleaner way to evaluate simulation results. We can modify the previously introduced network model to just log incoming messages and forward them immediately, thus implementing a message logger. An execution logger can be implemented by modifying the 1 or 2 processor models by adding a logging component and removing time delays.

With similar mechanisms as used for logging, breakpoints can be inserted into the execution of a model by using a special actor that pauses the execution (see Figure 17 for a potential collision-avoidance breakpoint).

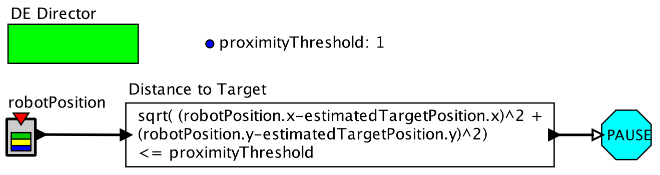
DE Director
proximityThreshold: 1
robotPosition
Distance to Target
sqrt((robotPosition.x−estimatedTargetPosition.x)^2 + (robotPosition.y−estimatedTargetPosition.y)^2) <= proximityThreshold
PAUSE

Fig. 17: Model breakpoints as aspects

VII. Conclusion

This paper discusses how aspect-oriented modeling helps manage the complexity of actor-oriented models of cyber-physical systems. Aspects enable separate development of models for cross-cutting concerns. We show how AOM can cleanly model communication network effects, the execution time of software, hardware design choices, faults, and scheduling policies. We also show how aspects can be used to integrate contract monitoring and logging in models. These cross cutting models enable design-space exploration and facilitate separate development of models for different aspects. All of these capabilities have been prototyped in the open-source Ptolemy II modeling and simulation framework, and all models shown in this paper are executable. Models in this paper are simplified for illustration purposes and use hierarchy to abstract away orthogonal details. Nonetheless, a more comprehensive case study which illustrates the use of aspects in an industry scale application is given in [16].

During this work, we have built up a library of aspects for common cross-cutting concerns in industrial cyber-physical system designs. These models are all available for download at http://ptolemy.org.

References

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. Orlando, Florida: IEEE, 2008.

[2] A. Fisher, C. Jacobson, E. A. Lee, R. Murray, A. Sangiovanni-Vincentelli, and E. Scholte, "Industrial cyber-physical systems — iCyPhy," in *Complex Systems Design & Management (CSD&M)*. Paris, France: Springer, 2013.

[3] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, 2003.

[4] M. M. Tiller, *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[5] B. Akesson, A. Molnos, A. Hansson, J. A. Angelo, and K. Goossens, "Composability and predictability for independent application development, verification, and execution," in *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, 2011.

[6] G. Karsai, A. Lang, and S. Neema, "Design patterns for open tool integration," *Software and Systems Modeling*, vol. 4, no. 2, 2005.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European Conference in Object-Oriented Programming (ECOOP)*, vol. LNCS 1241. Finland: Springer-Verlag, 1997.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353.

[9] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and C. Sampaleanu, *Professional Java Development with the Spring Framework*. Wiley, 2005.

[10] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado, "An exploratory study of fault-proneness in evolving aspect-oriented programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 65–74.

[11] H. Yin, C. Bockisch, and M. Aksit, "A fine-grained debugger for aspect-oriented programming," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012.

[12] C. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, vol. 8, no. 3, 1977.

[13] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, no. 9, 1990.

[14] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-oriented design of embedded hardware and software systems," *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, 2003.

[15] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Berkeley, CA: Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[16] I. Akkaya, Y. Liu, and E. A. Lee, "Modeling and simulation of network aspects for distributed cyber-physical energy systems," in *Cyber Physical Systems Approach to Smart Electric Power Grid*. Springer, 2015.

[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstr, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM TECS*, vol. 7, no. 3, 2008.

[18] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? reasoning about the trends and challenges of system level design," *Proceedings of IEEE*, vol. 95, no. 3, 2007.

[19] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donze, and S. A. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, 2014.

[20] A. Wasicek, P. Derler, and E. A. Lee, "Aspect-oriented modeling of attacks in automotive cyber-physical systems," in *Proceedings of the 51st Design Automation Conference (DAC)*, June 2014.

[21] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer, "A survey on UML-based aspect-oriented design modeling," *ACM Comput. Surv.*, vol. 43, 2011.

[22] J. Liu and L. Zhang, "QoS modeling for cyber-physical systems using aspect-oriented approach," in *ICNDC '11*. Washington, DC, USA: IEEE Computer Society, 2011.

[23] M. Wehrmeister, C. Pereira, and F. Rammig, "Aspect-oriented model-driven engineering for embedded systems applied to automation systems," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 4, 2013.

[24] H. Espinoza, H. Dubois, S. Gérard, J. Medina, D. C. Petriu, and M. Woodside, "Annotating UML models with non-functional properties for quantitative analysis," in *MoDELS*, ser. Lecture Notes in Computer Science, vol. LNCS 3844. Springer-Verlag, 2005.

[25] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan, "An approach for supporting aspect-oriented domain modeling," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, ser. GPCE '03. New York, NY, USA: Springer-Verlag New York, Inc., 2003.

[26] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, no. 4, 2003.

[27] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[28] D. de Niz and P. H. Feiler, "Aspects in the industry standard AADL," in *Proceedings of the 10th international workshop on Aspect-oriented modeling*, ser. AOM '07. New York, NY, USA: ACM, 2007.

[29] S. Loukil, S. Kallel, B. Zalila, and M. Jmaiel, "AO4AADL: Aspect oriented extension for AADL," *Central European Journal of Computer Science*, vol. 3, no. 2, pp. 43–68, 2013.

[30] Y. U. Cao, A. S. Fukunaga, and A. Kahng, "Cooperative mobile robotics: Antecedents and directions," *Autonomous robots*, vol. 4, no. 1, 1997.

[31] N. Michael, S. Shen, K. Mohta, Y. Mulgaonkar, V. Kumar, K. Nagatani, Y. Okada, S. Kiribayashi, K. Otake, K. Yoshida *et al.*, "Collaborative mapping of an earthquake-damaged building via ground and aerial robots," *Journal of Field Robotics*, vol. 29, no. 5, pp. 832–841, 2012.

[32] Tokyo Electric Power Company (TEPCO). (2015, April) An unprecedented challenge: Robots obtain crucial information on conditions inside the Fukushima Reactor. [Online]. Available: http://www.tepco.co.jp/en/news/library/archive-e.html?video_uuid=o9fi533l&catid=69631

[33] ——. (2011, July) Cleanup work by using a robot in Unit 3 Reactor Building at Fukushima Daiichi Power Station. [Online]. Available: http://www.tepco.co.jp/en/news/library/archive-e.html?video_uuid=j18842aw&catid=61793

[34] M. Störzer and C. Koppen, "Pcdiff: Attacking the fragile pointcut problem, abstract," in *European Interactive Workshop on Aspects in Software*, Berlin, Germany, Sep. 2004.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[36] I. Akkaya, S. Emoto, and E. A. Lee, "PILOT: An Actor-oriented Learning and Optimization Toolkit for Robotic Swarm Applications," in *Second International Workshop on Robotic Sensor Networks, part of CPSWeek (RSN'15)*. ACM, 2015.

[37] S. Vestal, "An overview of the architecture analysis & design language (AADL) error model annex," in *AADL Workshop*, 2005.

[38] J. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "A time-centric model for cyber-physical applications," in *Proceedings of 3rd International Workshop on Model Based Architecting and Construction of Embedded System (ACESMB 2010)*, October 2010, pp. 21–35.

[39] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, 2012, in press.

[40] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Automatic synthesis of robust embedded control software." in *AAAI Spring Symposium: Embedded Reasoning*. AAAI, 2010.