Unified Graphical Co-Modelling of Cyber-Physical Systems using AADL and Simulink/Stateflow

Haolan Zhan[1,2], Qianqian Lin[1,2], Shuling Wang[1], Jean-Pierre Talpin[3⋆], Xiong Xu[1], and Naijun Zhan[1,2(✉)]

[1] State Key Lab. of Computer Science, Institute of Software, CAS, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Institut National de Recherche en Informatique et en Automatique (INRIA) Rennes, France
{zhanhl,linqq,wangsl,xux,znj}@ios.ac.cn,
jean-pierre.talpin@inria.fr

Abstract. The efficient design of safety-critical embedded systems involves, at least, the three modelling aspects common to all cyber-physical systems (CPSs): *functionalities*, *physics* and *architectures*. Existing modelling formalisms cannot provide strong support to take all of these three dimensions into account uniformly, e.g., AADL is a precise formalism for modelling architecture and prototyping hardware platforms, but it is weak for modelling physical and software behaviours and their interaction. By contrast, Simulink/Stateflow is strong for modelling physical and software behaviour and their interaction, but weak for modelling architecture and hardware platforms. To address this issue, we consider the combination of AADL and Simulink/Stateflow, two widely used graphical modelling formalisms for CPS design in industry. This combination provides a unified graphical co-modelling formalism supporting the design of CPSs from all three software, hardware and physics perspectives uniformly. This paper focuses on the required concepts to combine them, and outlines how to verify and simulate a system model defined using the combined graphical views of its constituents, by considering the case study of an Isollete System.

Keywords: AADL · Simulink/Stateflow · Co-simulation · Code generation · Analysis

1 Introduction

Cyber-physical systems (CPSs), networked embedded systems (e.g., IoT, sensor networks), exploit computing units to monitor and control physical processes via wired or wireless communication. CPSs are omnipresent, from high-speed train control systems, power and control grids, automated plants and factories, transportations, to ground, sea, air and space. Most CPSs are entrusted with mission- and safety-critical tasks. Therefore, the efficient and verified development of safe and reliable embedded systems is a priority mandated by many standards, yet a notoriously difficult and challenging domain.

⋆ Jean-Pierre Talpin is partially supported by Nankai University

As to standards, model-based design (MBD) has become a predominant development approach in the embedded system industry. In the MBD methodology, the development of a system starts with a model, based on which extensive analysis and verification are conducted, so that errors can be identified and corrected as early as possible, and ideally before the system is implemented or built. Subsequently, abstract system-level models are refined to semantically more concrete models and to source code, by model-transformation.

The merits of MBD hence include at least the following:

- Complexity becomes tractable and controllable, thanks to system level abstraction.
- Errors can be identified and corrected at the very early stages of system design.
- Correctness and reliability can be guaranteed by refinement.
- Developers can fully reuse existing components and/or systems, to improve development efficiency even more.

Unsurprisingly, available formalisms and environments for CPS design are numerous, e.g., hybrid automata [8], Hybrid CSP (HCSP) [22,44], dynamic differential logic [33], hybrid Event-B [10,11], Ptolemy [34], Metropolis [9], Crescendo [21], C2E2 [18], etc. in academia; Simulink/Stateflow [1,2], Modelica [39], SCADE [3], Labview, etc., in industry; UML, SysML [4], MARTE [38] and so on, for MBD. Because of the tight coupling of hardware, software, and physics in CPS design, one has to model a complex CPS from the perspectives of functionality (software), physicality (physical environment and hardware platform), and architecture uniformly, but unfortunately, most of existing modelling techniques do not support all of these three aspects well and uniformly.

For instance, the Architectural Analysis & Design Language (AADL) [20] is an architectural-centric model-based language developed by SAE International. It features strong capabilities to describe the architecture of a system due to the pragmatic (and practice-inspired) effectiveness of combining software and hardware component models. Meanwhile, it also supports the formal description of discrete behaviour using its BLESS Annex. Thanks to its succinct syntax, effective functionality and facilitated extensibility (by annexes i.e. plugins), AADL has been widely exploited in various embedded system domains, e.g., avionics, automotive. However, the core of the AADL only supports modelling of embedded system hardware structures and abstraction of its relevant discrete behaviour relevant to verification. It does not support the description of the continuous physical processes to be controlled by the embedded system and its combination with software, although some attempts have been made [7,6].

By contrast, Simulink [1] is the de facto standard toolbox that has demonstrated strong capabilities for model-based analysis and design of signal processing systems. It contains a large palette of functional blocks and supports their composition by continuous-time synchronous data-flow, as well as an intuitive graphical modelling language reminiscent of circuit diagrams. It is thus appealing to practitioners and engineers for whom it is designed. Moreover, Stateflow [2] is a toolbox adding facilities for modelling and simulating reactive systems by means of hierarchical statecharts, extending Simulink's scope to event-driven and hybrid forms of embedded control.

However, Simulink/Stateflow can hardly model system architectures and hardware platforms. To address this issue, we complement Simulink with AADL to provide a

unified graphical modelling formalism to support all the three perspectives of CPS design uniformly. An overview of the combination is given in Fig. 1. For each CPS system to be modelled, it will be characterized from three different layers: architecture layer, software layer and physical layer. The modelling process is sketched as follows:

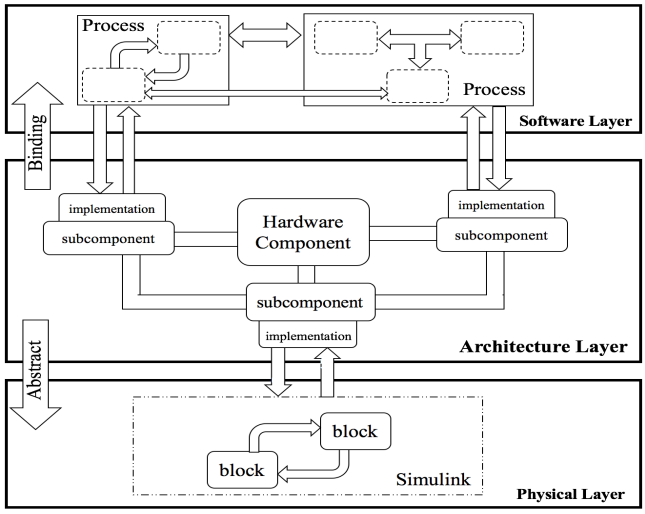


Fig. 1: An overview of the combination of AADL and Simulink

System architecture and hardware platform: are given as AADL components in the architecture layer.

Software behaviour: is modelled either as AADL components or Simulink/Stateflow diagrams in the software layer.

Physical processes and its interaction with software: are modelled as Simulink/Stateflow diagrams in the physical layer.

Type classifier for Simulink/Stateflow diagrams: are generated as AADL components in the architecture layer. Given a Simulink/Stateflow diagram, a type classifier abstracts away the implementation details, and instead, defines the port declarations and the constraints for the behavior. The AADL abstract type classifier will be combined with the other AADL components to form the whole system in the architecture layer.

First, we translate Simulink/Stateflow diagrams into HCSP to obtain a formalisation of their port declarations [47,42,46]. Second, we use Daikon [19], or invariant generation [28] or, possibly, compositional proof tactics [30] to associate type classifiers with formal contracts.

Simulation of the whole graphical model, defined by the combination of AADL and Simulink views, amounts to coordinating code generated by both AADL and Simulink model simulators through effective port communications. Verification of the combined

models is performed by translation to HCSP [27,47,41,14]. Similar to the translation from Simulink/Stateflow to HCSP and the inverse [47,46,42], the correctness of the translation can be guaranteed using Unifying Theories of Programming (UTP) [15,42].

Contribution. In this paper, we propose a unified graphical framework using AADL and Simulink/Stateflow to model, simulate and verify cyber-physical systems. This framework depicts a methodology to design and simulate CPSs in a unified graphical environment while supporting formal verification of its functional, physical and structural artifacts uniformly using HCSP. Our graphical framework consists of AADL, its BLESS Annex and Simulink/Stateflow. It is implemented by a simulation environment called **AADLSim**, which integrates a set of tools, including an automatic translator from AADL into C, and a simluation engine combining AADL and Simulink/Stateflow models. To demonstrate the above framework and tool, the case study of an Isolette System is provided.

Paper Organization. The rest of the paper is organized as follows. Sect. 2 provides an overview of AADL, Simulink/Stateflow, and the notion of design by contract. Sect. 3 presents the Isollete case study, which will be used as a running example throughout the paper. Sect. 4 depicts our combined framework composed of AADL and Simulink/Stateflow, especially how to compute the type classifiers and define the contracts for Simulink/Stateflow diagrams. Sect. 5 presents in detail how to implement the co-modelling and co-simulation in the unified framework. Sect. 6 gives the related work and Sect. 7 concludes this paper and discusses some future work.

2 Preliminaries

In this section, we first provide an overview of the AADL standard, by highlighting its structure and BLESS Annex, then introduce Simulink/Stateflow and its most relevant features. Finally, we briefly introduce the notion of design by contract.

2.1 AADL

AADL provides means to specify both the application software and the execution hardware of an embedded system, and supports textual, graphical and XML Metadata Interchange (XMI) specification formats. Components with *type* and *implementation* classifiers are instantiated and connected together to structure the system architecture. The AADL core language constructs are categorised into application software, execution platform and composite components. A *system* component represents a composite entity containing software, execution platform and system components.

Components and Connections The execution platform category represents computation and communication resources including *processor*, *memory*, *bus* and *device* components. A processor component represents the hardware and software responsible for scheduling and executing task threads. Properties can be assigned to a processor component to specify scheduling policies, high-level operating system services and communication protocols. A memory component is used to represent storage entities for data

and code. A device component can model a physical entity in the external environment: a plant or the software simulation of the plant. It can also be used as an interactive component like sensor or actuator. A bus component represents the physical connections among execution platform components.

The application software category includes *process*, *data*, *subprogram*, *thread*, and *thread group* components. A process component represents the protected address space, which is bound to a memory component. A data component can be used to abstract data type, local data or parameter of a subprogram. A subprogram models executable code that is called, with parameters, by thread and other subprograms. Thread is the only schedulable component with execution semantics to model system execution behavior. A thread represents sequential flow of the execution and the associated semantic automation describes life cycle of the thread.

A component type declaration defines interface elements and may contain *features*. Features comprise *data*, *event* and *event data* ports to transmit and receive data, control, and data/control respectively. Port communication is typed and directional. An *in* port receives data/control and an *out* port sends data/control while an *in out* port can send and receive data/control. Communication is realized through *connections* between ports, parameters and access to shared data.

BLESS Annex. The Behavior Language for Embedded System with Software (BLESS) is a standardised annex independent of the core AADL language. BLESS extends AADL with the ability of specifying behaviour of component interfaces, providing formal semantics for AADL behavioural descriptions and automatically generating verification conditions to be proven. BLESS models state machines using guards and actions to give precise specifications of discrete hardware/software behaviours. BLESS also introduces *assert* and *invariant* sections in AADL to specify assertions and predicates that behavioural models must satisfy.

We refer to AADL standard document AS5506-B [36] for further details.

2.2 Simulink/Stateflow

Simulink is an environment for model-based design of dynamical systems, and has become a de facto standard in the embedded systems industry. It provides an extensive library of pre-defined blocks for building and managing block diagrams, and also a rich set of fixed-step and variable-step solvers for simulating dynamical systems. It also provides features such as subsystems for building large systems in a hierarchical way. Stateflow is a toolbox adding facilities for modelling and simulating reactive systems by means of hierarchical statecharts. It extends Simulink scope to event-driven and hybrid forms of embedded control.

A Simulink model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by dataflow through the connecting wires. An elementary block receives input signals and computes output signals according to user-defined parameters altering its functionality. One typical parameter is sample time, which defines how frequently the computation is performed. Blocks are classified into two types: continuous blocks with sample time 0, and discrete blocks with positive sample time. For

continuous blocks, the continuous state changes over time continuously, e.g. the position or the speed of a moving car. It is usually represented by an ordinary differential equation (ODE). Simulink provides an amount of ODE solvers for solving ODEs based on the numerical integration methods.

Stateflow offers the modelling capabilities of statecharts for reactive systems. It can be defined as Simulink blocks, fed with Simulink inputs and producing Simulink outputs. A stateflow diagram is composed of transitions, states and junctions. Each transition connects a source state to a destination state. It it is labelled with $E[C]\{cAct\}/tAct$, where E is an event, C is the condition, *cAct* the condition action, and *tAct* the transition action. The event E triggers the transition to take place, provided that the condition C is true. As soon as C evaluates to true, the action $cAct$ will be executed immediately, while *tAct* will be left pending and put in a queue first, and will be executed until a valid transition path is completed. A state is labelled by three optional types of actions: *entry action*, *during action*, and *exit action*.

Stateflow supports to construct *flow charts* using connective junctions and transitions, which can be used between states to specify decision logics to form transition networks. The Stateflow states can be composed to form hierarchical diagrams: *Or diagram*, for which the states are mutually exclusive and only one state becomes active at a time, and *And diagram*, for which the states are parallel and all of them become active simultaneously.

Being based on a large palette of individually simple function blocks and their composition by continuous-time synchronous dataflow as well as the modelling capabilities of statecharts for reactive systems, Simulink/Stateflow offers an intuitive graphical modelling language of CPSs for practicing engineers. Ordinary users can quickly build the model's framework by connecting the corresponding graphical modules and defining interfaces. Therefore, it is convenient and efficient to design and analyse the components using Simulink/Stateflow for co-simulation.

2.3 Design by Contract

Design by contract (DbC) is an engineering methodology whereby system designers should define semantically founded, precise and verifiable interface specifications for hardware and software components. These specifications extend the ordinary notion of abstract data type with logical properties describing the pre-conditions, post-conditions and invariants of a software function or of a hardware block.

The term design-by-contract is due to Bertrand Meyer in connection with the definition of the Eiffel programming language and his book Object-Oriented Software Construction [31]. It is rooted in Hoare logic, where the contract (A,G) of a program P naturally corresponds to the provable assertion $C \vdash \{A\}P\{G\}$ in some logical context C. Contracts have been algebraically meta-theorized by Benveniste et al. [13], systematically applied to model-based design frameworks like BIP [12].

Recently, [30] extended the reach of design-by-contract to the case of modularly verifying hybrid system models by the introduction of contracts in a compositional design methodology for Differential dynamical Logic (ddL) [33]. In this context, and by contrast, the contract of a given model $\Gamma \vdash [\alpha]\phi$ consists of the evolution domain H of the specification α, as assumption, and differential invariant ϕ, as guarantee.

3 Isollete System: A Running Example

In this section, we introduce the *Isolette* System which we use as a running example. Isolette is an infant incubator described by the Federal Aviation Administration (FAA) in the Requirement Engineering Management Handbook (REMH) [26]. This example is concise but rich enough to contain both discrete control behaviour and continuous plants, as a classical hybrid system [7]. We will first introduce the system and then the design requirements.

3.1 Isollete System

The isollete example has been widely used to explain the detailed behaviour of AADL-based development and new annexes, such as the BLESS Annex and the Error Model Annex [17]. The isollete system is used to maintain the temperature of the isollete box, a physical environment, within a desired range that is beneficial to an infant.

Fig. 3 depicts the AADL graphical model of the isollete system. The architecture of the system includes a processor, a bus, a sensor, an actuator, a controller, and a controlled process with internal threads. The software level defines the implementation of the controller, which obtains the temperature inside the box through the sensor, then computes an appropriate command to control the temperature through the actuator to switch on or off the heater combined with the isollete box. The physical layer defines the continuous behavior of the plant, i.e. the isollete box.

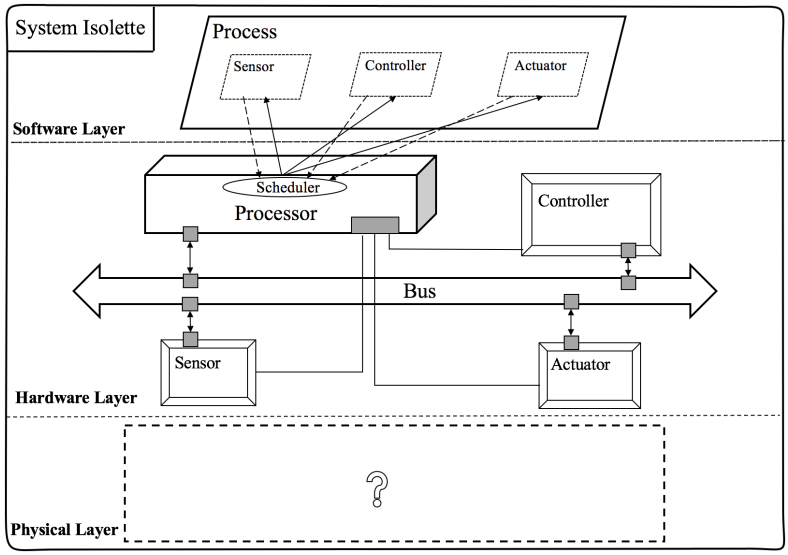


Fig. 2: AADL graphical model of Isolette system

The continuous evolution of temperature depends on the current status of the actuator. If the heater is **on**, the temperature will increase, otherwise decrease. According to

the specification in the Section A.5.1.3 of the REMH [26], when the isolette is properly switched on, the temperature of the heater will change at a rate of no more than 1 °F *per minute*. Based on this specification, the temperature of the isollete box (denoted by c) and the temperature of the heater (denoted by q) are formally modelled by the ODEs (1) below.

$$\begin{cases} \dot{c} &= -0.026 \cdot (c - q) \\ \dot{q} &= 1 \qquad \text{if heater is on} \\ \dot{q} &= -1 \qquad \text{if heater is off} \end{cases} \tag{1}$$

The constant 0.026 stands for the thermal conductivity. When the controller commands the actuator to switch the heater on, the rise in temperature q will result in c going up. In this specification, we assume that the room temperature outside the box to be constant at 73 °F, although its variations could also be modelled.

3.2 Requirements

Referring to environmental assumptions provided in the REMH, the following safety should be satisfied.

– **Safety:** The temperature inside the isollete box should be kept in between 97 °F and 100 °F, i.e., 97 °F $\leq c \leq$ 100 °F.

Moreover, considering the uncertainties from initial states, sensor errors, disturbance of dynamics, and numerical error caused by floating-point calculation, etc., it is required that:

– **Stability and Robustness:** The inside temperature c will be finally steered towards the valid range after some time.

At this point, it is obviously hard to specify this physical model using AADL and its annexes alone, notwithstanding its interaction with the digital controller, hence the question mark in Fig. 2 needs a complementary hybrid annex.

4 Combination of AADL and Simulink/Stateflow

The combination of AADL and Simulink/Stateflow aims at providing a unified graphical co-modelling formalism for CPSs, with which software, physical environment and execution hardware of a CPS can be modelled in a uniform framework. Sect. 4.1 presents a general explanation of the combined framework, Sect. 4.2 and Sect. 4.3 define the type classifiers for given Simulink/Stateflow diagrams, including the port declarations and contracts, respectively.

4.1 General Framework

As shown in Fig. 1, we describe the high-level architecture of the proposed unified graphical framework together with the connection among the three different physical,

hardware and software layers. The architecture layer, described as AADL system composite components, specifies the types of hardware and software components, and (part of) their implementation (an abstraction of their actual implementation), as well as their composition. It usually consists of a central processor unit classifier with several subcomponent devices (like sensor, controller, and actuator etc.). Each of these classifiers has its own type and implementation. For software functionality and physical processes, the architecture layer usually needs their *abstractions*, i.e., the *type classifiers* of these software and physical components. The type classifier of a component declares the set of input and output ports, specifies the contract of its behaviour, that are accessible from outside. By contrast, the implementation classifier of a component binds its type classifier with a concrete implementation in the software and physical layers.

Our framework provides two methods to describe the type classifier of a given Simulink/Stateflow model. The first one is to derive a type classifier, which is satisfied by the Simulink/Stateflow diagram, directly from its behaviour; see Sect. 4.2; the other is to define a contract in the style of an assume/guarantee pair, and then prove the given Simulink/Stateflow diagram satisfying this contract; see Sect. 4.3.

In the software layer, software components are defined by their functionality, which can be done using either AADL or Simulink/Stateflow. In AADL, the functionality is defined by processes, and in each process, one or more threads may exist to describe specific controlling behaviours. The BLESS Annex can further be employed to specify the behavior of the system precisely. In order to establish a stable communication between different processes, a port declaration must be defined. The AADL implementation in this layer binds to the corresponding software and hardware components in the architecture layer.

In the physical layer, the continuous behaviour of physical processes is implemented as Simulink/Stateflow diagrams. In order to integrate the Simulink/Stateflow diagrams for implementing software or physical processes into the architectural layer, we need to define a type classifier for each Simulink/Stateflow diagram so that it can be assembled with other abstract components to form the architecture of the whole system at the architecture layer. We will explain the details of this process in the rest of this section.

Example 1. Now we can build a complete graphical model of the Isollete system with the combination as shown in Fig. 3, in which the Simulink/Stateflow diagram is given as Fig. 4.

Fig. 4 implements the ODEs defined in (1). It receives the heat command from the actuator, depending on which the heater temperature q is implemented by an integrator block. The other integrator block computes the temperature c for the isollete box, which will be sent back to the sensor of the controller. This implementation will be abstracted as a AADL type classifier, to fill the definition of the isollete box in the physical layer in Fig. 3.

Simulation. To simulate the graphical model with the combination of AADL and Simulink, we propose a cross-layer co-simulation framework, in which the hardware platform, control software, and physical dynamics in the designed CPS can be taken into account uniformly. We will explain the details of such specification in Sect. 5.

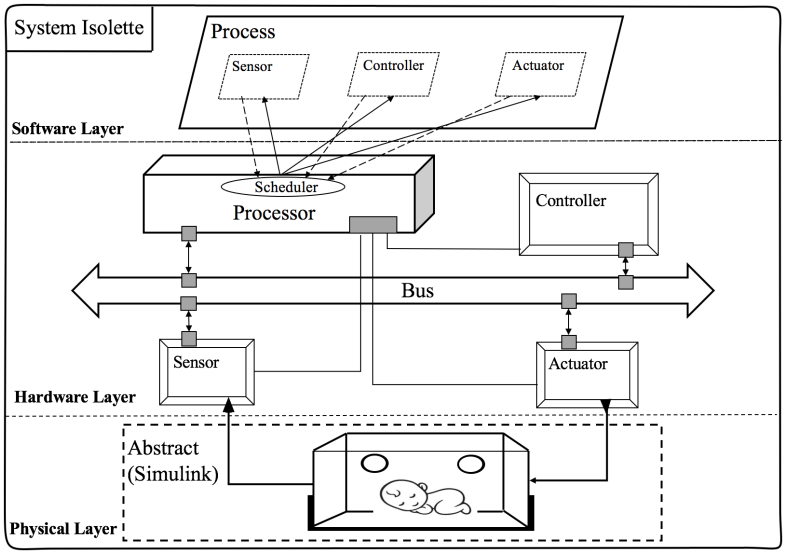


Fig. 3: AADL graphical model of Isolette system

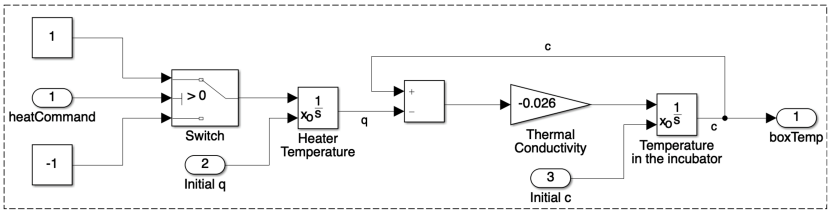


Fig. 4: Simulink model of Isollete box

Verification. To further verify a graphical model given by the AADL-Simulink combination, we translate it into HCSP, which is an extension of CSP introducing differential equations to model the continuous evolution of the plant and three types of interrupts to model the interaction between continuous and discrete behaviours [22,44]. The formal verification of HCSP can be done along the lines of our previous work [27,45,41,14]. Moreover, the correctness of the translation from Simulink/Stateflow to HCSP can be strictly proved using higher-order UTP [15], which extends the classic Unifying Theories of Programming (UTP) [23] to hybrid systems by introducing higher-order quantifications and differential relations. The technical details of this part will be reported in another paper.

4.2 Computing Type Classifier for Simulink/Stateflow Diagrams

As we explained above, when combining Simulink/Stateflow with AADL, we need to provide an abstraction for each Simulink/Stateflow diagram, i.e., its type classifier,

so that it can be assembled with other components to form the whole system at the architecture layer, while the diagram itself will be used as the implementation classifier of the component. Normally, the type classifier of a component consists of two parts: *port declaration* and *constraints*.

The *port declaration* declares a set of ports used to input and output data between the component and other ones. However, Simulink diagrams can be hierarchical, and hence its external ports can sometimes not be extracted directly. For example, consider the triggered subsystems in a Simulink diagram, they do not have any input and output ports, but are triggered by events. Therefore, we need to analyse the whole system in detail in order to obtain all external ports, particular, event ports. Moreover, this often gets worse when Stateflow models are additionally considered.

To address this problem, we exploit the tool `Sim2HCSP`, a component in our toolkit MARS [14], which can translate a Simulink/Stateflow diagram into a formal HCSP process. By applying `Sim2HCSP`, all external ports of a Simulink/Stateflow diagram can now be translated, and exposed, by a set of channels in the corresponding HCSP model, which is stored in a separate file. In the case of the Simulink diagram in Fig. 4, we can for instance obtain the following port declaration:

$$\texttt{heatCommand}?q; \cdots ; \texttt{boxTemp}!c$$

from which the abstract type for `babybox` can be defined correspondingly:

```
abstract babybox
features
  heatCommand: in data port;
  boxTemp: out data port;
end babybox;
```

The reminder of the specification defines the contract of the component. It specifies the properties that should be satisfied by any execution of the component. In this paper, we adopt two approaches to generate the constraints for a given Simulink/Stateflow diagram. The first one uses Daikon [19]. The basic idea is to simulate the given Simulink/Stateflow diagram, and then run Daikon to generate a candidate invariant which is satisfied by all simulation runs. The more simulations are performed the more refined the generated invariant becomes. For example, considering the Simulink diagram in Fig. 4, by applying Daikon, we can obtain the following type classifier:

```
assert
  <<TIME: :(t >= 0.1)>>
  <<HEAT_T: :
      ((1.35107*10**15)*t-(1.35107*10**15)*q+9.862881*10**16=0)>>
  <<H_VAR: :
      ((2.111*10**13)*q-(2.111*10**13)*orig(q)+1.056*10**12=0)>>
  <<TEMP_VAR: :
      ((2.463*10**11)*c-(2.744*10**11)*orig(c)+2.055*10**12=0)>>
invariant
  <<TIME() and H_VAR() and H_VAR() and TEMP_VAR()>>
```

Alternatively, we can generate invariants directly from the Simulink/Stateflow diagram, or the translated HCSP process, by using techniques for invariant generation for hybrid systems, e.g., [28]. Consider again the Simulink diagram in Fig. 4. By using invariant generation, we can instead obtain the following type classifier:

```
assert
  <<L_LIMIT: : ((q-c)*e**(-0.026*t)+q*(c-97)<=0)>>
  <<H_LIMIT: : ((q-c)*e**(-0.026*t)+q*(100-c)>=0)>>
invariant
  <<L_LIMIT() and H_LIMIT()>>
```

The efficiency of the first approach is much higher, but the generated invariant (approximation) can only be linear. Moreover, it may not become an actual invariant, even by conducting enough runs to refine it. By contrast, the second approach can generate more expressive and semantically correct invariants, but the efficiency is normally very low. Improving the efficiency of invariant generation for hybrid systems is still a challenging problem.

4.3 Defining Type Classifier as Contracts

Our goal is to exploit the HCSP model provided by `Sim2HCSP` [14] to support modular, component-wise analysis and verification of system models combined from architectures described in AADL and hybrid systems in Simulink/Stateflow.

For HCSP models, our definition of contracts will naturally follow along the lines proposed by Lunel et al. for differential dynamic logic [30,29]. In that context, the contract of a specification α is defined by a pair (A, G) of properties. A, the assumption, is a formula defining the evolution domain of α and G, the guarantee, is a formula stating its differential invariant.

An alternative approach is to use the Hybrid Hoare Logic of HCSP [27,40]. In the HHL, the contract of an HCSP process P can be defined by the term $\{Pre\}\ P\ \{Post; HF\}$, where *Pre*, *Post* represent the pre- and post-conditions of P using first-order logic and *HF* its history formula using the duration calculus. This not only allows to express properties upon start and finish but also real-time and continuous invariants on the execution of P, resulting in an undoubtedly more expressive framework, however probably challenging for proof automation.

In either approaches, it is hence tempting to investigate an adaptation of the composition theorem proposed in [30] to the HCSP framework, as it provides a methodology to automate the proof of a system contract, e.g. $(A_1 \wedge A_2, G_1 \wedge G_2)$, from the (possibly tedious) proofs that its components, e.g. $C_{1,2}$ satisfy the differential invariant $G_{1,2}$ in the evolution domains $A_{1,2}$, respectively.

This theorem is obtained using the parallel composition defined in [30, Def. 7], which amounts to decomposing the components $C_i \mathrel{\widehat{=}} \mathbf{disc}_i \cup \mathbf{cont}_i^*$ into discrete and continuous specifications $\mathbf{disc}_i$ and $\mathbf{cont}_i$, and recompose them as:

$$C_1 \otimes C_2 \mathrel{\widehat{=}} (\mathbf{disc}_1 \cup \mathbf{disc}_2 \cup (\mathbf{cont}_1, \mathbf{cont}_2)^*)$$

Assuming proof trees $\Gamma_i \vdash [C_i]G_i$, stating that the G_is are invariants of the components C_is in contexts Γ_i, for all $i = 1, 2$, and assuming non-interference of the definitions

between the C_is nor with the guarantees of the G_js ($i \neq j$), [30, Th. 2] exhibits the derivation of a contract for the composed components: $\Gamma_{i=1,2} \vdash [\otimes_{i=1,2} C_i](\wedge_{i=1,2} G_i)$, yielding an automated proof tactic.

This model of compositional contracts can be employed to implement Sangiovanni-Vincentelli's "meet in the middle" design methodology [37] to mitigate software, hardware and physics constraints at system architecture level. In the case of the isolette, for instance, it can be used to verify the safety requirement of the isollette in nominal mode, Sec. 3.2 (i.e. after an initialization period) by cross-validating the differential invariant of the physical model with the (adequate) operations of its controller on the sensors and actuators, all four expressed by separate logical contracts. A use case of this method with KeymaeraX, concerning the well-known controlled water-tank problem, is given in Lunel's PhD Thesis [29].

5 Co-Modelling and Co-Simulation

This section details the implementation of the unified framework introduced in Subsec. 4.1 for designing and analyzing CPSs. The design flow of the framework is shown in Fig. 5. It consists of three stages: co-modelling, model translation, and co-simulation.

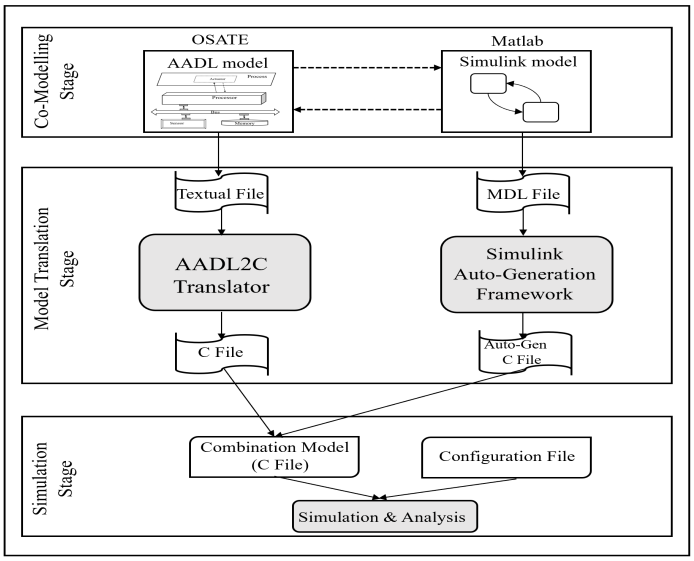


Fig. 5: Co-modelling and co-simulation of AADL and Simulink/Stateflow

In the co-modelling stage, designers can exploit the toolkit OSATE/AADL and Matlab/Simulink/Stateflow to model different parts of systems. Port definitions are required in each of the separate parts in order to establish the connection between AADL and Simulink/Stateflow models. Then, in the model translation stage, in order to integrate the two separate models and analyse them as a whole, we translate both the AADL

and Simulink/Stateflow models into C code. For AADL, we developed a toolkit named *AADL2C Translator*, a novel code generation plugin to parse the AADL standard textual file and translate it into C code. Auto generation of C code from Matlab/Simulink/Stateflow models is done directly by using the Real Time Workshop (RTW) toolbox of Matlab. In the co-simulation stage, the translated C code from AADL and Simulink will be combined at first by performing integration and parameter configuration. After that, the generated model code will be compiled by a C compiler, with the co-simulation results produced. The co-simulation results provide a feedback for engineers to analyse and revise their original designs in the modelling stage. All three stages are introduced in detail in the subsequent subsections.

5.1 Co-Modelling in AADL and Simulink/Stateflow

AADL modelling The OSATE platform provides two different approaches for engineers to build AADL models: graphical models and textual code. To exploit the internal mechanism of AADL, we choose the textual form to build our system. With the BLESS Annex, AADL is also able to specify the discrete behaviour of components. AADL adopts a top-down pattern to build a system: a *system* classifier is defined at the beginning, and then all its hardware and software subcomponents are declared in *system implementation*. For the Isollete example, the type classifier and implementation for the whole system shown in Fig .3 are given as follows:

```
system isollete
end isollete;
system implementation isollete.impl
  subcomponents
    heatCPU: processor heatCPU;
    heatSW: process heatSW.impl;
    babybox: abstract babybox.impl;
  connections
    cnx1: port heatSW.heatCommand -> babybox.heatCommand;
    cnx2: port babybox.boxTemp -> heatSW.boxTemp;
  properties
    ......
end isollete.impl;
```

The `heatCPU` element defines the central processor. The `heatSW` element, the central process for specifying the functionality of the sensor, the actuator and the controller. The `babybox` stands for the isollete box. In the connections section, the ports of `heatSW` and the ports of `babybox` are connected, for transferring the heat command (representing the off or on status of the controlled variable) and the box temperature respectively. The properties section stipulates a binding relationship between the software and hardware subcomponents. We omit the details of it here. The behaviours of the sensor, actuator and the controller are implemented as threads in AADL.

In particular, the model of the controller is defined as follows:

```
thread controller
  features
    measuredTemp: in data port;
    diff: out data port;
end controller;
thread implementation controller.impl
  properties
    Dispatch_Protocol => Periodic;
    Priority => 10;
    Deadline => 20ms;
    Period => 20ms;
  annex BLESS {**
    invariant <<true>>
    variables: gain ;
    states s : initial complete final state;
    transition t : s -[on dispatch]-> s
    { gain := 10;
      if(measuredTemp > 100) diff := gain*(measuredTemp - 100);
      elsif(measuredTemp < 97) diff :=gain*(measuredTemp - 97);
      else diff :=0; end if; };
  **};
end controller.impl;
```

The controller receives the measured temperature of the isollete box from the sensor via the input port `measuredTemp`, and sends the difference between the temperature and the threshold to the actuator via output port `diff`. As defined in the implementation, the controller is executed periodically every 20ms, with the deadline and priority defined; Its functionality is defined using the BLESS Annex. The local variable `gain` defines the gain coefficient for computing the difference, and the transition system of the controller includes one state and one transition, which computes the difference `diff` depending on the measured temperature. After receiving the value of `diff`, the actuator will decide whether to turn on or off the heat.

Simulink/Stateflow modelling We use Simulink/Stateflow to model the continuous behaviour of the CPS under design. For the Isollete box, we need to model the continuous behaviour defined by the ODE (1). The Simulink diagram has been given in Fig. 4.

Combination of models After building the models separately in AADL and Simulink/Stateflow, we combine them to form the whole system. Our approach is to define *abstract* components in AADL and connect each of them to the corresponding Simulink/Stateflow models. For each abstract component, the type classifier declares all the ports connecting AADL and Simulink/Stateflow models, and the constraints for the actual behaviour. The abstract AADL type classifier of the isollete box is given in Sec. 4.2.

5.2 Model Translation to C

Translating the AADL model The translation from AADL to C is the most crucial part in the unified framework. It uses a collection of mapping rules from AADL concepts to C implemented by the compiler *AADL2C Translator*. Fig. 6 illustrates the model translation flow from AADL to C. A graphical model only describes the high-level architecture, while a textual model includes the details such as the functional behaviours.

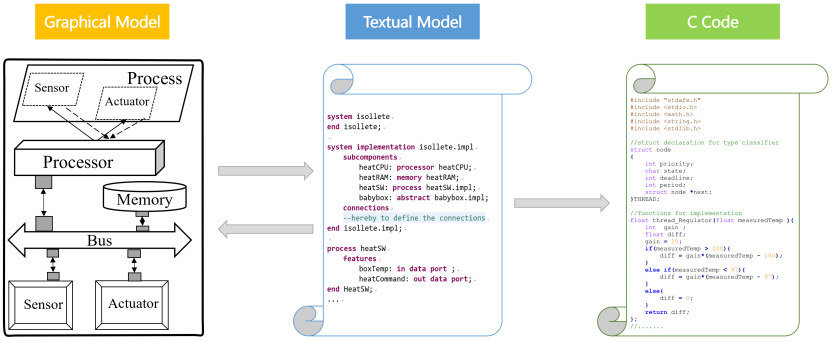


Fig. 6: An example illustrating the model translation flow from AADL to C

The compiler *AADL2C Translator* receives a source file as input and automatically generates the corresponding C code. According to the AADL grammar, a model is usually composed of several components, each with two parts: type declaration and implementation. The *AADL2C Translator* creates a `struct` object for each type declaration, e.g..system, process, thread, etc., and defines a collection of properties of the corresponding type classifiers. The implementation classifier is translated into individual sub-functions, associated with relevant type classifiers and specific names. Especially, for a thread implementation, two extra functions are specified for thread scheduling: *create_thread()* and *thread_scheduling()*. The *create_thread()* function adds a thread to a thread queue for dispatch, while *thread_scheduling()* is designed to execute threads according to the scheduling protocol specified in the AADL model, e.g., Rate-Monotonic Scheduling (RMS), Highest Priority First (HPF), etc.

In order to implement the port communications between different components efficiently, an additional Global Port Data Management (GPDM) unit is introduced in the target C code. The GPDM will store all of the output ports as global variables, with names of the form *componentType_componentName_outputPortName*. In each simulation cycle, the values of these variables will be updated once.

Translating Simulink/Stateflow model Matlab provides an automatic code generation tool that helps to translate Simulink/Stateflow models into C code. It greatly improves the quality and efficiency of development and simulation. To apply the code generation tool provided by Matlab, we need to set some configuration parameters, such as the model solver, the format of the generated code, etc.

Co-simulation requests a starting point of program execution so that we specify a main method as an interface to connect the C code generated from Simulink/Stateflow model with the C code interface generated from AADL model.

5.3 Co-simulation

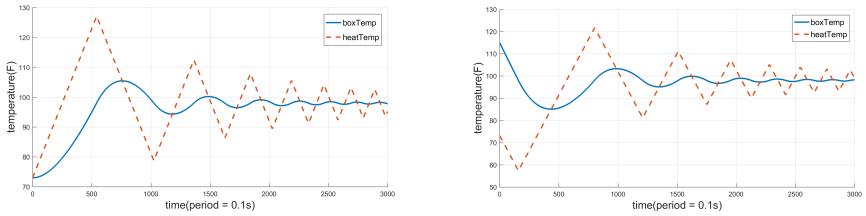
Now all the different parts of the system, including hardware components, application softwares and physical processes modelled in AADL and Simulink/Stateflow, have been translated into C code separately. In the co-simulation stage, we need to integrated all the separate C code files by defining the communication between them. The communication of distributed parts is implemented through the GPDM block mentioned above, which can be regarded as a global data memory storing all the data interfaces (external in/out ports) information of each component, such as AADL thread components and Simulink/Stateflow models. Local variables inside components are not considered by the GPDM block.

After the C code files are integrated, the simulation of the whole system can be performed. At the beginning of the simulation, we need to set some configuration parameters, including the global simulation clock, the periodical simulation clock, the initial values of the system variables, and so on.

Simulation results of Isollete For the Isollete case study, we check whether it fulfils the requirements mentioned in Sec. 3.2 by simulation. We first translate the AADL and Simulink model of the Isollete to C code, then consider two cases by setting different initial values for the variables. In the first case, both the temperature inside isollete box and heat actuator are initially set as 73 °F, same as the general room temperature. In the second case, the initial temperature inside the isollete box is set as 115 °F (higher than the maximum safety temperature), and the temperature for the heat actuator is still set as 73 °F. For both of them, the simulation period is set to 0.1 s, and the simulation time is set to 300s. Fig. 7(a) and Fig. 7(b) show the simulation results for the two cases respectively, where the blue solid curve and yellow dashed curve represent the trajectories for continuous variables c (for box temperature) and q (for heat temperature) respectively. The simulation results show that, under the control of heat actuator, the temperature inside the isollete box will finally reach a stable state, within the safety range between 97 °F and 100 °F. The requirement defined in Sect. 3.2 is satisfied.

6 Related Work

AADL provides the notion of annex to support extensions to its core language. The key standardized annexes include the Behaviour Annex (BA) which extends AADL with the ability of defining component behaviour via state machines, and the BLESS Annex[25], which improves the state transition formalism by introducing assertions for supporting contract-based specifications. The simulation and analysis of AADL models have also been explored a great deal. ADeS is a simulation tool that considers the environment in which the system evolves. AADL Inspector, produced by the Ellidiss company, is a powerful software that encompasses various features including schedulability



(a) Initial stage: c=73 °F, q=73 °F (b) Initial stage: c=115 °F, q=73 °F

Fig. 7: Results of co-simulation from different inital stages

analysis and dynamic simulation. There have also been some works on translation of AADL to other languages for analysis and simulation, e.g. AADL to BIP [16], AADL to Sync [24], AADL to Maude [32] and so on. However, most of them focus on the discrete-time behaviours.

There have been some works on the extension of AADL for hybrid systems. [43] models hybrid systems with AADL based on networks of timed automata, and uses the model checker UPPAAL for property analysis. [35] discusses a sublanguage extension to AADL to describe continuous behaviour, but it has difficulty in modelling complex continuous behaviour expressed with differential equations. In [5], a Hybrid Annex is presented, which is much more expressive in its ability to specify hybrid systems, yet it lacks relevant tools for further simulation and analysis of the hybrid models.

Compared with the above mentioned works, our proposed AADLSim framework coalesces AADL with Simulink for modelling of both discrete and continuous behaviours, and the flexible interaction between them. Moreover, it also provides extensive support for the analysis and simulation of the combined models through translating them into the same target language.

7 Conclusion and Future work

In this paper, we propose an unified graphical co-modelling and co-simulation framework for the design of cyber-physical systems. This proposed framework combines AADL and Simulink/Stateflow with which the gap between discrete control and continuous plant can be filled. The combined models are translated into C code for further analysis by co-simulation. Throughout the paper, we clarify the main concepts of the framework, outline the specific co-simulation flow and the verification process of the combined models. An Isollete system case study is provided to illustrate the framework.

For future work, we will investigate the translation of the combination into HCSP, a formal modelling language encoding hybrid system dynamics by means of an extension of CSP. Formal verification of HCSP is supported by an interactive Hybrid Hoare Logic prover based on Isabelle/HOL. As a consequence, the combined AADL and Simulink/Stateflow models can be verified. To make sure that the generated HCSP

model is correct, the consistency between observational behaviours of the models at AADL and Simulink/Stateflow, and HCSP must be guaranteed in a rigorous way. This question, however, is known to be difficult, due to the inherent complexity of hybrid systems. To solve this problem, we consider to define the semantics of all the separate models in higher-order UTP [15], which extends the classic Unifying Theories of Programming (UTP) [23] to hybrid systems by introducing higher-order quantifications and differential relations. Moreover, we need to show that the domain of hybrid designs proposed in [15] together with the operations over hybrid designs forms a complete partial order (CPO), therefore can be used as a semantic domain.

References

1. *Simulink User's Guide*, 2013. `http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf`.
2. *Stateflow User's Guide*, 2013. `http://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf`.
3. *Esterel Technologies, SCADE suite*, 2018. `http://www.esterel-technologies.com/products/scade`.
4. *SysML 1.6 Beta Specification*, 2019. `http://www.omg.org/spec/SysML`.
5. E. Ahmad, Y. Dong, B. Larson, J. Lü, T. Tang, and N. Zhan. Behavior modeling and verification of movement authority scenario of chinese train control system using aadl. *Science China Information Sciences*, 58(11):1–20, 2015.
6. E. Ahmad, Y. Dong, S. Wang, N. Zhan, and L. Zou. Adding formal meanings to AADL with hybrid annex. In *International Conference on Formal Aspects of Component Software*, pages 228–247. Springer, 2014.
7. E. Ahmad, B. R. Larson, S. C. Barrett, N. Zhan, and Y. Dong. Hybrid annex: An AADL extension for continuous behavior and cyber-physical interaction modeling. In *ACM SIGAda Ada Letters*, volume 34, pages 29–38, 2014.
8. R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer, 1993.
9. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.
10. R. Banach, M. Butler, S. Qin, N. Verma, and H. Zhu. Core Hybrid Event-B I: Single Hybrid Event-B machines. *Science of Computer Programming*, 105:92–123, 2015.
11. R. Banach, M. Butler, S. Qin, and H. Zhu. Core hybrid event-b ii: Multiple cooperating hybrid event-b machines. *Science of Computer Programming*, 139:1–35, 2016.
12. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.
13. A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J. Raclet, P. Reinkemeier, A. L. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen. Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3):124–400, 2018.
14. M. Chen, X. Han, T. Tang, S. Wang, M. Yang, N. Zhan, H. Zhao, and L. Zou. MARS: A toolchain for modelling, analysis and verification of hybrid systems. In *Provably Correct Systems*, pages 39–58. Springer, 2017.

15. M. Chen, A. P. Ravn, S. Wang, M. Yang, and N. Zhan. A two-way path between formal and informal design of embedded systems. In *UTP'16*, volume 10134 of *LNCS*, pages 65–92. Springer, 2016.
16. M. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - application to the verification of real-time systems. In *MODELS'08*, volume 5421 of *LNCS*, pages 5–19. Springer, 2008.
17. J. Delange and P. Feiler. Architecture fault modeling with the AADL error-model annex. In *40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 361–368. IEEE, 2014.
18. P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok. C2E2: A verification tool for annotated Stateflow models. In *TACAS 2015*, volume 9035 of *LNCS*, pages 68–82, 2015.
19. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.
20. P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.
21. J. Fitzgerald, P. G. Larsen, and M. Verhoef, editors. *Collaborative Design for Embedded Systems: Co-modelling and Co-simulation*. Springer, 2014.
22. J. He. From CSP to hybrid systems. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 171–189. Prentice Hall International (UK) Ltd., 1994.
23. C. A. R. Hoare and J. He. *Unifying theories of programming*. Prentice Hall, 1998.
24. E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *EMSOFT'07*, pages 134–143. ACM, 2007.
25. B. R. Larson, P. Chalin, and J. Hatcliff. BLESS: Formal specification and verification of behaviors for embedded systems with software. In *NFM'13*, volume 7871 of *LNCS*, pages 276–290. Springer, 2013.
26. D. L. Lempia and S. P. Miller. Requirements engineering management handbook. *National Technical Information Service (NTIS)*, 2009.
27. J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for hybrid CSP. In *APLAS'10*, volume 6461 of *LNCS*, pages 1–15. Springer, 2010.
28. J. Liu, N. Zhan, and H. Zhao. Computing semi-algebraic invariants for polynomial dynamical systems. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 97–106. ACM, 2011.
29. S. Lunel. *Parallelism and modular proof in differential dynamic logic. (Parallélisme et preuve modulaire en logique dynamique différentielle)*. PhD thesis, University of Rennes 1, France, 2019.
30. S. Lunel, B. Boyer, and J. Talpin. Compositional proofs in differential dynamic logic dL. In *ACSD'17*, pages 19–28, 2017.
31. B. Meyer. *Object-oriented Software Construction (2Nd Ed.)*. Prentice-Hall, Inc., 1997.
32. P. Ölveczky, A. Boronat, and J. Meseguer. Formal semantics and analysis of behavioral AADL models in real-time Maude. In *Formal Techniques for Distributed Systems*, volume 6117 of *LNCS*, pages 47–62. Springer, 2010.
33. A. Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.
34. C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
35. Y. Qian, J. Liu, and X. Chen. Hybrid AADL: A sublanguage extension to AADL. In *Internetware '13*. ACM, 2013.
36. SAE International Standards. Aarchitecture analysis & design language (AADL), Revision B. 2012.

37. A. Sangiovanni-Vincentelli. Quo vadis, sdl: Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, 95(3), 2007.
38. B. Selic and S. Gerard. *Modeling and Analysis or Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. The MK/OMG Press, 2013.
39. M. Tiller. *Introduction to Physical Modeling with Modelica*. The Springer International Series in Engineering and Computer Science. Springer, 2001.
40. S. Wang, N. Zhan, and D. Guelev. An assume/guarantee based compositional calculus for hybrid CSP. In *TAMC'12*, volume 7287 of *LNCS*, pages 72–83. Springer, 2012.
41. S. Wang, N. Zhan, and L. Zou. An improved HHL prover: an interactive theorem prover for hybrid systems. In *ICFEM'15*, volume 9407 of *LNCS*, pages 382–399. Springer, 2015.
42. N. Zhan, S. Wang, and H. Zhao. *Formal Verification of Simulink/Stateflow Diagrams*. Springer, 2017.
43. Y. Zhang, Y. Dong, F. Zhang, and Y. Zhang. Research on modeling and analysis of CPS. In *International Conference on Autonomic and Trusted Computing*, pages 92–105. Springer, 2011.
44. C. Zhou, J. Wang, and A. P. Ravn. A formal description of hybrid systems. In *Hybrid systems*, volume 1066 of *LNCS*, pages 511–530, 1996.
45. L. Zou, J. Lv, S. Wang, N. Zhan, T. Tang, L. Yuan, and Y. Liu. Verifying chinese train control system under a combined scenario by theorem proving. In *VSTTE'13*, volume 8164 of *LNCS*, pages 262–280, 2013.
46. L. Zou, N. Zhan, S. Wang, and M. Fränzle. Formal verification of Simulink/Stateflow diagrams. In *ATVA'15*, volume 9364 of *LNCS*, pages 464–481. Springer, 2015.
47. L. Zou, N. Zhan, S. Wang, M. Fränzle, and S. Qin. Verifying Simulink diagrams via a hybrid Hoare logic prover. In *EMSOFT'13*, pages 1–9. IEEE, 2013.