20

An End-to-End Framework for Designing Networked Control Systems

Alie El-Din Mady and Gregory Provan
Cork Complex Systems Lab (CCSL), Computer Science Department
University College Cork (UCC), Cork
Ireland

1. Introduction

Designing a control system over Wireless Sensor/Actuator Network (WSAN) devices increases the coupling of many aspects, and the need for a sound discipline for writing/designing embedded software becomes more apparent. Such a WSAN-based control architecture is called a Networked Control System (NCS). At present, many frameworks support some steps of the NCS design flow, however there is no end-to-end solution that considers the tight integration of hardware, software and physical environment. This chapter aims to develop a fully integrated end-to-end framework for designing an NCS, from system modelling to embedded control-code generation. This framework aims to generate embedded control code that preserves the modelled system properties, and observes the hardware/software constraints of the targeted platform.

Existing approaches for control code design typically ignore the embedded system constraints, leading to a number of potential problems. For example, network delays and packet losses can compromise the quality of control that is achievable Mady & Provan (2011). Designing embedded control that accounts for embedded system constraints requires dealing with heterogeneous components that contain hardware, software and physical environments. These components are so tightly integrated that it is impossible to identify whether behavioural attributes are the result of computations, physical laws, or both working together.

Contemporary embedded control systems are modelled using hybrid system Henzinger (1996) that captures continuous aspects (e.g. physical environment) and discrete-event behaviour (e.g. control decision). Even though many tools support model-based code generation (e.g. Simulink), the emphasis has been performance-related optimizations, and many issues relevant to correctness are not satisfactorily addressed, including: (a) the precise relationship between the model and the generated code is rarely specified or formalized; (b) the generated code targets a specific embedded platform and cannot be generalized to multi-targeted platforms, moreover there is no generator considers the embedded WSAN; (c) the generated code does not respect the targeted platform hardware/software constraints; (d) the continuous blocks are either ignored, or discredited before code generation. Therefore, the correspondence between the model and the embedded code is lost.

This chapter proposes a framework for designing embedded control that explicitly accounts for embedded system constraints. We develop an end-to-end framework for designing an NCS where we model the system using a hybrid systems language. We focus on adopting a distributed control strategy that explicitly considers hardware/software constraints. Our approach enables us to generate code for multiple embedded platforms.

Fig. 1 shows an overview of the framework. The framework consists of four design stages, outlining the role for each stage as follows: (i) the reference model captures the control/diagnosis[1] strategies and the physical plant using a hybrid system; (ii) the control model is a projection of the source model, in which the physical/continuous aspects are abstracted/transformed to a generic control specification; (iii) the embedded control model combines the control model and the hardware/software constraints to define an embedded control model satisfying the platform constraints; (iv) the target embedded model, that captures the embedded platform code, is generated from the control model considering the hardware/software constraints.

Consequently, the reference and control models abstract the hardware/software constrains effect on the control algorithm. These constrains can be classified to: (a) processing resources constrains; and (b) memory space constrains. The processing resources constrains consider the hardware/software factors that affect the algorithm execution time (e.g., CPU speed), which can lead to incompatibility between the processing capacity and the algorithm execution. Whereas, the memory space constrains check if there is enough memory space for executing the control algorithm.

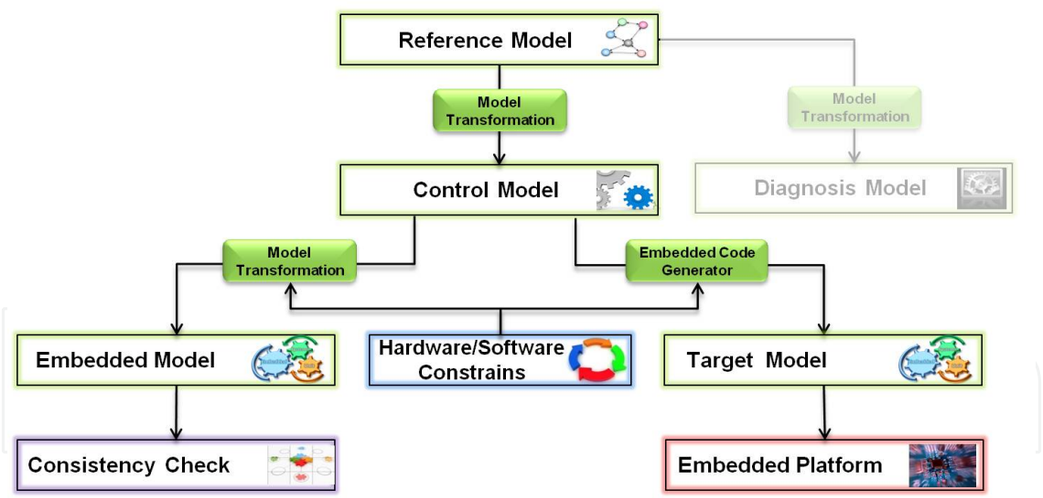


Fig. 1. Framework Overview

An NCS is widely used in many applications, such as habitat monitoring, object tracking, fire detection and modern building control systems. In particular, Building Automation Systems (BAS) often uses a large wireless/wired sensor network. BAS is selected as our application domain as it is considered as a cornerstone application for decreasing energy consumption; around 40% of total energy use in the West is consumed in the industrial building sector, which accounts for nearly one-third of greenhouse gas emissions. Of this figure, approximately

[1] In this chapter, we consider only the control aspects in the reference model.

one-third can be attributed to Heating, Ventilation and Air-Conditioning (HVAC) systems present in buildings. In this chapter, we consider an Air Handling Unit (AHU) control system as a case study for the framework development. We apply the framework design stages on this case starting from the reference model to the embedded control code.

We assume that we are given as input a set $\mathfrak{R}$ of top-level system requirements, for example user comfort requirements that any set-point will differ from operating value by less or equal to 10%. We provide empirical guarantees that the requirements $\mathfrak{R}$ are met by the generated models and the corresponding transformation rules, by empirically evaluating the system property for each generated model (i.e. reference, control, target model). If the system property is respected for each design stage model, then the models are transformed correctly.

Our contributions in this chapter will be as follows:

- we formulate an end-to-end framework for designing a network control system. This framework preserves the modelled system's properties under hardware/software constraints;
- we identify the transformation rules between the framework design stages;
- we formulate a typical AHU control system as a case-study of the framework;
- we empirically check that our framework preserves the system's requirements $\mathfrak{R}$ by applying it to an Air Handling Unit (AHU) model as a case-study.

The remainder of the chapter is organized as follows: Section 2 provides a survey covering the related work and discusses our contribution comparing to the state-of-the-art. The framework architecture and the description for each design stage modelling are discussed in Section 3. The model transformation rules between the design stages are explained in Section 4. The application domain for the case-study are highlighted in Section 5, and its experiments design is shown in Section 6. We end in Section 7 by giving a discussion of our work and outlining future perspectives.

2. Related work

Modelling Frameworks: Modeling languages define a representation method for expressing system design. Given the heterogeneity of engineering design tasks, modelling languages consequently cover a wide range of approaches, from informal graphical notations (e.g., the object modelling technique (OMT) Rumbaugh et al. (1991)), to formal textual languages (e.g., Alloy for software modelling Jackson (2002)).

Semantic meta-modelling is a way to uniformly abstract away model specificities while consolidating model commonalities in the semantics meta-model. This meta-modelling results in a mechanism to analyze and design complex systems without renouncing the properties of the system components. Meta-modelling enables the comparison of different models, provides the mathematical machinery to prove design properties, and supports platform-based design.

An abstract semantics provides an abstraction of the system model that can be refined into any model of interest Lee & Sangiovanni-Vincentelli (1998). One important semantic meta-model

framework is the *tagged signal model* (TSM) Lee & Sangiovanni-Vincentelli (1998), which can compare system models and derive new ones.

There are several prior studies on the translational semantics approach. For example, Chen et al. (2005) use the approach to define the semantic anchoring to well-established formal models (such as finite state machines, data flow, and discrete event systems) built upon AsmL Gurevich et al. (2005). Further, they use the transformation language GME/GReAT (Graph Rewriting And Transformation language) Balasubramanian et al. (2006). This work, through its well-defined sets of semantic units, provides a basis for similar work in semantic anchoring that enables for future (conventional) anchoring efforts.

Tools Several embedded systems design tools that use a component-based approach have been developed, e.g., MetaH Vestal (1996), ModelHx Hardebolle & Boulanger (2008), Model-Integrated Computing (MIC) Sztipanovits & Karsai (1997), Ptolemy Lee et al. (2003), and Metropolis Balarin et al. (2003). These tools provide functionality analogous to the well-known engineering tool MATLAB/Simulink. In particular, Metropolis and Ptolemy II are based on semantic metamodelling, and hence obtain the entailed abstract semantics (and related abstract metamodels) of the approach. In these tools, all models conforming to the operational versions of the TSM's abstract semantics also conform to the TSM's abstract semantics. One drawback of these tools is that "components" can be assembled only in the supporting tool. As a consequence, different systems and components must all be developed in the same environment (tool) to stay compatible. However, the most recent version of Metropolis, Metropolis II, can integrate foreign tools and heterogeneous descriptions.

One tool that is closely related to our approach is the Behaviour Interaction Priority (BIP) tool Basu et al. (2006). BIP can combine model components displaying heterogeneous interactions for generating code for robotics embedded applications. BIP components are described using three layers, denoting behaviour, component connections and interaction priorities. Our approach focuses more on the higher-level aspects, in that it uses two levels of meta-model (i.e., meta-model and meta-meta-model) to define all underlying specifications. Moreover, our approach considers the consistency check between the system model and the hardware/software constraints for the embedded platform.

To our knowledge, our approach is unique in its use of two levels of meta-models, a single centralized reference model with a hybrid systems semantics, and its generation of embeddable code directly from the centralized meta-model considering hardware/software constrains.

3. Modelling framework architecture

In this section, we provide a global description for modelling framework. In addition, each design stage of our framework is formulated.

3.1 Modeling objectives

Our objective is to abstract the essential properties of the control generation process so that we can automate the process. It is clear that the process has two quite different types of inputs:

Control constraints The control-theoretic aspects concern sequences of actions and their effects on the plant. These constraints cover order of execution, times for actions to be executed, and notions of forbidden states, etc. Note that these constraints assume infinite computational power to actually compute and effect to stated control actions.

Embedded System constraints These aspects concern the capabilities of the hardware and software platforms, and are independent of the applications being executed on the platform.

It is clear that both property types are needed. Because of the significant different between the two types, we must use different constraint representations for each type. As a consequence, we then must use a two-step process to enforce each constraint type. This is reflected in the fact that we have a two-step model-generation process. Step 1 maps from a reference model ϕ_R to a control model ϕ_C using mapping rules $\mathcal{R}_C$; the second step maps ϕ_C to ϕ_E using mapping rules $\mathcal{R}_E$. If we represent this transformation process such that $\phi_C = f(\phi_R, \mathcal{R}_C)$, and $\phi_E = f(\phi_C, \mathcal{R}_E)$, then we must have the full process represented by $\phi_E = f(f(\phi_R, \mathcal{R}_C), \mathcal{R}_E)$.

3.2 Meta-meta-model formal definition

The meta-meta-model is formulated using a typical hierarchal component-based modelling Denckla & Mosterman (2005), as shown in Fig. 2. We can define a meta-meta-model Γ as $\Gamma = \langle C, \mathrm{Y} \rangle$. C represents a set model components, where control components C_C, plant components C_P and building-use components C_B are C instances, i.e., $C \in \{C_C, C_P, C_B\}$. The connection between the components C are described by Y.

Each component $c \in C$ is represented as $c = \langle P_{io}, \phi, C \rangle$, where P_{io}[2] is a set of component input/output ports and ϕ describes the relation between the input and output ports. Moreover, c can contain a set of components C to represent hierarchical component levels. Relation ϕ can be expressed based on the use (application domain) of the framework. In this article, we consider four meta-model instances of ϕ: $\phi \in \{\phi_R, \phi_C, \phi_E, \phi_T\}$, where ϕ_R is the reference meta-model, ϕ_C is the control meta-model, ϕ_E is the embedded meta-model and ϕ_T is the target embedded meta-model. Each of these meta-models captures the DSML abstraction for the framework design stage. In this case, for each design stage we can use any Domain-Specific Modelling Language (DSML) that can be represented using the corresponding meta-model. In this article we have selected the DMSLs that support designing NCS for BAS system. However, this design methodology can be adapted to match any other application domain.

3.3 Reference meta-model formal definition

In the context of BAS modelling, Hybrid Systems (HS) are used to create our reference model ϕ_R. We have used HS to present BAS models as it captures both discrete (e.g. presence detection) and continuous (e.g. heat dissipation) dynamics, where the continuous dynamic is represented using algebraic/differential equation. Hence, Linear Hybrid Automata (LHA) become a suitable HS candidate for this model.

[2] In our framework, we assume that the used ports are unidirectional ports.

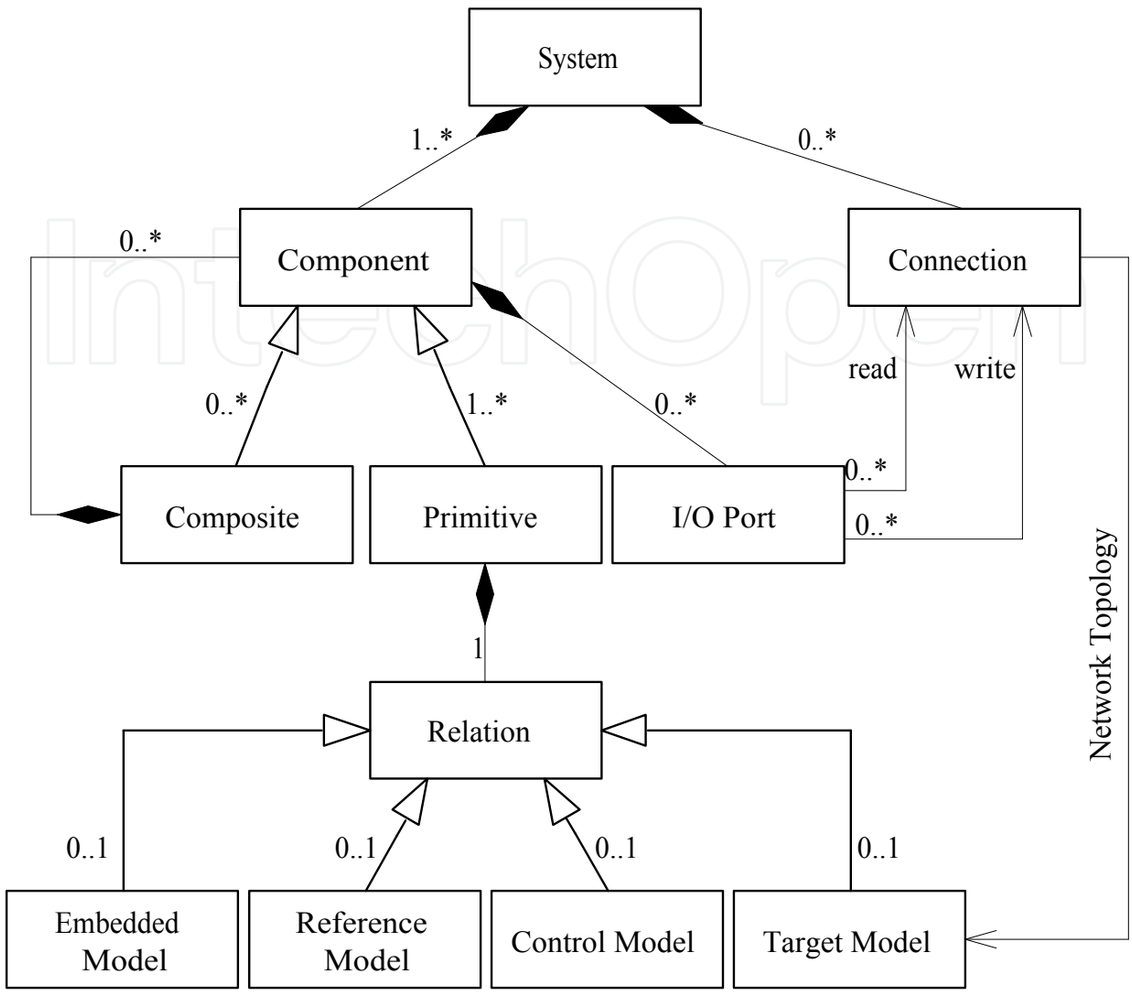


Fig. 2. Framework Meta-Meta-Model

The reference meta-model is considered as one instance of the component relation $\phi = \phi_R$. As shown in Fig. 3, we represent ϕ_R as a standard LHA ($\phi_R = H$) representation, as described in Def. 3.1.

Definition 3.1. ***[Linear Hybrid Automata]*** *A linear hybrid automaton H is a 3-tuple, i.e. $H = \langle V, M_s, E \rangle$, with the following structural extensions:*

- *V is a finite set of component binding variables. It is used to bind/connect the component ports P_{io} with ϕ. V is represented as following: $\bar{V} = \{\bar{v}_1, ..., \bar{v}_n\}$ for real-valued variables, where n is the dimension of H. $\dot{V} = \{\dot{v}_1, ..., \dot{v}_n\}$ represents the first derivatives during continuous change. $\acute{V} = \{\acute{v}_1, ..., \acute{v}_n\}$ represents values at the conclusion of discrete change.*
- *M_s is a set of hierarchical system-level modes that describe the system statuses. $m_s \in M_s$ captures the system-level status using either H or single mode m, i.e., $m_s \in \{H, m\}$. m can be diagnosis mode m_d or control mode m_c, i.e. $m \in \{m_d, m_c\}$. We assume that m_c controls nominal system behaviour and does not consider fault modes (e.g., fault actuation). Two vertex functions assigned to each mode $m \in M$ or $m_s \in M_s$. Invariant ($inv(m)$) condition is a predicate whose free variables*

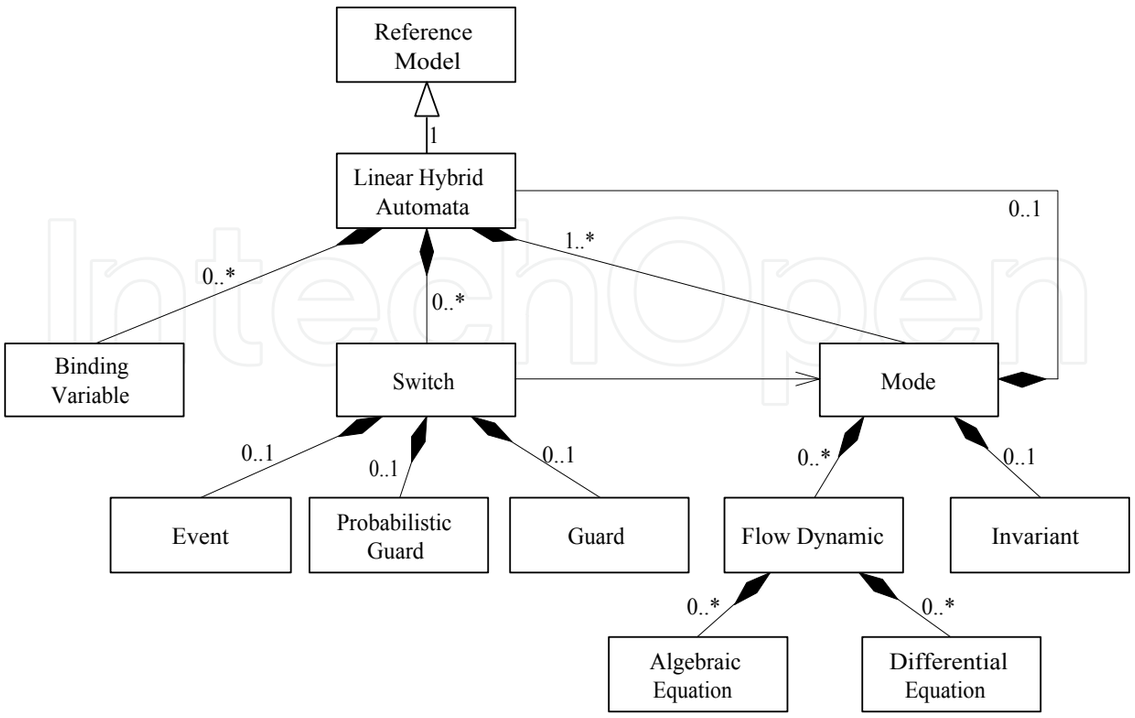


Fig. 3. Meta-Model for the Framework Reference Model

are from V and flow dynamic $flow(m)$ is described using algebraic equation $flow_{alg}(m)$ and/or differential equation $flow_{dif}(m)$ for $\bar{V} \cup \dot{V}$ binding variables.

- *E is set of switches (edges). An edge labelling function j that assigns to each switch $e \in E$ a predicate. E is also assigned to a set Σ of events, where event $\sigma \in \Sigma$ is executed if the corresponding predicate j is true, where j can be presented using probabilistic or deterministic predicate. Each jump/guard condition $j \in J$ is a predicate whose free variables are from $\bar{V} \cup \acute{V}$. For example, $e(m_i, m_l)$ is a switch that moves from mode m_i to m_l under a guard j then executes σ, i.e., $e(m_i, m_l) : m_i \xrightarrow{j/\sigma} m_l$.*

3.4 Control meta-model formal definition

The control meta-model ϕ_C aims to capture the control components C_C in a discrete behaviour. Therefore, Fig. 4 shows a Finite-State Machine (FSM) F that used to describe the components relation $\phi = \phi_C$. In Def. 3.2, we formally define the control meta-model using FSM, i.e., $\phi_C = F$.

Definition 3.2. ***[Finite-State Machine]*** *A finite-state machine F is a 3-tuple, i.e. $H = \langle V, S, T \rangle$, with the following structural extensions:*

- *V is a finite set of component binding variables. It is used to bind/connect the component ports P_{io} with ϕ. V is represented as discrete change only.*
- *S is a set of states that used to identify the execution position.*
- *T is set of transitions to move from one state s to another. Similar to E in LHA, a set Σ of events (actions) and jump/guard condition J are assigned to T. For example, a transition $t(s_i, s_l) \in T$ is*

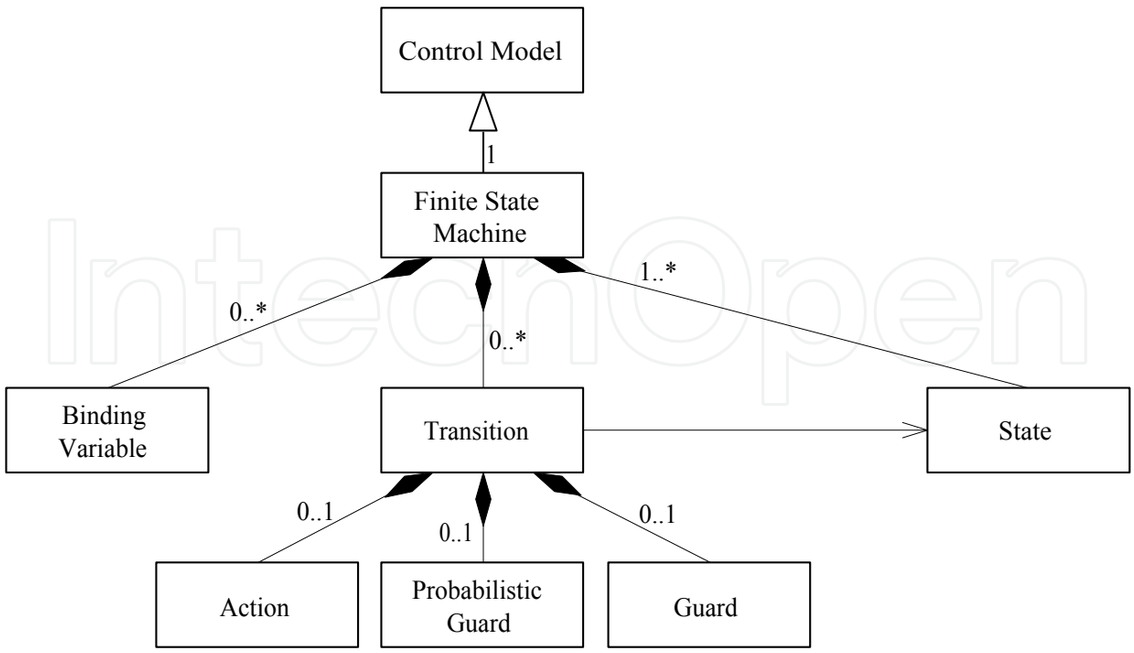


Fig. 4. Meta-Model for the Framework Control Model

used to reflect the move from state s_i to s_l under a guard j (i.e., deterministic or probabilistic) then executing σ, i.e., $t(s_i, s_l) : s_i \xrightarrow{j/\sigma} s_l$.

3.5 Embedded meta-model formal definition

In order to check the consistency between our control model and the hardware/software constrains, we identify the embedded model following its corresponding meta-model ϕ_E shown in Fig. 5. One standard modelling language that can present our embedded model is Analysis and Design Language (AADL)[3] (i.e., SAE Standard). However, any other language/tool that can capture ϕ_E elements can be used in the hardware/software consistency check.

Definition 3.3. ***[Embedded Model]*** *An embedded model ϕ_E is a 5-tuple, i.e. $\phi_E = \langle Mem, Pro, Bus, Thr, Dev \rangle$, with the following structural extensions:*

- *Mem is a RAM memory identification used in the embedded platform during the control algorithm execution. This memory contains 3-tuple used to identify the memory specification, i.e., $Mem = \langle Spc, Wrd, Prt \rangle$, where Spc identifies the memory space size, Wrd identifies the memory word size, and Prt identifies the memory communication protocol (i.e., read, write, read/write).*
- *Pro is a processor identification used in the embedded platform to execute the control algorithm. In ϕ_E, the Pro specification is identified using the processor clock period Clk, i.e., $Pro = \langle Clk \rangle$.*
- *Bus is a bus identification used to communicate between different hardware components, such as Mem and Pro. This Bus contains 2-tuple to identify its specification such as the bus latency Lcy and massage size Msg, i.e., $Bus = \langle Lcy, Msg \rangle$.*

[3] http://www.aadl.info/

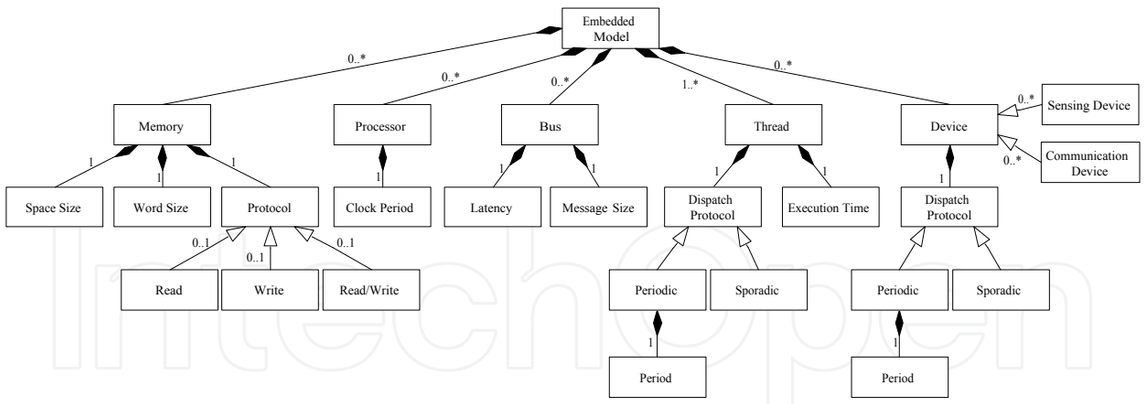


Fig. 5. Meta-Model for the Framework Embedded Model

- *Thr is a thread identification used to run/execute the control algorithm over Pro. This Thr contains a 2-tuple to identify its specification, such as the dispatch protocol $Dprt$ and execution time Tex, i.e., $Thr = \langle Dprt, Tex \rangle$. The $Dprt$ can be either sporadic or periodic. In case of periodic dispatch protocol, we need to identify the period for the thread switching.*
- *Dev is a device identification used to sense/actuate the physical plant, or activate (sending or receiving) the wireless communication devices. In Dev, we identify dispatch protocol $Dprt$ to trigger Dev, i.e., $Dev = \langle Dprt \rangle$. The $Dprt$ can be either sporadic or periodic. In case of periodic dispatch protocol, we need to identify the Dev activating period.*

We can deduce that the hardware/software constraints needed for the consistency check are:

1. Memory space size.
2. Memory word size.
3. Memory communication protocol.
4. Processor clock period (CPU).
5. Bus latency.
6. Bus massage size.
7. Each thread execution time.
8. Each thread switching protocol.
9. Device activation/fire protocol.

The embedded model evaluates:

1. The binding consistency between the hardware and software description, considering the hardware/software constrains.
2. The processing capacity, i.e., how much the described processor is loaded with the execution for the described software. This evaluation factor must be less than 100%, otherwise the designer has to increase the processing resources (e.g., number of the processors).

3.6 Target embedded meta-model formal definition

The target embedded model presents the embedded code that will be deployed on the embedded platform. Fig. 6 depicts the meta-model ϕ_T for the target embedded model. This ϕ_T captures the primitive building elements that needed to describe an embedded code. Consequently, we can use any embedded language by identifying the corresponding semantic for each ϕ_T element. In this article, we focus on embedded Java for Sun-SPOT[4] sensors and base-station.

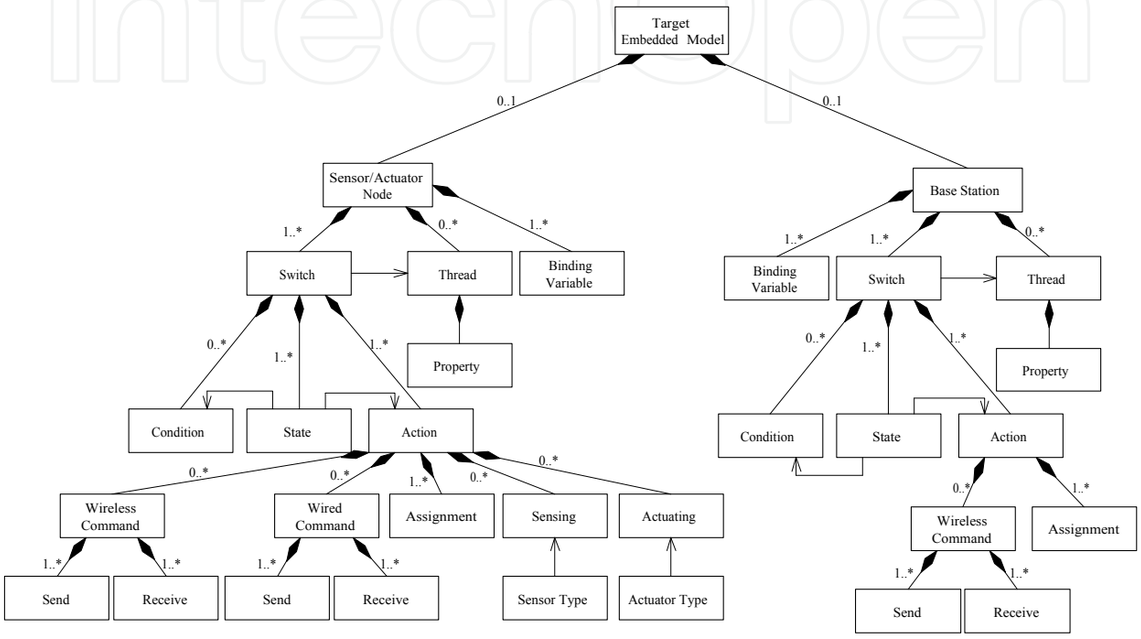


Fig. 6. Meta-Model for the Framework Targeted Model

The target embedded meta-model can formally be described as in Def. 3.4.

Definition 3.4. ***[Target Embedded Model]*** *An embedded model ϕ_T is a 2-tuple, i.e. $\phi_T = \langle N_S, N_B \rangle$, where $N_S = \langle T, Thr, V \rangle$ and $N_B = \langle T, Thr, V \rangle$ are the semantic definitions needed in case of sensor node and base-station deployment, respectively, with the following structural extensions:*

- *T is a set of switch semantic definition. Considering a switch $t \in T$, $t = \langle j, s, \sigma \rangle$, where j is the predicate that allows executing the action σ, and s the a Boolean variable to identify the execution position. State s is included in the semantic of j and σ, as following:*

```
if (state && condition){
    action;
    assignment(s);
}
```

The action σ semantic can contain wireless/wired commands, typical assignment, and/or sensing/actuation commands. However, at this stage some hardware/software constrains are needed

[4] http://www.sunspotworld.com/

in order to identify the sensor/actuator type and the sensors ID, which identified from the network topology. For example, in case of sending a wireless data from sensor node, the semantic is as follows:

```
try {
     SendDg.reset();
     SendDg.writeInt(NodeID);
     SendDg.writeDouble(DataToBeSent);
     SendConn.send(SendDg);
}
catch (Exception e) {
                     throw e;
}
```

- *The thread Thr is used to identify the semantics of a thread as follows.*[5]

```
public class Thread_name extends Thread{
    public void run(){
        while(true/SynchCond){
            Switches semantics;
        }
    }
}

Thread_name.sleep(period);
```

- *V is a finite set of binding variables that can be identified as Integer, Double, Boolean.*

Regarding the base-station node N_B, it has the same aforementioned description with less actions. Typically, N_B is used to communicate wirelessly between the sensors and the actuator, therefore its action contains the wireless command and assignment semantics only.

4. Model transformation

In this section, we provide the transformation rules R used to transform an instance ϕ of the source relation to the corresponding instance of the target representation. As mentioned before, the developed framework aims to generate control code, therefore R considers the control components C_C (i.e., sensor, actuator, controller, base-station). The connections Y and Ports P_{io} that identified in the meta-meta-model remain the same. However, each model has a corresponding representation for Y and P_{io}. For example, in the reference model the components can be connected through a variable, whereas in the target model the connection can be a wireless channel.

[5] If there is only one thread under execution, the thread doesn't have a critical role in this case. Moreover, *Thr* has a period property to be identified.

4.1 Reference model to control model transformation

The transformation rules R_C transforms a ϕ_R model to a ϕ_C model. For simplifying the transformation rules R_C, we have used the same symbols for the unchanged terms between ϕ_R and ϕ_C. Using R_C, the control model can be generated from the reference model as $\phi_C = f(\phi_R, R_C)$, where R_C abstracts the flow dynamics, invariants, and hierarchy in ϕ_R. In this section, we have highlighted the transformation rules as follows:

- Binding variables V in H are transformed to an equivalent format in F, i.e., $V^H \xrightarrow{R_C} V^F$. In this case, the continuous change variables in H will be updated discretely in F.
- The control modes M_c in H are transformed to states S in F, i.e., $M_c^H \xrightarrow{R_C} S^F$. In this case, the states S capture the positions of M_c without its operations (e.g. invariant, flow dynamics). This transformation assumes that there is no fault control modes, therefore the diagnosis modes M_d (presenting faulty modes) is not transformed.
- The switches E in H (E^H) are transformed to transitions T in F, i.e., $E^H \xrightarrow{R_C} T^F$.
- The invariant $inv(m)$ and flow dynamics equations $flow(m)$ for each mode in H are transformed in F to a transition from/to the same state, i.e., $t(s,s) : s \xrightarrow{j/\sigma} s$. The transition guard j is triggered if the mode invariant is true. In order to avoid activating multiple transitions at the same time, we add (logical and) to j the set $\grave{J}$ of the inverse guards for the transitions moving out from state s, i.e., $j = inv(m) \wedge \bar{\grave{J}}$, where $\grave{J}$ is the guards of the transitions leaving s, i.e., $t(s,*)$. This multiple transitions activation can happen if the invariant and any other transition guard are true, which is acceptable in H but not in F. The flow dynamics equations $flow(m)$ are transformed to the transition action σ, i.e., $flow(m) \xrightarrow{R_C} \sigma$. However, differential equations $flow_{dif}(m)$ in the form of $\dot{x(t)} = f$ are approximated to discrete behaviuor using Euler's method, i.e., $x(k+h) = x(k) + fh$, where k is the step number and h is the model resolution that reflects the time step (as much h is small, as the approximation is accurate).

  These rules can be summarised as: $flow(m), inv(m) \xrightarrow{R_C} t(s,s) : s \xrightarrow{inv(m) \wedge \bar{\grave{J}}/flow(m)} s$.

The aforementioned rules do not consider the hierarchical feature in the system-level modes M_s. Given a system-level mode $m_s \in M_s$ that contains a set of inherited control modes M_c, i.e., $m_s \preceq M_c$. An initial control mode m_{init} is identified in M_c, m_{init} is the first mode under execution whenever m_s is triggered. The transformation of these hierarchical modes is as follows:

- The set of control modes M_c in m_s is transformed to a set of states S_c. Consequently, m_{init} is transformed to s_{init}.
- The union of: (a) the set of switches E_c^H that enters m_{init}, i.e., $E_c^H(*, m_{init})$; and (b) the set of the switches E_s^H that enters m_s, i.e., $E_s^H(*, m_s)$ is transformed to a set of transitions T_c in F that enters the initial state s_{init}, called $T_c^F(*, s_{init})$. Therefore, $E_c^H(*, m_{init}) \cup E_s^H(*, m_s) \xrightarrow{R_C} T_c^F(*, s_{init})$.

- The set of switches E_c^H that enters $m_c \in M_c$, i.e., $E_c^H(*, m_c)$ is transformed to a set of transitions T_c in F that enters $s_c \in S_c \setminus s_{init}$[6], i.e., $T_c^F(*, s_c)$. Therefore, $E_c^H(*, m_c) \xrightarrow{R_C} T_c^F(*, s_c)$.
- The set of switches E_c^H that exits m_c, i.e., $E_c^H(m_c, *)$, is transformed to a set of transitions T_c in F that exits $s_c \in S_c$, called $T_c^F(s_c, *)$. Therefore, $E_c^H(m_c, *) \xrightarrow{R_C} T_c^F(s_c, *)$. However, in order to give a higher priority to the system transitions in case of activating control and system transitions at the same time, we introduce the inverse of the condition for the system transition to J of $T_c^F(s_c, *)$, i.e., $J(T_c^F(s_c, *)) = J(E_c^H(m_c, *)) \wedge \overline{J(E_s^H(m_c, *))}$.
- The set of the switches E_s^H that exits m_s, i.e., $E_s^H(m_s, *)$ is transformed to be added to the set of transitions T_c in F that exits $s_c \in S_c$. Therefore, $E_s^H(m_s, *) \xrightarrow{R_C} T_c^F(s_c, *)$.
- The invariants $inv(m_c)$ and $inv(m_s)$, and flow dynamics equations $flow(m_c)$ and $flow(m_s)$ for m_c and m_s, respectively, are transformed in F to a transition from/to the same state, i.e., $t(s_c, s_c) : s_c \xrightarrow{j/\sigma} s_c$. The transition guard j is triggered if the mode invariant for m_c or m_s, and $\bar{\bar{j}}$ (explained earlier) are true, i.e., $j = (inv(m_s) \vee inv(m_c)) \wedge \bar{\bar{j}}$. The flow dynamics equations $flow(m_c)$ and $flow(m_s)$ are assigned to $t(s_c, s_c)$ action σ, i.e., $flow(m_c) \uplus flow(m_s) \xrightarrow{R_C} \sigma$.

4.2 Control model to embedded model transformation

The embedded model is performed from the control model as $\phi_E = \phi_C \otimes R_E$ and can be performed from the reference model as $\phi_E = \phi_R \otimes (R_C \oplus R_E)$, where the transformation rules R_E considers the hardware/software constrains and the hardware/software architecture.

As mentioned before, the control model ϕ_C focuses on the control components C_C. In a BAS system, C_C can be classified to a set of sensor component instances C_C^S, actuator component instances C_C^A and processing/controller component instances C_C^P (which execute the control algorithm), i.e., $C_C \in \{C_C^S, C_C^A, C_C^P\}$. Based on the type of the C_C instance, ϕ_E creates the corresponding hardware architecture, e.g., Fig. 7, Fig. 8, and Fig. 9 show a sensor component architecture, an actuator component architecture, and a processing/controller component architecture, where each architecture contains the appropriate *Pro*, *Mem*, *Bus*, *Dev*. As shown in Fig. 5, each of these hardware components has its own hardware/software constrains.

In order to add the software impact to this architecture, we consider a transformation rule R_E from control model ϕ_C. Typically, each control model ϕ_C contains only one F, therefore F transformed to a thread in ϕ_E, i.e., $F \xrightarrow{R_E} Thr^E$, where Thr^E captures the algorithm execution constrains.

The multi-threading appears in case if a control component C_C contains several inherited components. In this case, each component is transformed to a thread and all the transformed threads will be executed under the same processor.

[6] This rule does not consider s_{init}

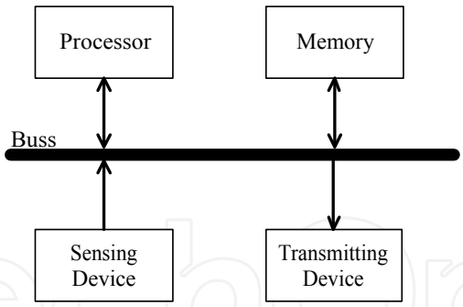


Fig. 7. Sensor Hardware Architecture

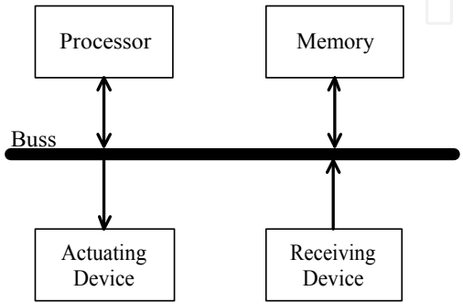


Fig. 8. Actuator Hardware Architecture

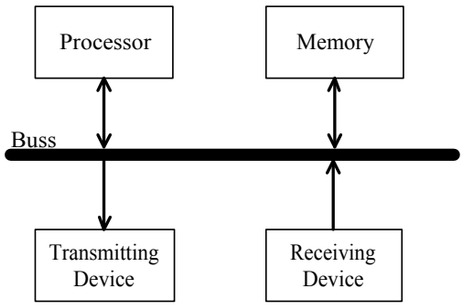


Fig. 9. Controller Hardware Architecture

4.3 Control model to target embedded model transformation

The target embedded model is performed from the control model as $\phi_T = \phi_C \bigotimes R_T$ and can be performed from the reference model as $\phi_T = \phi_R \bigotimes (R_C \bigoplus R_T)$. The transformation rules R_T considers only the hardware/software constrain, whereas R_E considers both hardware/software constrains and architecture. The R_T considers only constrains as ϕ_T directly uses the hardware architecture on the targeted platform.

In this section, we show the transformation rules R_T between the control model ϕ_C and the target embedded model ϕ_T, as follows:

- The set of binding variables V of ϕ_C (called V^C) is transformed to an equivalent set of variables of corresponding type (e.g., Double, Integer, Boolean) in the binding variables V^T of ϕ_T, i.e., $V^C \xrightarrow{R_T} V^T$.

- The set of transitions T in ϕ_C (called T^C) is transformed to a set of switches T^T of ϕ_T, i.e., $T^C \xrightarrow{R_T} T^T$.
- The set of guards J, including probabilistic and deterministic, of T^C is transformed to a set of conditions J of T^T, i.e., $J(T^C) \xrightarrow{R_T} J(T^T)$.
- The set of actions Σ of ϕ_C is transformed to a set of actions Σ of ϕ_T, i.e., $\Sigma(T^C) \xrightarrow{R_T} \Sigma(T^T)$. However, additional P_{io} constrains of N_B and N_S are needed to R_T for identifying the P_{io} specification. For example, for a sensor node N_B one port should read from a sensor, this port name should be identified and the corresponding sensor type. Therefore, whenever an action reads from this port, a sensor reading action is added. The same is done for the ports read/write from/to a wireless network, and these ports are identified with the connected ID node, where this process identifies the network topology.
- The set of states S of ϕ_C is transformed to a set of states (i.e., Boolean variables) in ϕ_T, i.e., $S^C \xrightarrow{R_T} S^T$. In addition, the set of transitions $T^T(s^T, *)$ that exiting $s^T \in S^T$ includes s^T in its conditions and $s^T = false$ in its actions. The set of transitions $T^T(*, s^T)$ that enter s^T includes $s^T = true$ in its actions. This transformation rule should be followed in the specified order, which means starting from the transitions exiting the states and then applying the ones entering the states.

5. Application domain

Heating, Ventilating, and Air-Conditioning (HVAC) systems provide a specified ambient environment for occupants with comfortable temperature, humidity, etc. One way to regulate the temperature in a room is Air Handling Unit (AHU), it is a set of devices used to condition and circulate air as part of an HVAC system. Several control strategies have been introduced to control the temperature regulation, where an operating scheduling is pre-defined. In this context, standard PI control algorithms are adequate for the control of HVAC processes Dounis & Caraiscos (2009). Therefore, we consider an AHU model used to regulate the temperature for a single room, as out framework case-study. The AHU heating/cooling coils are controlled using a PI algorithm, where the switch between cooling and heating coil is performed using a system level decision.

We describe in this section the system specification for the used model and evaluation metrics for the system property. Moreover, we apply the end-to-end transformation process for the system reference model.

5.1 System evaluation metric

In this case-study, we regulate the temperature (for maintaining thermal comfort) with respect to user discomfort. The expected discomfort metric penalizes the difference between the measured indoor air temperature $y(k)$ at time-step k (where k varies from k_s to k_f), and the reference temperature $r(k)$. We use the Root Mean Square Error (RMSE) to reflect the indoor temperature variation around the reference temperature in ^oC, during only the scheduling period $p(k) = 1$:

$$DI = \sqrt{\sum_{k=k_s}^{k_f} \frac{(y(k) - r(k))^2}{k_f} p(k)} \quad (1)$$

In order to validate the transformation rules between the models, each generated model[7] has to respect the system property $DI \leq 2\ ^oC$.

5.2 System specification

As shown in Fig. 10, an AHU uses coils to heat or cool the indoor temperature. The heated/cooled air is pumped to the room using a supply-air fan. In order to use the heated/cooled indoor air more efficiently, the AHU recycles some of the return air via an air loop, which has a return-air fan to mix the indoor air with the outdoor air, as controlled by three dampers. In our case-study, we assume fixed settings for the fan speeds and the damper settings, and we control only the valve settings for the coils.

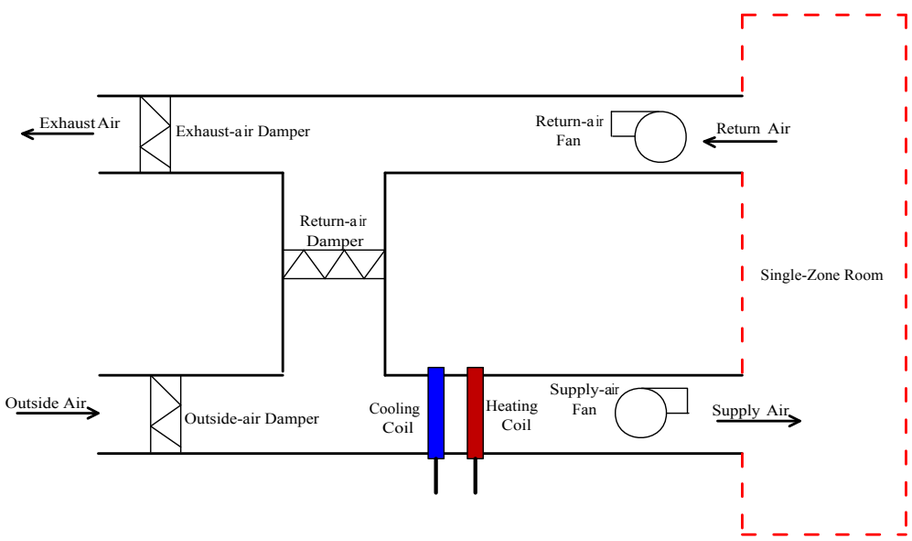


Fig. 10. AHU Structure

Fig. 11 shows the reference model that used as our framework case-study. This model contains two main groups of components: (a) environment/plant components reflect the plant physics (e.g., walls, coils); (b) control components show the control/sensing algorithm that used to modify/monitor the environment.

5.2.1 Environment components

All models described below are lumped-parameter models. Two variables are identified to evaluate the model behaviour: external temperature T_{ext} and set-point temperature T_{sp}. Moreover, five environment/plant components have been used: Wall, Window, Heating/Cooling Coil, Return Air and Indoor Air models, as follows:

[7] This rule will not be applied to embedded model ϕ_E, as this model used to validate the hardware/software consistency

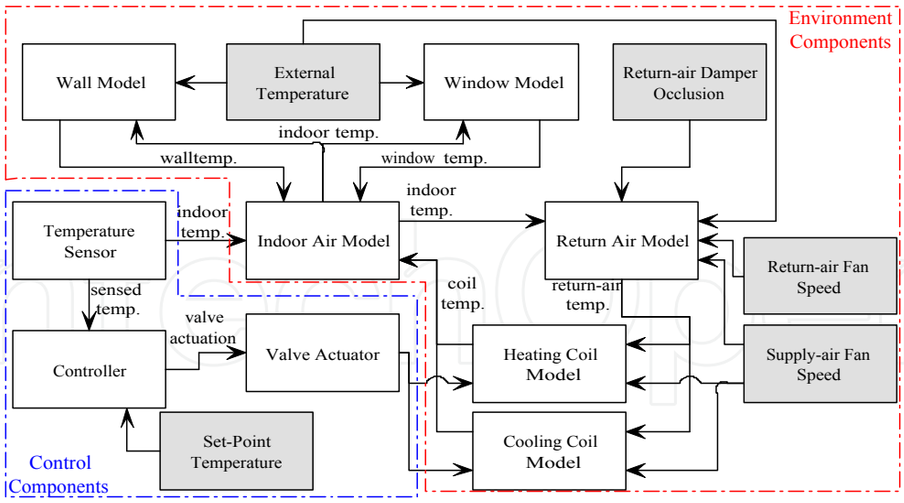


Fig. 11. AHU Component-Based Model

1. *Wall Model*: One of the room walls is facing the building façade, which implies heat exchanges between the outdoor and indoor environments. In general, a wall can be modelled using several layers, where greater fidelity is obtained with increased layers in the wall model. In our case, four layers have been considered to reflect sufficient fidelity Yu & van Paassen (2004), using the following differential equation; Eq. 2:

$$\rho_{wall} V_{wall} c_{wall} \frac{dT_{wall}}{dt} = \alpha_{wall} A_{wall} (T_{ext} - T_{wall}) \tag{2}$$

ρ_{wall} is the wall density $[kg/m^3]$, V_{wall} is the wall geometric volume $[m^3]$, c_{wall} is the wall specific heat capacity $[J/kg.K]$, T_{wall} is the wall temperature $[^oC]$, α_{wall} is the wall thermal conductance $[W/(m^2.K)]$, A_{wall} is the wall geometric area $[m^2]$ and T_{ext} is the outdoor temperature $[^oC]$.

2. *Heating/Cooling Coil Model*: The coil model uses the temperature difference between the water-in and water-out in order to heat/cool the room. In this case, the temperature is controlled through the radiator water flow using the valve occlusion (called actuation variable u). Moreover the coil exchanges temperature with its environment, such as the indoor air temperature and returned air temperature (considering the supply fan speed) as shown in equations: Eq. 3 and Eq. 4.

$$M_{wtr} c_{wtr} \frac{dT_{coil}}{dt} = \dot{m_{wtr}} c_{wtr} (T_{wtrin} - T_{wtrout}) - Q \tag{3}$$

$$Q = \alpha_{air} A_{coil} (T_{coil} - T_{air}) + \dot{m_{supfan}} c_{air} (T_{coil} - T_{retair}) \tag{4}$$

Where, M_{wtr} is water mass $[kg]$, c_{wtr} is water specific heat capacity $[J/kg.K]$, T_{coil} is coil temperature $[^oC]$, $\dot{m_{wtr}}$ is water mass flow rate throw the coil valve $[kg/s]$, T_{wtrin} is water temperature going to the coil $[^oC]$, T_{wtrout} is water temperature leaving from the coil $[^oC]$, α_{air} is air thermal conductance $[W/(m^2.K)]$, A_{coil} is coil geometric area $[m^2]$, T_{air} is indoor

air temperature [^oC], $\dot{m}_{supfan}$ is air mass flow rate throw the supply-air fan [kg/s], c_{air} is air specific heat capacity [$J/kg.K$], and T_{reair} is returned air temperature [^oC].

3. *Return Air Model*: This model acts as an air mixer between the indoor air flow and the external air based on return-air damper occlusion ε, as shown in equations: Eq. 5.

$$T_{retair} = T_{air} + \varepsilon(\frac{\dot{m}_{retfan}}{\dot{m}_{supfan}})(T_{air} - T_{ext}) \tag{5}$$

Where, $\dot{m}_{retfan}$ is air mass flow rate throw the return-air fan [kg/s].

4. *Indoor Air Model*: In order to model the indoor temperature T_{air} propagation, all HVAC components have to be considered as they exchange heat with the air inside the controlled room following equation 6 .

$$\rho_{air} V_{air} c_{air} \frac{dT_{air}}{dt} = Q_{air} + Q_{wall} + Q_{window} \tag{6}$$

$$Q_{wall} = \alpha_{air} A_{wall}(T_{wall} - T_{air}) \tag{7}$$
$$Q_{air} = \dot{m}_{supfan} c_{air}(T_{coil} - T_{air}) \tag{8}$$
$$Q_{window} = \alpha_{air} A_{window}(T_{window} - T_{air}) \tag{9}$$

Where, ρ_{air} is air density [kg/m^3], V_{air} is air geometric volume [m^3], A_{window} is window geometric area [m^2], and T_{window} is window temperature [^oC].

5. *Window Model*: A window has been modelled to calculate the effects of glass on the indoor environment. Since the glass capacity is very small, the window has been modelled as an algebraic equation, Eq. 10, that calculates the heat transfer at the window node.

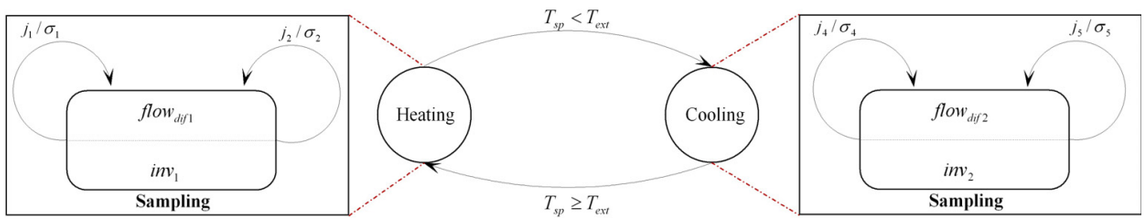
$$\alpha_{air}(T_{ext} - T_{window}) + \alpha_{air}(T_{air} - T_{window}) = 0 \tag{10}$$

5.2.2 Control components

A temperature sensor, PI-controller, and valve actuator are used to monitor, control and actuate the environment model, respectively. The temperature sensor samples the indoor temperature (T_s=1 min) and sends the sampled value to the controller. In case the sampled value within the operating time slot (identified by the operating schedule), then the PI-controller calculates the next actuation value and sends it to the valve actuator in order to adjust the coil valve occlusion.

The reference model for the control algorithm has two levels of hierarchy, as shown in Fig. 12. The high level represents the system modes of the system decisions, where the heating mode is activated if T_{ext} is greater than T_{sp}, and the cooling mode is activated if T_{sp} is greater than T_{ext}. In these system modes, a PI-control algorithm is used to adjust the valve occlusion for the corresponding coil and deactivate the other coil. Consequently, the controller updates the actuation value each sampling period implemented using a continuous variable to present the controller timer.

In this document, we consider the controller component as an example to apply the end-to-end transformation rules. Fig. 12 shows the reference model for the controller component with its transitions, actions, invariants and flow dynamics. However, we have used the same design approach to design the reference model for the sensor and actuator components.



$$
\begin{aligned}
j_1 &: ControlTime > SamplingPeriod \wedge Schedule \\
\sigma_1 &: ControlTime = 0 \\
&: u_{heating} = f_{PI}(T_{sp}, T_{air}) \\
&: u_{cooling} = 0 \\
j_2 &: ControlTime > SamplingPeriod \wedge \overline{Schedule} \\
\sigma_2 &: ControlTime = 0 \\
&: u_{heating} = 0 \\
&: u_{cooling} = 0 \\
flow_{dif1} &: \frac{dControlTime}{dt} = 1 \\
inv_1 &: ControlTime \leq SamplingPeriod
\end{aligned}
$$

Fig. 12. Reference model for the controller component, where the high level automata describes the system level modes M_C and the inherited level represents the control modes M_C.

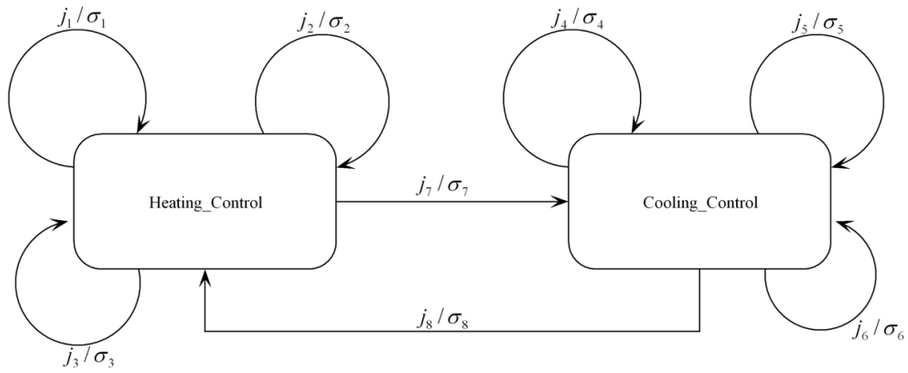
Where, $ControlTime$ is a continuous variable used to reflect the controller time, $SamplingPeriod$ is the sampling period needed to receive the sensor sample and then update the actuator, $Schedule$ is a Boolean variable to indentify if the operation schedule is triggered or not, $u_{heating}$ is the valve occlusion value for the heating-coil, f_{PI} is the heating PI optimization function, and $u_{cooling}$ is the valve occlusion value for the cooling-coil.

The cooling system mode uses similar LHA as in heating. However, it actuates on $u_{cooling}$ using $\grave{f}_{PI}$ instead of $u_{heating}$, where $\grave{f}_{PI}$ is the cooling PI optimization function.

5.3 Case-study control model

We have followed the transformation rules R_C in Sec. 4.1 to transform the controller reference model in Fig. 12 to the controller control model in Fig. 13 with the corresponding transition conditions J and actions Σ.

We can deduce that system modes hierarchy has been removed by adding the complement of its transition conditions to the conditions in the control model transitions. Moreover, the mode dynamic flow and invariant are transformed to $t_3(j_3, \sigma_3)$.



$$
\begin{aligned}
j_1 &: ControlTime > SamplingPeriod \wedge Schedule \wedge \overline{T_{ext} > T_{sp}} \\
\sigma_1 &: ControlTime = 0 \\
&: u_{heating} = f_{PI}(T_{sp}, T_{air}) \\
&: u_{cooling} = 0 \\
j_2 &: ControlTime > SamplingPeriod \wedge \overline{Schedule} \wedge \overline{T_{ext} > T_{sp}} \\
\sigma_2 &: ControlTime = 0 \\
&: u_{heating} = 0 \\
&: u_{cooling} = 0 \\
j_3 &: ControlTime \leq SamplingPeriod \wedge \overline{T_{ext} > T_{sp}} \\
\sigma_3 &: ControlTime = ControlTime + h \\
j_7 &: T_{ext} > T_{sp} \\
\sigma_7 &: ControlTime = 0
\end{aligned}
$$

Fig. 13. Control model for the controller component, where the system mode hierarchy, continuous variables are abstracted.

The rest of the control model transitions follow the same conditions and actions as the aforementioned transitions, but for the cooling-coil.

5.4 Case-study embedded model

In our case-study, we create the controller component hardware architecture for the embedded model as in Fig. 9 following R_E in Sec. 4.2. We have emulated the embedded model using Sun-SPOTs with the following hardware specifications: (a) CPU=180MHz, 32 bit ARM 920T Processor, and (b) RAM=512 Kbyte. Therefore, hardware/software constraints captured by the embedded model as follows:

1. Memory space size: $Spc(Mem) = 512kbyte$.
2. Memory word size: $Spc(Wrd) = 32bit$.

3. Memory communication protocol: $Spc(Prt) = read/write$.
4. Processor clock period (CPU): $Clk(Pro) = 1/180M$.
5. Bus latency: $Lcy(Bus) = 11ns$.
6. Bus massage size: $Msg(Bus) = 32bit$.
7. Thread execution time: $Tex(Thr) = 5 - 10ms$. In this example, we have considered only one thread as the controller component is a primitive component. However, in case of a composite component, it is translated to multi-thread.
8. Thread switching protocol: $Dprt(Thr) = periodic$ with switch period $20ms$. This constrain is not critical for our case-study as the processor execute only one thread.
9. Sending/Receiving Communication Device activation/fire protocol: $Dprt(Dev) = periodic$ with a period equals to the network sampling rate T_s.

5.5 Case-study target embedded model

In the target embedded model, we follow R_T in Sec. 4.3 to generate the embedded language semantic. In our case-study we have used embedded Java for Sun-SPOT devices. In the controller component, $u_{heating}$, $u_{cooling}$ and T_{air} are identified as the component wireless P_{io}, which are corresponding to u_heating, u_cooling, and IndoorTemp variables in Fig. 14, respectively. These P_{io} identifications are transformed to wireless communication commands in case of reading/writing from/to P_{io}. Moreover, the sending/receiving IDs are identified for each port in order to structure the network topology, and consequently these IDs are included in the wireless commands. For example, Fig. 14 shows the first transition transformation from the control model, where $sensorID$ is the sensor ID that sends T_{air} ($IndoorTemp$) to the controller and the controller ID is identified as $hostID$.

6. Experimental design

In order to check if the developed framework preserves the systems properties, we run a set of experiments for the AHU system as a case-study at each model (i.e., reference, control, and target). Then, we evaluate if each model respects the identified system property $\Re = DI \leq 2^oC$.

In our case-study, we vary external temperature and the set point temperature (T_{ext}, T_{sp}) and then evaluate the corresponding DI. For example, if we have an input 2-tuple $(15, 23)$ to present (T_{ext}, T_{sp}), we will measure outputs $DI = 0.73^oC$.

6.1 Dependent/independent variables

In our case-study, we evaluate each model over the domains of the independent variables as follows:

1. External Temperature $T_{ext} \in \{5, 10, 15, 20\}$;
2. Set-point Temperature $T_{sp} \in \{18, 20, 23\}$;

where some constraints are identified for the variables' search space to eliminate the physically unrealizable solutions as $\mid T_{ext} - T_{sp} \mid \leq T_{max}$, $T_{max} = 13^oC$ is the maximum allowable temperature difference and $T_{ext} \neq T_{sp}$.

```
if (Heating_Control && ControlTime > SamplingPeriod && Schedule
        && !(ExternalTemp > OptimalTemp)){

    ControlTime = 0.0;
    //Receiving indoor temperature from sensor with ID: sensorID.
    try {
        RecConn.receive(RecDg);
        recAddr = RecDg.getAddress();
        sensorID=RecDg.readInt();
        if (ID=sensorID) IndoorTemp = RecDg.readDouble();
    } catch (Exception e) {
        throw e;
    }
    //Updating the heating actuation value using PI algorithm.
    u_heating=heatingPIOptimization(OptimalTemp,IndoorTemp);
    //Sending the heating actuation value packet to the actuator
    //with the base station ID (hostID).
    try {
        recAddr = RecDg.getAddress();
        SendDg.reset();
        SendDg.writeInt(hostID);
        SendDg.writeDouble(u_heating);
        SendConn.send(SendDg);
    } catch (Exception e) {
        throw e;
    }
    //Deactivating the cooling-coil.
    u_cooling=0.0;
    //Sending the actuation value packet to the actuator
    //with the base station ID (hostID).
    try {
        recAddr = RecDg.getAddress();
        SendDg.reset();
        SendDg.writeInt(hostID);
        SendDg.writeDouble(u_cooling);
        SendConn.send(SendDg);
    } catch (Exception e) {
        throw e;
    }
    //Update the execution state.
    Heating_Control=true;
    Cooling_Control=false;
}
```

Fig. 14. Target embedded model of the first transition t_1 of the controller component.

The dependent variable for each model is the average of DI over the independent variable cross-product space given by $T_{ext} \times T_{sp}$.

6.2 Empirical results

Fig. 15, 16 and 17 show the DI experiments for ϕ_R, ϕ_C, and ϕ_T. The reference and control models are empirically evaluated using the CHARON tool [8], whereas the target embedded model is emulated using Sun-SPOT nodes, with the following hardware specifications: (a) CPU=180MHz, 32 bit ARM 920T Processor, and (b) RAM=512 Kbyte. The embedded model is captured using the AADL language and then evaluated using the OSATE tool[9].

Figs. 15, 16 and 17 show that the DI decreases when the difference between T_{ext} and T_{sp} decreases, as the initial overshoot the settling time decreases. Moreover, we see that control model gives the worst DI values, the reference model improves the DI and the target model gives the best DI values. Because of the model resolution h for updating the dynamic flow ($flow_{dif}$, $flow_{alg}$) decreases in case of the target model (in range of msec) in comparison to the reference mode (in range of sec), and it increases for the control model (in range of 3 sec) over the reference model. However, the average of DI for all models respects the system property $DI \leq 2^oC$, where $DI = 0.95, 2, 0.44^oC$ for ϕ_R, ϕ_C, and ϕ_T, respectively.

We have evaluated the binding consistency and the processing capacity for the embedded model ϕ_E to show a consistency binding and 50% processing capacity.

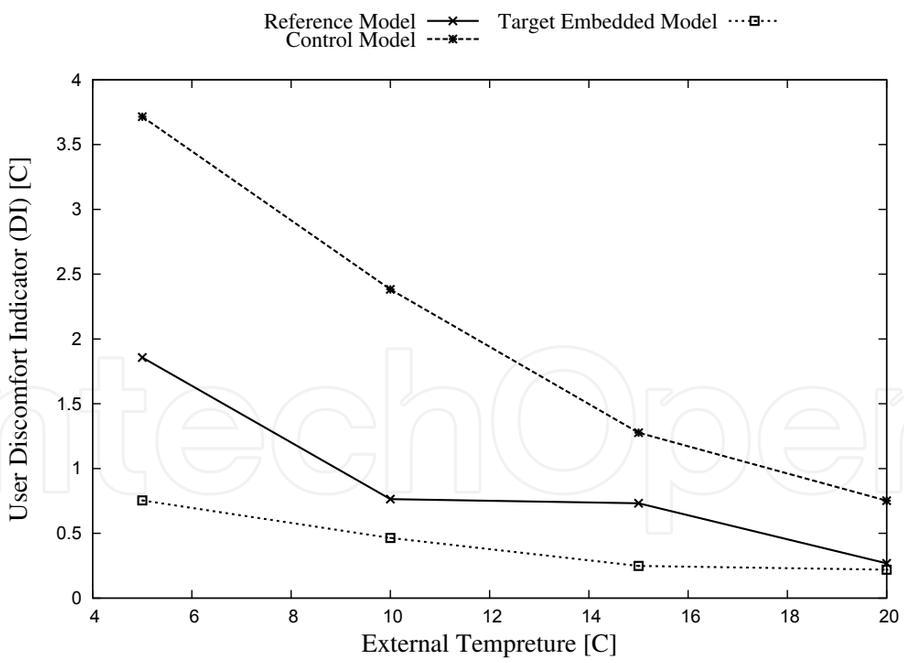


Fig. 15. User Discomfort Evaluation of Framework Models for $T_{sp} = 18$

[8] http://rtg.cis.upenn.edu/mobies/charon/index.html
[9] http://www.aadl.info/aadl/currentsite/tool/index.html

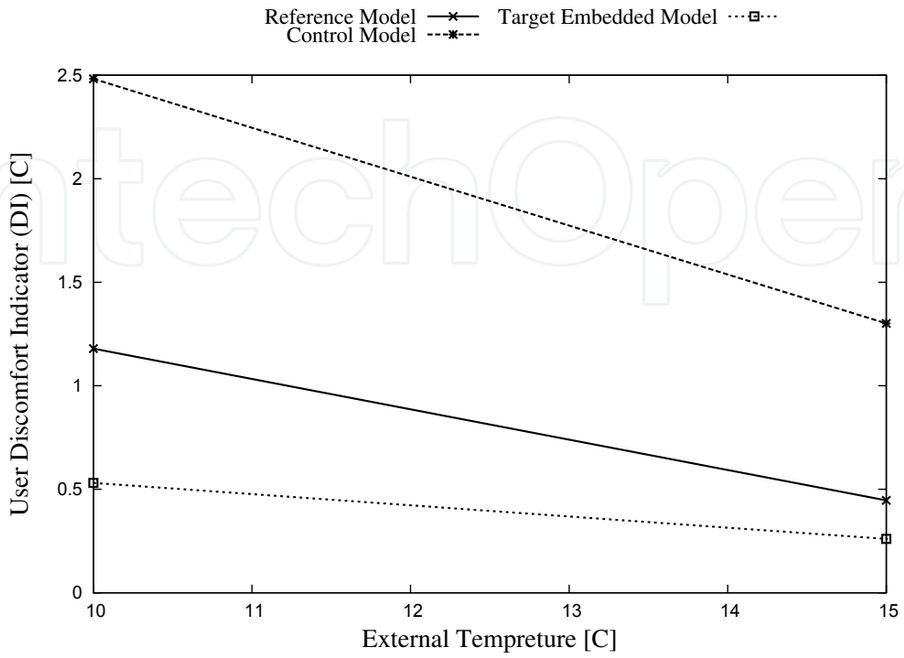


Fig. 16. User Discomfort Evaluation of Framework Models for $T_{sp} = 20$

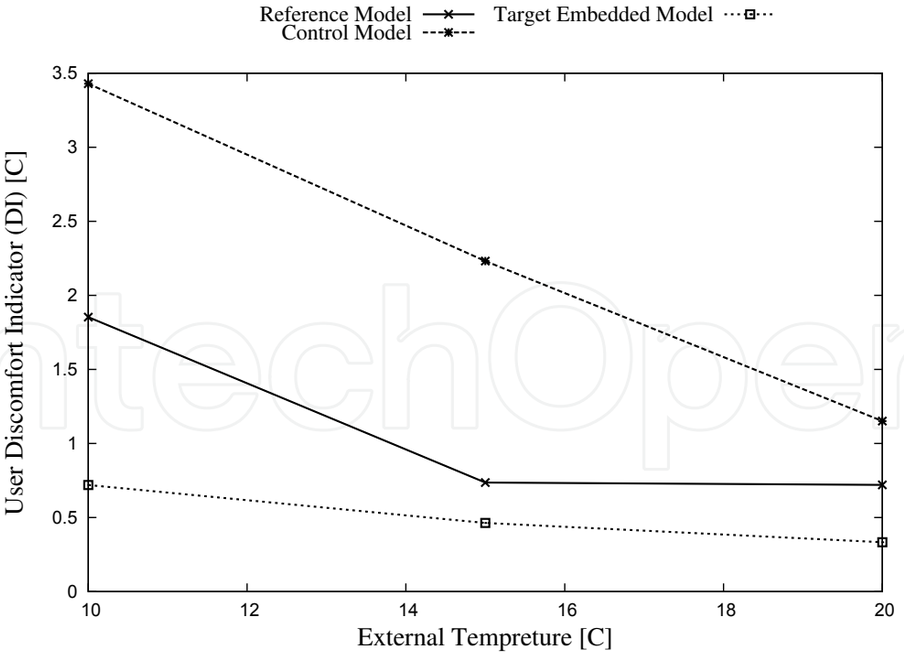


Fig. 17. User Discomfort Evaluation of Framework Models for $T_{sp} = 23$

7. Conclusion

In this article, we have proposed a novel end-to-end framework for designing an NCS for BAS system embedded over a distributed WSAN. This framework considers the design flow stages along from modelling to embedded control-code generation. The developed framework is empirically validated using AHU system as a case-study.

A future trend of this framework is integrating a *de facto* standard tool, such as Simulink, to the reference model. Moreover, we need to consider the timing and delay for the generated embedded code.

8. References

Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C. & Sangiovanni-Vincentelli, A. (2003). Metropolis: An integrated electronic system design environment, *Computer* pp. 45–52.

Balasubramanian, D., Narayanan, A., van Buskirk, C. & Karsai, G. (2006). The graph rewriting and transformation language: Great, *Electronic Communications of the EASST* 1.

Basu, A., Bozga, M. & Sifakis, J. (2006). Modeling heterogeneous real-time components in BIP, *SEFM*, Vol. 6, Citeseer, pp. 3–12.

Chen, K., Sztipanovits, J. & Neema, S. (2005). Toward a semantic anchoring infrastructure for domain-specific modeling languages, *Proceedings of the 5th ACM international conference on Embedded software*, ACM New York, NY, USA, pp. 35–43.

Denckla, B. & Mosterman, P. (2005). Formalizing causal block diagrams for modeling a class of hybrid dynamic systems, *Proc. 44th IEEE Conference on Decision and Control,and European Control Conference (CDC-ECC'05)* pp. 4193–4198.

Dounis, A. & Caraiscos, C. (2009). Advanced control systems engineering for energy and comfort management in a building environment-a review, *Proc. of Renewable and Sustainable Energy Reviews* pp. 1246–1261.

Gurevich, Y., Rossman, B. & Schulte, W. (2005). Semantic essence of AsmL, *Theoretical Computer Science* 343(3): 370–412.

Hardebolle, C. & Boulanger, F. (2008). ModHel'X: A component-oriented approach to multi-formalism modeling, *Lecture Notes In Computer Science* 5002: 247–258.

Henzinger, T. (1996). The theory of hybrid automata, *Proc. 11th Annual IEEE Symposium on Logic in Computer Science (LICS 96)* pp. 278–292.

Jackson, D. (2002). Alloy: a lightweight object modelling notation, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11(2): 256–290.

Lee, E., Neuendorffer, S. & Wirthlin, M. (2003). Actor-oriented design of embedded hardware and software systems, *Journal Of Circuits Systems And Computers* 12(3): 231–260.

Lee, E. & Sangiovanni-Vincentelli, A. (1998). A framework for comparing models of computation, *IEEE Transactions on computer-aided design of integrated circuits and systems* 17(12): 1217–1229.

Mady, A. & Provan, G. (2011). Co-design of wireless sensor-actuator networks for building controls, *the 50th IEEE Conference on Decision and Control and European Control Conference (IEEE CDC-ECC)*.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object oriented modeling and design*, Prentice Hall, Book Distribution Center, 110 Brookhill Drive, West Nyack, NY 10995-9901(USA).

Sztipanovits, J. & Karsai, G. (1997). Model-integrated computing, *IEEE computer* 30(4): 110–111.

Vestal, S. (1996). MetaH programmerŠs manual, *Technical report*, Version 1.09. Technical Report, Honeywell Technology Center.

Yu, B. & van Paassen, A. (2004). Simulink and bond graph modeling of an air-conditioned room, *Simulation Modelling Practice and Theory, Elsevier* .

IntechOpen