A Design Flow for Building Automation and Control Systems

Yang Yang
EECS Department, UC Berkeley
yangyang@eecs.berkeley.edu

Alessandro Pinto
United Technologies Research Center
PintoA@utrc.utc.com

Alberto Sangiovanni-Vincentelli
EECS Department, UC Berkeley
alberto@eecs.berkeley.edu

Qi Zhu
Intel Corporation
qi.dts.zhu@intel.com

Abstract

We propose a system-level design flow for building automation and control (BAC) systems. The input to the design flow is a high level description of the control algorithms given in a model-based environment such as Simulink. The input specification is translated into an intermediate format, and then automatically refined into a distributed implementation. Refinement includes optimal mapping of the functional specification on a set of computation and communication resources, and software synthesis, which generates code for each component in the mapped design while guaranteeing semantic equivalence with the original specification. Experiments with a temperature control system are presented to illustrate the flow.

1 Introduction

The building stock in the US accounts for 40% of total energy consumption and 70% of electricity consumption [18]. Limits on carbon emissions are driving new regulations that will require buildings to be energy efficient according to standards that are likely to be more stringent than the ASHRAE 90.1 [2]. The design of low energy buildings – zero energy in the ideal case – is challenging but not impossible. There are today examples of zero energy buildings [24], but they are the results of *ad-hoc* designs that are not easy to generalize.

The design methodology used today for large buildings is top-down. Different sub-systems (e.g., mechanical and electrical) are designed in isolation by domain experts following design documents flown down after the bid process. This methodology is not suitable for low energy buildings that require interaction among architects, mechanical engineers and control engineers. Consider for instance adopting low energy solutions such as natural ventilation and active facade. In this case, architectural design (e.g. building orientation), the design of the mechanical equipments of the HVAC system and the design of the control algorithms cannot be done in isolation. In this new context, the design of the building automation system (i.e. the embedded processors and networks supporting the building operations, and the software running on them) is non-trivial. Control algorithms become multi-input, multi-output, hybrid and predictive, as opposed to single-input single-output controllers coordinated by simple switching conditions as today (and mainly dictated by standards). Moreover, several sub-systems such as HVAC, lighting, vertical transportation and fire and security will interact through the network to allow information sharing.

In this paper we focus on a design flow for building automation systems that bridges the gap between a desirable design entry point – at a high abstraction level using model-based design tools such as Simulink [7] – and the available back-end tools able to generate low-level code. It enables the integration of models from different high-level languages, allowing the interaction between domain experts. Further, it automatically optimizes the implementation of the control algorithms on a distributed platform by selecting computation and communication resources, and by performing code generation while meeting the specification.

2 Proposed Design Flow

The design flow proposed in this article consists of a front-end and a back-end. The front-end is used to model the system including the control algorithms and the behavior of the environment. The back-end includes a set of tools that, given the specification of the control algorithms and a set of available computation and communication resources, automatically refines the specification into an optimal distributed implementation. The design flow is shown in Fig-

1052-8725/10 $26.00 © 2010 IEEE
DOI 10.1109/RTSS.2010.26

IEEE computer society

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

ure 1. The front-end and the back-end exchange models over an intermediate format (IF). The introduction of this intermediate layer is necessary to build a design flow which is general with respect to the user input (e.g. Simulink, Modelica [5] and LabVIEW [6]), and to the output code (e.g. C and EIKON [3]). Using an IF, pieces of the input specification expressed in different languages can be composed. This feature will hopefully foster collaboration among experts in different disciplines who could exchange models and evaluate their system taking into account the interactions with other sub-systems. The intermediate level allows targeting several implementation platforms. Building control system vendors usually provide architecture-specific languages for programming their platforms, along with tool chains for simulation, analysis, debugging and code generation. These tools can be leveraged by translating the intermediate format into the vendor specific language. Compared to providing customized design flows from each high-level language to each architecture specific language, the intermediate format reduces the number of translators needed from a quadratic number to a linear number.

The translation process may become very involved given the expressiveness of model-based languages. Our approach to deal with the complexity of this step is to define a *library* of primitives at the intermediate level designed to capture a large class of building control algorithms and that can be extended by users. This library is then mirrored by equivalent libraries defined in the source languages. The set of models that can be translated into the intermediate format is the one obtained as composition of the library elements. This architecture simplifies the translation process and will be described later.

The back-end is responsible for mapping the functional model described in the intermediate format to the architectural model that captures the implementation platform. Specifically, the part of the functional model to be mapped is a control algorithm. The architecture platform captures computation resources (e.g. terminal control units, embedded processors and workstations), communication resources (e.g. wired buses and wireless links), sensors (e.g. temperature sensors and CCTV video cameras) and actuators (e.g. valves and switches). During mapping, the functional model is abstracted into the composition of functional tasks and messages among them. There may be constraints that come with the specification such as latency, energy and cost. The architecture platform is described in the form of a library of available architectural components that are characterized by their functionality, cost, performance, etc. The plant model is abstracted into a set of physical constraints imposed on the system. The mapping problem is cast into an optimization problem that is solved by algorithms designed to find the best mapping, with respect to a set of objective functions, from the tasks and messages in the functional model to the components in the architectural model, while satisfying a set of design constraints.

After mapping, code needs to be generated for final deployment. The third step of the design flow is software synthesis that starts from the mapped design and includes code generation for each processor in the distributed system, and communication interface synthesis for process communication. During code generation, we translate the functional tasks mapped onto each processor to either generic C code – if compilation tools are available for the processor – or a vendor specific language for which the code generator is usually provided. The synthesis of communication interfaces is essential to ensure the correctness of the system when the architecture platform does not directly support the semantics of the functional model. For instance, a synchronous Simulink model is not naturally supported by an asynchronous architecture that is common in building control systems. In this case, the generated communication interfaces help to ensure the synchronous functionality is correctly preserved on the asynchronous platform.

3 Step 1: Intermediate Format Translation

In the first step of the proposed design flow, models capturing the specification of the control algorithms and of the environment are translated into an intermediate representation. This representation is based on a language called intermediate format (*IF*) that should facilitate the other steps in the design flow, namely mapping and code generation. Because the type of specifications that we are interested in are in general hybrid systems [16] with multiple semantics, the IF representation may become very complex [23, 20], and thus not directly usable in the mapping and code generation steps. In the envisioned final form of our design method, IF will be manipulated and partitioned to make the mapping and code generation steps effective. In this paper that is a first step towards the ideal scenario, we restrict the intermediate format to dataflow semantics [15] which is amenable to efficient mapping and code generation. We retain the nomenclature introduced in [23] as we plan to extend this work to more general intermediate representations. In particular, each process (also called *actor*) is characterized by an input-output function described by a set of "equations". When the process is scheduled to run, the equations are executed according to an order determined by an *equation manager* (EM) that is local to the process. The set of processes in the system is scheduled by an *equation resolve manager* (ERM). Processes communicate over media that, in the restricted case dealt with in this paper, are implemented as FIFOs.

To enable fast translations to IF, we define a domain specific IF library for HVAC control systems in buildings, and we export the library to different specification languages.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

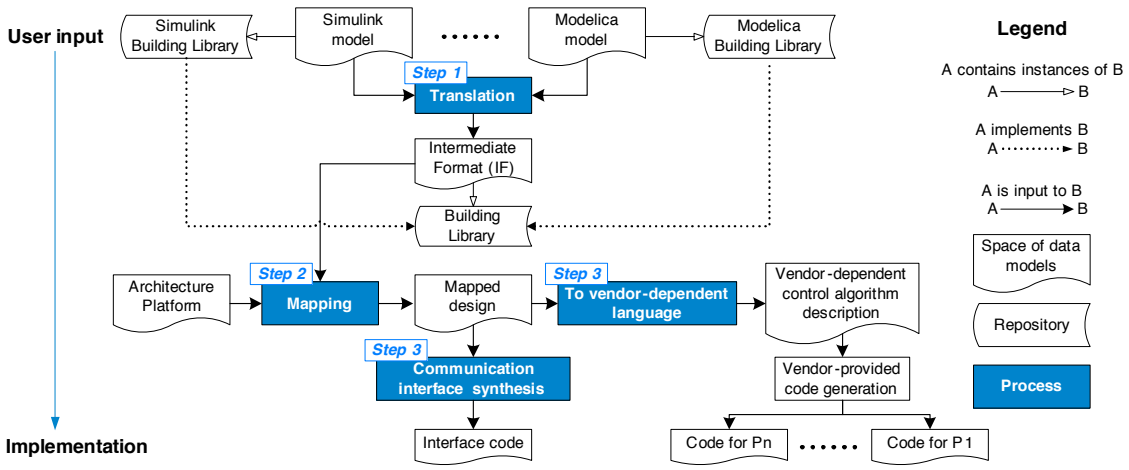

Figure 1. Design flow for building automation and control

We reviewed 71 HVAC-related component models in the GPL language from Johnson Controls [8], 70 in Automated Logic EIKON language [9], 42 in Honeywell Spyder [10], and 59 in the HVAC library defined by the Lawrence Berkeley National Laboratory [26]. Based on these information, we defined a set of basic components used in HVAC control systems and the corresponding processes in the IF, including:

- *Mathematical functions*: ADD, SUB, MUL, DIV, ABS, SQRT, MIN, MAX, SUM, AVG, INTEGRATOR, DERIVATIVE, GAIN.
- *Logic functions*: INV, AND, OR, XOR.
- *Signal processing functions*: SWITCH, LIMIT, SPAN, COMP, PID.
- *Time functions*: TIME, DATE, DELAY, TIMER, OCCUPENCYSCHEDULE.
- *Psychrometric functions* [1]: ENRH, WBTRH, DPTRH, ENW, WBTW, DPTW.

As an example, the PID component in our IF library is described as follows:

```
PID {
  parameter double Kp, Ki, Kd, Kc;
  parameter bool outMin, outMax;
  port double setPoint, var, out;
  equations{
    err = setPoint - var;
    sum = Kp*err + Ki*int(err) + Kd*dev(err)
        + Kc*diff(out,sum);
    out = (sum>outMax)*outMax
        + (sum<outMin)*outMin
        + (sum<=outMax && sum>=outMin)*sum;
  }
}
```

The PID component uses anti-windup to avoid integrator windup when the actuator saturates because of its physical limitations (e.g. a control valve cannot go beyond fully open or fully closed). It contains three equations that are scheduled in the order in which they appear.

Each component may have slightly different implementations in each language. For instance, there are many different algorithms for PID controllers besides the one defined above. We chose a few common cases in our component definitions as a proof of concept. Additional components can be added to the library by designers. Also note that the library contains components at different abstraction levels. A PID controller is at a higher level of abstraction than the mathematical functions and can be constructed from them. This enables translations at different abstraction levels and provides a trade-off between accuracy and complexity, as demonstrated later in the case study.

3.1 Case Study of IF Translation

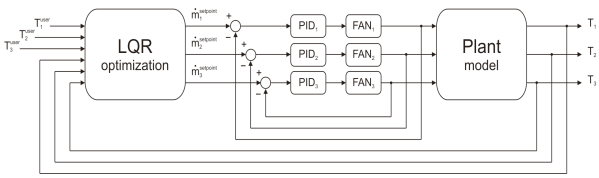

Figure 2. Room temperature control system

We use a temperature control system as example throughout the paper. The functional model captures a two-level control algorithm as shown in Figure 2. The higher

[1]The psychrometric functions describe the thermodynamic properties of moist air that are important for the comfort level of human. The IF library includes enthalpy calculators ENRH and ENW, wet-bulb temperature calculators WBTRH and WBTW, and dew point temperature calculators DPTRH and DPTW.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

level LQR (linear-quadratic regulator) controller determines the set points for lower level PID (proportional-integral-derivative) controllers. The LQR coordinates among multiple rooms to optimize the total energy consumption while maintain a certain comfort level. The PIDs track the set points and interact with the physical environment. This functional model is initially described in Simulink.

In this part of the case study, we show how IF is used to import models from a specification language, and to generate code. We mentioned earlier that several vendor specific languages and tool chains are available for code generation on embedded targets. Thus, we use the term "code generation" here to denote the exporting of a model from the IF to a given target language (that can be directly transformed into runnable code using the vendor tool-chain). Consider Simulink as input specification and the *G* language from National Instruments (NI) as output code, as shown in Figure 3. NI provides both simulation and C code generation for the G language.

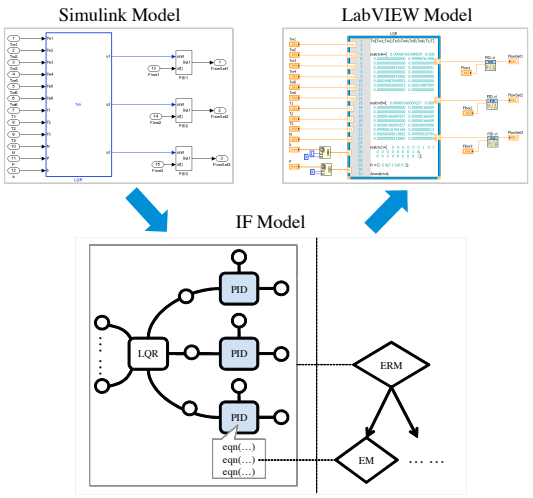

Figure 3. Case study for IF translation

The IF model shown in Figure 3 consists of one LQR process, three PID processes (in darker color), communication media between processes, as well as ERM and EM schedulers. The translations from Simulink to the IF, and from the IF to LabVIEW are straightforward: there exists a one-to-one correspondence between the components in each language and the components in the IF library. The scheduling in Simulink for the dataflow type semantics we consider is based on the causality relation between components. This is translated to the ERM scheduling in the IF, then translated to the component scheduling in LabVIEW, which is also based on causality relations. The Runge-Kutta ODE solver used in Simulink employs a fixed time step and can be translated to the scheduling in ERM and EMs, then translated to the Runge-Kutta solver available in LabVIEW. The plant model that captures the physical environment is not translated since our focus is on the control system.

In order to validate the accuracy of our translations, we directly compared the simulation results of the Simulink model and the LabVIEW model. The plant model from Simulink is imported to LabVIEW to provide a fair comparison of the control system part. In Figure 4, the room temperature and the air flow level of Room 1 from the simulations of the two models are shown. The other rooms have similar plots. The length of the simulation is one day. As shown in figure, the results from the two models are fairly close to each other.

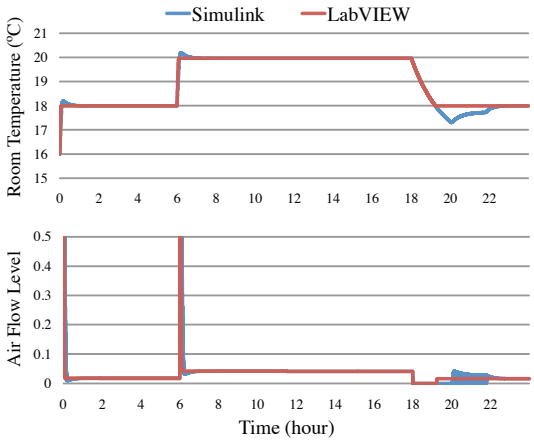

Figure 4. Comparison of Simulink and LabVIEW models

We observed that the differences between the simulations mainly came from the different implementations of the PID controllers. The PID component in the IF library faithfully implements the PID controller in Simulink. However, in the translation from IF to LabVIEW, we could not find a PID controller implementing the same control algorithm. We had to choose a similar PID in LabVIEW that also uses anti-windup but with different algorithm flow. Generally speaking, this difference is a result of translating at higher level of abstraction, where higher level components are viewed as basic units. To reduce the difference, we can break down those components to lower level of abstraction, where more information about the components is exposed and can be potentially maintained. In this case, instead of translating at the PID level, we can break down the PID to lower level components, translate them from Simulink through IF to LabVIEW, and then assemble those lower level LabVIEW components to construct a PID in LabVIEW. This process is shown in Figure 5.

The comparison of the translations at different abstraction levels is shown below. Table 1 shows the absolute differences of the room temperatures between Simulink and

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

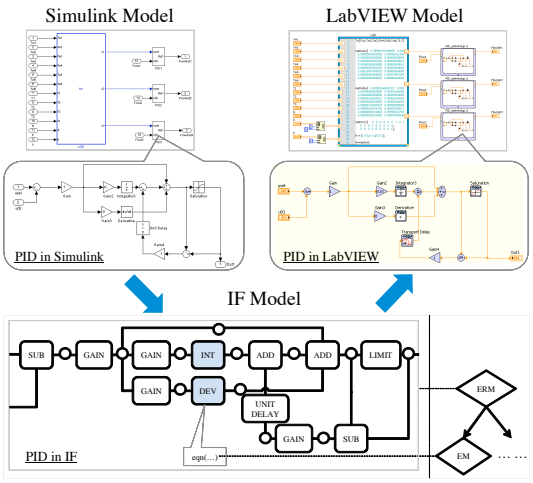

Figure 5. IF translation at lower level

LabVIEW simulations at two different abstraction levels. Table 2 shows the relative differences of the cumulative air flow levels. We can see that the differences are reduced by 10^1 to 10^3 times.

Table 1. Comparison of room temperature

	Level	Room1	Room2	Room3
Avg. differences (^oC)	High	0.0538	0.0538	0.0744
	Low	0.00202	0.00202	0.00415
Max. differences (^oC)	High	0.741	0.741	0.797
	Low	0.0555	0.0555	0.0880

Table 2. Comparison of cumulative air flow

	Level	Room1	Room2	Room3
Cumulative air flow level differences (%)	High	1.29	1.29	1.55
	Low	6.26 $\times 10^{-3}$	6.26 $\times 10^{-3}$	8.15 $\times 10^{-4}$

4 Step 2: Mapping between Functional and Architectural Models

The mapping step selects computation resources, allocates control functions to processors, and synthesizes the communication network. Formulated as an optimization problem, the mapping step minimizes a set of objective functions subject to design constraints.

4.1 General Mapping Flow

The functional model in IF consists of processes, media and schedulers. For mapping purposes, processes and media are abstracted into tasks and messages, respectively, by hiding their internal implementation and computing cost and performance metrics of interest. For example, the equations inside a process are used to estimate the execution time of its corresponding task on various processors. However the real computation sequence is abstracted away. The schedulers in the IF model are not explicitly represented in the mapping, but the causality relations that must be taken into consideration when performing scheduling are reflected in the connections between tasks through messages. Formally, the functional model is represented as a directed graph $\mathcal{F} = (\mathcal{T}, \mathcal{M})$, where $\mathcal{T}$ is the set of tasks and $\mathcal{M}$ is the set of messages that are communicated between tasks.

The architecture platform captures the computation and communication resources that can be used to realize the functional specification. It is defined as a library of architectural components $\mathcal{A} = \{\mathcal{A}_k = (\mathcal{P}_k, \mathcal{L}_k) : \mathcal{P}_k \subseteq \mathcal{P},\ \mathcal{L}_k \subseteq \mathcal{L}\}$, where a component $\mathcal{A}_k$ is the composition of a set of basic computation components $\mathcal{P}_k$ through a set of basic communication components $\mathcal{L}_k$. The set $\mathcal{P}$ contains all available basic computation components such as sensors, actuators and processors. Similarly, the set $\mathcal{L}$ contains all basic communication components such as wired or wireless communication links, routers and repeaters. Labeling functions are defined to associate components in $\mathcal{P}$ and $\mathcal{L}$ with parameters, representing certain characteristics of the components such as performance, cost, bandwidth and latency. Note that $\mathcal{P}$ and $\mathcal{L}$ can contain virtual components, which are place holders that can be refined to real components in later design stages. The parameters associated with the virtual components represent design requirements rather than implementation.

The constraints and objective functions of the mapping problem may include the cost of the electronic system, data acquisition frequencies, real-time constraints such as end-to-end latencies from sensors through controllers to actuators, utilization constraints on computation and communication resources. Further, the building floorplan and geometry impose constraints on the locations of sensors, actuators and processing units, and wire layout.

Our mapping flow is shown in Figure 6. A three-step approach is used to cope with complexity. In the first step, a set of computation components $\mathcal{P}_S$ is selected from the architecture platform and connected by virtual communication components $\mathcal{L}_S$. This constitutes an architectural model $\mathcal{A}_S$ onto which the functional model can be mapped.

In the second step, the tasks in the functional model are allocated to the computation components in the architectural model, and if needed, the priorities of the tasks are assigned. The messages are temporarily allocated to the virtual communication components. The output is the mapped model $G_C = (V_C, E_C)$, where V_C denotes the computation components with tasks allocated onto them and E_C denotes the message-allocated virtual communication components.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

In some cases, when complexity is manageable, the first and second step can be combined and solved together, as shown later in our case study.

Finally, in the third step, the virtual communication components are synthesized to a communication network, in which the communication between two computation components may flow through multiple links, routers and repeaters, and each link may be shared across multiple end-to-end communications. The output G_I is the eventual implementation of the functional model on the architecture platform.

In our flow, we optimize the computation first because the complexity of optimizing computation and communication together is prohibitive for typical industrial size systems, and a fixed set of computation components greatly reduces the complexity of communication optimization. If needed, these steps can be iterated to improve the quality of the solution.

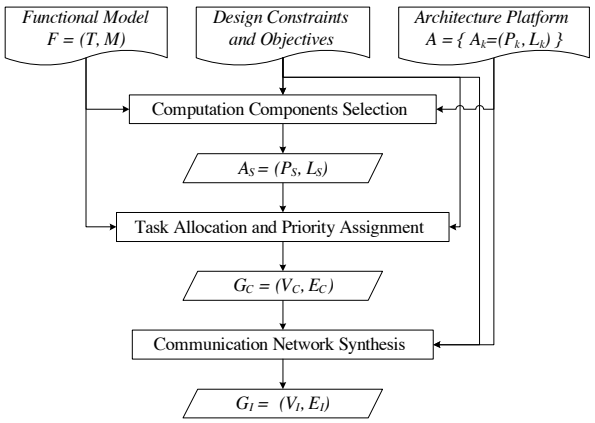

Figure 6. Mapping flow

4.2 Building Mapping Formulation and Algorithm

The mapping flow above is generic: when given specific design requirements and platforms, each of the three steps in the flow can be formulated accordingly and solved by customized algorithms. In this section, we target a typical building design case: given the functional model $\mathcal{F}$, the architecture platform $\mathcal{A}$, and a set of *design constraints* including building floorplan, candidate locations of sensors, actuators, embedded processors and routers, end-to-end latency deadlines on selected paths, utilization and memory constraints on embedded processors, we explore the *design space* containing the selection of computation components, allocation of tasks to embedded processors, assignment of task priorities, and communication network, to minimize *system cost*, which includes the prices of the components and the installation cost.

For this specific problem, we combine the first and second step in the mapping flow and then perform communication network synthesis.

4.2.1 Computation Components Selection, Task Allocation and Priority Assignment

The set of candidate computation components is denoted as $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, which includes sensors, actuators and embedded processors. In our use case, we assume for each sensing or actuating task in the functional model, one sensor or actuator is selected manually from the library by the designer, depending on the physical environment and design requirements. For the selection of processors, there are usually various options on how many and what type should be used. As an example, for the functional model shown in Figure 2, we can either select a single powerful processor for running all PID and LQR tasks, or select multiple less powerful but cheaper processors (in the extreme case, one processor can be used for each PID or LQR block). We denote the set of candidate processors as $\mathcal{P}'$, a subset of $\mathcal{P}$. For each processor $p_i \in \mathcal{P}'$, V_{p_i} denotes its cost including both price and installation cost, R_{p_i} denotes its maximum available instruction memory, and U_{p_i} denotes its utilization upper bound, which represents the maximum fraction of time the processor can be busy running functional tasks.

The set of tasks in $\mathcal{F}$ is denoted as $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_m\}$. Each task τ_i is labeled with period T_{τ_i}. C_{τ_i,p_j} denotes the worst case execution time of task τ_i on computation component p_j, which can be obtained via either static analysis or dynamic profiling. M_{τ_i,p_j} denotes the required instruction memory for τ_i on p_j. We denote the set of tasks that must be mapped onto processors as $\mathcal{T}'$, which is a subset of $\mathcal{T}$ excluding sensing and actuating tasks (as explained above, they are one-to-one mapped to manually chosen sensors and actuators).

We use Boolean variable a_{τ_i,p_j} to represent whether task τ_i is mapped onto computation component p_j (1 if mapped, 0 otherwise). $\mathcal{P}_{\tau_i}$ denotes the set of candidate computation components that τ_i can be mapped to. If τ_i is a sensing or actuating task, value of a_{τ_i,p_j} is decided by the manual selection and $\mathcal{P}_{\tau_i}$ is set to the chosen sensor or actuator. Boolean variable h_{τ_i,τ_j} denotes whether τ_i and τ_j are mapped onto the same computation component. Boolean variable s_{p_j} denotes whether processor $p_j \in \mathcal{P}'$ is selected.

The communication between tasks occurs through a set of messages $\mathcal{M} = \{m_1, m_2, ..., m_l\}$. src_{m_i} and dst_{m_i} denote the source task and destination task of message m_i respectively. Boolean variable g_{m_i} is 1 if m_i is a global message, i.e. src_{m_i} and dst_{m_i} are mapped to different components, otherwise g_{m_i} is 0. Variable l_{m_i} denotes the worst case transmission delay of m_i, which represents the largest

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

time interval from src_{m_i} sending m_i to dst_{m_i} receiving m_i. The value of l_{m_i} depends on which computation components src_{m_i} and dst_{m_i} are mapped to, and the communication latency between the components. We use L_{p_i,p_j} to denote the communication latency from computation component p_i to p_j, which can be estimated based on the given physical locations of sensors, actuators and candidate processors. Note that this is only a high level estimation of the latency without the details of communication network. In the case that $p_i = p_j$, L_{p_i,p_j} represents the local communication latency between two tasks on the same computation component.

A path $\rho = (\tau_{\rho,1}, m_{\rho,1}, \tau_{\rho,2}, m_{\rho,2}, ..., m_{\rho,k-1}, \tau_{\rho,k})$ in the directed graph of functional model $\mathcal{F}$ is an ordered interleaving sequence of tasks and messages. The worst case end-to-end latency of a path ρ is denoted as l_ρ. The deadline of path ρ, denoted by d_ρ, is an application requirement that may be imposed on selected paths. A typical path for BAC systems would start from a sensing task, pass through tasks running control algorithms, and end at an actuating task.

Let r_{τ_i} denote the worst case response time of a task τ_i, which is the largest time interval from the activation of the task to its completion. The worst case end-to-end latency of a path can be computed as follows.

$$l_\rho = \sum_{\tau_i \in \rho} r_{\tau_i} + \sum_{m_i \in \rho} l_{m_i} + \sum_{m_i \in \rho \wedge m_i \in GS} T_{dst_{m_i}}$$

where GS is the set of all global messages. The periods of the destination tasks of global messages are included in the latency because of the asynchronous nature of the communication. In the worst case, the input global message of a periodical task may arrive immediately after the task was just activated and has to wait for an activation period of the task before it can be read. The formula is similar to one in [13, 29], except that here message latencies are more abstract since we do not have the details of the communication network at this stage of the design.

The computation of worst case task response time r_{τ_i} depends on the scheduling policy of the processor to which the task is mapped. In our case study, we assume the processors employ a preemptive scheduling based on pre-assigned priorities. Under this assumption and in the case of $r_{\tau_i} \leq T_{\tau_i}$, r_{τ_i} can be computed as follows, based on the analysis from [13, 17].

$$r_{\tau_i} = C_{\tau_i} + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{r_{\tau_i}}{T_{\tau_j}} \right\rceil C_{\tau_j}$$

where $hp(\tau_i)$ refers to the set of higher priority tasks on the same processor.

A mixed-integer linear programming (MILP) formulation of the optimization problem is then:

$$\forall \tau_i \in \mathcal{T}, \quad \sum_{p_j \in \mathcal{P}_{\tau_i}} a_{\tau_i,p_j} = 1 \quad (1)$$

$$\forall \tau_i \in \mathcal{T}, p_j \notin \mathcal{P}_{\tau_i}, \quad a_{\tau_i,p_j} = 0 \quad (2)$$

$$\forall p_j \in \mathcal{P}', \quad \sum_{\tau_i \in \mathcal{T}'} a_{\tau_i,p_j} \geq s_{p_j} \quad (3)$$

$$\forall \tau_i \in \mathcal{T}', p_j \in \mathcal{P}', \quad a_{\tau_i,p_j} \leq s_{p_j} \quad (4)$$

$$\forall p_j \in \mathcal{P}', \quad \sum_{\tau_i \in \mathcal{T}'} a_{\tau_i,p_j} * C_{\tau_i,p_j}/T_{\tau_i} \leq U_{p_j} \quad (5)$$

$$\forall p_j \in \mathcal{P}', \quad \sum_{\tau_i \in \mathcal{T}'} a_{\tau_i,p_j} * M_{\tau_i,p_j} \leq R_{p_j} \quad (6)$$

$$\forall p_k \in \mathcal{P}, \quad a_{\tau_i,p_k} + a_{\tau_j,p_k} - 1 \leq h_{\tau_i,\tau_j} \quad (7)$$

$$\forall p_k, p_q \in \mathcal{P}, \quad 2 - a_{\tau_i,p_k} - a_{\tau_j,p_q} \geq h_{\tau_i,\tau_j} \quad (8)$$

$$\forall m_i \in \mathcal{M}, \quad 1 - h_{src_{m_i},dst_{m_i}} = g_{m_i} \quad (9)$$

$$\forall \rho_k, \quad l_{\rho_k} \leq d_{\rho_k} \quad (10)$$

$$\sum_{\tau_i \in \rho_k} r_{\tau_i} + \sum_{m_i \in \rho_k} (l_{m_i} + g_{m_i} * T_{dst_{m_i}}) = l_{\rho_k} \quad (11)$$

$$\sum_{p_k,p_q \in \mathcal{P}} f_{src_{m_i},p_k,dst_{m_i},p_q} * L_{p_k,p_q} = l_{m_i} \quad (12)$$

$$a_{\tau_i,p_k} + a_{\tau_j,p_q} - 1 \leq f_{\tau_i,p_k,\tau_j,p_q} \quad (13)$$

$$a_{\tau_i,p_k} \geq f_{\tau_i,p_k,\tau_j,p_q} \quad (14)$$

$$a_{\tau_j,p_q} \geq f_{\tau_i,p_k,\tau_j,p_q} \quad (15)$$

$$\sum_{\tau_j \in \mathcal{T}} \sum_{p_k \in \mathcal{P}} z_{\tau_i,\tau_j,p_k} * C_{\tau_j,p_k} + \sum_{p_j \in \mathcal{P}} a_{\tau_i,p_j} * C_{\tau_i,p_j} = r_{\tau_i} \quad (16)$$

$$y_{\tau_i,\tau_j,p_k} - M * (1 - o_{\tau_i,\tau_j}) \leq z_{\tau_i,\tau_j,p_k} \leq y_{\tau_i,\tau_j,p_k} \quad (17)$$

$$z_{\tau_i,\tau_j,p_k} \leq M * o_{\tau_i,\tau_j} \quad (18)$$

$$x_{\tau_i,\tau_j,p_k} - M * (1 - a_{\tau_i,p_k}) \leq y_{\tau_i,\tau_j,p_k} \leq x_{\tau_i,\tau_j,p_k} \quad (19)$$

$$y_{\tau_i,\tau_j,p_k} \leq M * a_{\tau_i,p_k} \quad (20)$$

$$u_{\tau_i,\tau_j} - M * (1 - a_{\tau_j,p_k}) \leq x_{\tau_i,\tau_j,p_k} \leq u_{\tau_i,\tau_j} \quad (21)$$

$$x_{\tau_i,\tau_j,p_k} \leq M * a_{\tau_j,p_k} \quad (22)$$

$$0 \leq u_{\tau_i,\tau_j} - r_{\tau_i}/T_{\tau_j} < 1 \quad (23)$$

$$r_{\tau_i} \leq T_{\tau_i} \quad (24)$$

$$\forall \tau_i, \tau_j \in \mathcal{T}', \quad o_{\tau_i,\tau_j} + o_{\tau_j,\tau_i} = 1 \quad (25)$$

$$\forall \tau_i, \tau_j, \tau_k \in \mathcal{T}', \quad o_{\tau_i,\tau_j} + o_{\tau_j,\tau_k} - 1 \leq o_{\tau_i,\tau_k} \quad (26)$$

$$\min \sum_{p_j \in \mathcal{P}'} s_{p_j} * V_{p_j} \quad (27)$$

In this MILP, (1) and (2) enforce that each task should be mapped to one computation component. Equation (3) and (4) define the selection of processors. Equation (5) and (6) set utilization and memory constraints on processors. There might be other types of resource constraints on the processors, for instance, power consumption or input/output number constraints. With proper abstraction, they can be similarly represented.

Equations (7) to (9) define whether a message is a global message. Equations (10) and (11) set up the end-to-end la-

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

tency constraints on paths. Equations (12) to (15) compute the message latency. Equations (16) to (24) compute the worst case task response time. The typical "big M" formulation in MILP programming is used to linearize the representation (by introducing a large constant M, conditional constraints can be linearized, e.g. either (17) or (18) will take effect depending on the value of o_{τ_i,τ_j} being 1 or 0). u_{τ_i,τ_j} is an integer variable. Note that if τ_i is a sensing or actuating task, the computation of r_{τ_i} becomes trivial. Similarly, for the computation of g_{m_i} and l_{m_i} if the source or destination task of m_i is either a sensing or actuating task. Equations (25) and (26) assure the correct assignment of priorities, where o_{τ_i,τ_j} is 1 if τ_j has a higher priority than τ_i, 0 otherwise. We only explore the priorities for tasks mapped onto processors since sensing and actuating tasks are mapped one-to-one. Equation (27) is the objective function. It does not include the costs of sensors, actuators and communication network. Since we assume sensors and actuators are chosen manually, their costs are not in the objective function. The communication network will be optimized later in the mapping flow, and we do not have an accurate way to estimate its cost at this stage yet. In our future work, we plan to extract high level information of the communication networks and include an abstract model of their cost in the MILP formulation.

By solving the MILP above, we select processors, allocation of tasks and priority assignment of tasks. These will be used for constructing a mapped model, which serves as the input of communication network synthesis.

4.2.2 Communication Network Synthesis

As shown in Figure 6, the communication network synthesis step takes a mapped model $G_C = (V_C, E_C)$ as input, and refines its virtual communication components E_C to a specific network of communication links, routers and repeaters.

We use COSI (Communication Synthesis Infrastructure) [21][2] for our communication network synthesis. The MILP introduced above provides the inputs to COSI. Specifically, the selected computation components and allocated tasks define V_C in graph G_C. Each computation component is labeled with parameters for representing certain characteristics such as cost, physical location, etc. The virtual communication components E_C on which the messages are allocated can be deduced from the MILP results. For two computation components, if there are tasks on them exchanging global messages, a virtual communication component is needed to connect them, and those global messages are naturally allocated to this virtual communication component. The traffic load and latency requirement on each virtual communication component can then be deduced.

[2]For more details of COSI formulation and algorithms, please refer to [21, 19, 22].

4.3 Mapping Case Study

We applied our mapping formulation and algorithm to the room temperature control example shown in Figure 2. To test the scalability of the algorithm, we extended the number of rooms from 3 to more than 40, while keeping the same structure. The building floorplan and physical constraints are from a real office building. The functional model consists of 61 sensing tasks, 1 LQR task, 61 PID tasks and 61 actuating tasks. There are 61 paths from sensing task to LQR to PID then to actuating task. The total number of messages is 183. The architecture platform is characterized in Table 3, part of which is the same as in [19]. We use ARCNET [1] daisy chain buses as communication library.

Table 3. Characterization of a realist architecture library for BAC systems

Component	Performance	Cost
Sensor	Delay: $12.6\mu s$	Price: \$110 Inst: \$50
Actuator	Delay: $12.6\mu s$	Price: \$200 Inst: \$50
Processor1	Speed: $16MHz$ Memory: $512KByte$	Price: \$600 Inst: \$300
Processor2	Speed: $40MHz$ Memory: $3MByte$	Price: \$1400 Inst: \$500
Bus(twisted-pair)	Delay: $5.5ns/m$ Bandwidth: $156Kbps$	Price: $\$0.6/m$ Inst: $\$7/m$
Router	Delay: $320ns$	Price: \$500 Inst: \$240

The MILP is solved using CPLEX 11.0 [4] on a 3.06GHz machine with 3G RAM. The timeout limit is set to 1000 seconds. After the MILP solving, two *Processor1* and one *Processor2* are selected, as shown in Figure 7. The LQR task is mapped to the only *Processor2* which is in the middle of the building floor. All sensors are connected to it since the sensing tasks communicate with LQR task through global messages. The PID tasks are distributed over the three processors, and are connected to actuating tasks, correspondingly. Figure 8 shows the final result after communication network synthesis.

The cost of the final solution breaks down as follows: \$3700 for the processors, \$25010 for sensors and actuators, \$18076.31 for the communication network including wires and routers. As a comparison, if we restrict our selection of processors to type *Processor2*, the cost of processors in the final solution will increase to \$3800 (two *Processor2* are selected), and the cost of the communication network

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

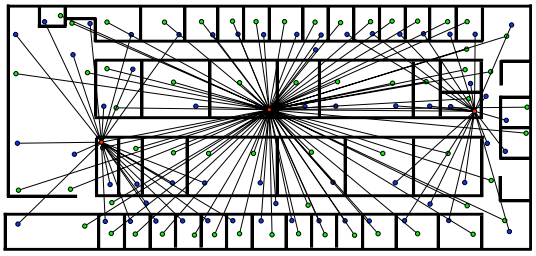

Figure 7. Mapping result after MILP

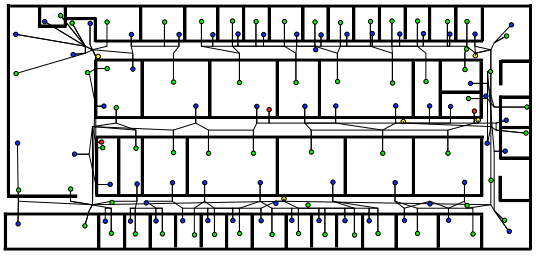

Figure 8. Final mapping result

will increase to $20196.22 (the final layout is not shown here due to page limit). In this particular example, different selections of computation components result in similar cost of processors, however lead to quite different cost of the communication network. This aspect demonstrates the importance of optimizing both computation and communication of the system together.

5 Step 3: Software Synthesis

After the functional model is mapped onto the architecture platform, the next step in the design flow is software synthesis, which includes code generation for each individual processor and the synthesis of communication interfaces between processors.

We leverage vendor tools or general compilers for code generation of individual processors. For instance, LabVIEW provides code generators for a variety of embedded, mobile and touch panel targets. However, most of these tools do not consider the communication semantics between processors and its impact on the generated software. In our work of software synthesis, we focus on synthesizing the communication interfaces of processors after their initial code is generated individually.

The goal of communication interface synthesis is to preserve the semantics of the input functional model when the architecture does not directly support it. A typical case in BAC is that the functional model is synchronous, which eases the design by orthogonalizing functionality and timing, while the architecture platform is distributed and asynchronous. We propose a communication interface synthesis approach to guarantee that the distributed asynchronous implementation has the same behavior as the original synchronous model. Our work extends the methods from [25, 14].

5.1 Communication Interface Synthesis

A method is proposed in [25] to implement synchronous functional models on a Loosely Time Triggered Architecture (LTTA) [11] while preserving stream equivalence. In LTTA, the computation components execute and access the communication medium in a quasi-periodic fashion, i.e. they are triggered periodically by local clocks that are not synchronized but deviate from each other by bounded drift and jitter. The semantic preservation method in [25] guarantees the data value stream on any communication link in LTTA is the same as in the synchronous model. To do so, first the synchronous model is mapped onto an intermediate layer called Finite FIFO Platform (FFP), which consists of a set of sequential processes communicating via bounded FIFO queues. A process skips a round when any of its input queues is empty or any of its output queues is full. By enforcing this, it is guaranteed that there is no data repetition or data loss on the communication flows between processes, and stream equivalence is preserved. Then the FFP model is mapped onto the LTTA platform. Specifically, the FFP queues are implemented as CbS (Communication by Sampling) channels with FFP APIs mapped to CbS APIs. The FFP processes are directly translated to the processes (tasks) on LTTA nodes, only by replacing the APIs of accessing FFP queues with the APIs of CbS.

In the building automation domain, the architectures typically follow the same loosely time triggered paradigm, where periodic sampling from the sensors and discrete-time control are common for applications such as HVAC. Therefore, we can leverage the method from [25] in our communication interface synthesis. However, the assumption that every process (or task after mapped to LTTA) can freely skip a round does not hold in our case if we want to preserve stream equivalence. Specifically, the sensing tasks in the BAC systems periodically sample inputs from the constantly-changing physical environment. Skipping a round on these tasks means the "old" environment inputs will be overwritten by the "new" inputs, and the data stream is no more equivalent to the synchronous model. Therefore, to preserve the synchronous specification, we set the following requirements on the system implementation:

1. A sensing task never skips a round. We assume the sensing tasks are activated periodically according to the local clocks, and send the sampled data in a non-blocking fashion.
2. There is no data loss or repetition on any communication link in the system.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

3. An actuating task can skip a round when it is activated if the input is not ready. However, it has to fire exactly once between any two consecutive fires of its corresponding sensing task to ensure the physical environment is consistent with the synchronous specification, assuming the sensing and actuating task have the same period in the synchronous model (the cases that they have different periods are discussed later). Here we ignore the impact of the exact time point at which the actuation happens between the two firings of the sensing task.

To satisfy requirement 2, we first add the control mechanism from [25] in the implementation of *non-sensing* tasks to allow them skip rounds when their input is not ready or output is full. We assume the CbS channels are implementable on our architecture platform so the tasks can check the availability of inputs/outputs. For discussion on how to implement the CbS primitives, please refer to [28, 27]. In addition, since the sensing tasks cannot skip rounds, we set timing constraints on communication links affected by them to avoid data loss, based on the analysis from [14]. We then further extend the analysis and set additional timing constraints on path latencies for completing the conditions of satisfying requirement 2, and for satisfying requirement 3.

Next we explain how timing constraints are set on communication links and path latencies for meeting the requirements.

5.2 Timing Constraints for Preserving Synchronous Semantics

After mapping, the functional tasks are allocated onto the computation components, which are connected by a communication network that includes communication links, routers and repeaters. For the analysis in this section, we regard computation components, routers and repeaters all as nodes that communicate to each other through communication links.

In a loosely time trigged distributed system, each node has a local clock that triggers all the periodic tasks on that node. For a task τ_i, the n-th tick of interest for the task, denoted by $t_{\tau_i}(n)$, is affected by clock drifts and jitters and can be characterized in Formula (29) and (28), similarly as in [14].

$$t_{\tau_i}(n) \in [\hat{t}_{\tau_i}(n),\ \hat{t}_{\tau_i}(n) + J_{\tau_i}] \tag{28}$$

$$\hat{t}_{\tau_i}(n+1) - \hat{t}_{\tau_i}(n) \in [T^m_{\tau_i},\ T^M_{\tau_i}],$$

$$T^m_{\tau_i} = T_{\tau_i}(1 - \delta^m_{\tau_i}),\ T^M_{\tau_i} = T_{\tau_i}(1 + \delta^M_{\tau_i}) \tag{29}$$

where T_{τ_i} is the reference period of the task, and $\hat{t}_{\tau_i}(n)$ is an auxiliary sequence satisfying the second equation. $\delta^m_{\tau_i} \in [0,\ 1)$ and $\delta^M_{\tau_i} \in [0,\ 1)$ are the relative bounds of the clock drift. We assume all the tasks located at the same node have the same bounds of the clock drift. We use $J^m_{\tau_i}$ and $J^M_{\tau_i}$ to denote the best and worst case of the clock jitter respectively.

To preserve the synchronous semantics, we first set timing constraint on communication links to guarantee there is no data loss (i.e. message being overwritten) when the source task cannot skip rounds, based on the analysis from [12, 14]. Specifically, for a pair of source task τ_w and destination task τ_r communicating through global messages on a communication link, Formula (30) guarantees that any message mg from τ_w is read by τ_r during its valid interval, i.e., from mg arriving at τ_r to it being overwritten by the next message from τ_w.

$$T^M_{\tau_r} + J^M_{\tau_r} < T^m_{\tau_w} + (J^m_{\tau_w} + l^m_{mg}) - (J^M_{\tau_w} + l^M_{mg}) \tag{30}$$

l^m_{mg} and l^M_{mg} are the best and worst case latency of message mg, which can be estimated based on the communication protocol and media. The right hand side is the lower bound of the valid interval. The formula ensures that there is at least one activation of τ_r during the valid interval. No buffer is assumed and extension can be made for the cases with fixed number of buffers.

In our systems, timing constraint (30) first has to be set on all communication links between sensing tasks and their successors since the sensing tasks cannot skip rounds. Furthermore, as the successors need to consume the messages from the sensing tasks in time, they cannot skip freely themselves. This reasoning can be applied to their successors as well. Therefore, a conservative approach is to set constraint (30) on all communication links that are in the "fan out" of the sensing tasks, which can be deduced from the functional model graph. Note that some of the messages between tasks are local messages, for which (30) becomes trivial.

The control mechanism from [25] and timing constraint (30) are not sufficient for preserving the synchronous semantics in our systems. To satisfy requirement 2 and 3, we set additional constraints on path latencies with respect to the local clocks of sensing tasks. First, an actuation decision may require inputs from multiple sensors. In this case, paths from different sensing tasks will converge at a certain task, which reads data from multiple input queues before it can fire. To ensure that the data on one input queue will not be overwritten because of the delay on another input queue, we set the following constraint: For any two sensing tasks τ_i and τ_j whose data is needed at a common task τ_k,

$$\forall n, \quad l_{\tau_i \to \tau_k} < t_{\tau_j}(n+1) - t_{\tau_i}(n) \tag{31}$$

where $l_{\tau_i \to \tau_k}$ is the latency for any path from τ_i to τ_k, $t_{\tau_i}(n)$ is the n-th tick of the local clock for τ_i, and $t_{\tau_j}(n+1)$ is the $(n+1)$-th tick of the local clock for τ_j. Note that if τ_i is not a sensing task but τ_j is, the constraint is still needed (not *vice versa* since non-sensing tasks can skip rounds).

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

In addition to avoiding data loss on communication links, we need to ensure that the actuators can fire in time to impact the physical environment as defined in requirement 3. For this, we set end-to-end latency constraints on paths from sensing tasks to actuating tasks, to make sure the actuators can fire before the next activation of its corresponding sensing tasks. Specifically, for a path ρ in the functional model that contains k_ρ unit-delay tasks (each delays one sampling period of the source sensing task), the end-to-end latency from the sensing task to the actuating task should be bounded as shown in Formula (32), where src_ρ is the source sensing task of the path. The worst case end-to-end latency l_ρ can be computed as in Equation (11). Since software synthesis is done after mapping, we will be able to have an accurate estimation of all the parts in Equation (11), including message latencies.

$$l_\rho < (k_\rho + 1) * T^m_{src_\rho} \tag{32}$$

In the case of no unit delay task, we have a simple constraint that $l_\rho < T^m_{src_\rho}$, and it can be deduced that if this is satisfied, all communication links on ρ satisfy Equation (30).

We have assumed all sensing and actuating tasks have the same period in the functional model. If this is not the case, constraint (32) need to be modified. Assuming on a path, $T_A/T_S = N$, where T_A is the period of the actuating task and T_S is the period of the sensing task (N is an integer), the actuator has to fire exactly once within N fires of the sensing task, i.e., $l_\rho < (k_\rho + 1) * N * T^m_{src_\rho}$.

Given a mapped system, to satisfy constraint (30), (31) and (32), we may need to adjust task periods and the drifts of local clocks. For instance, constraint (31) sets a bound on how much the local clocks of sensors can drift with respect to each other, which can be controlled through the use of synchronization mechanisms between local clocks.

Note that in the mapping step, we carried out the optimization using the task periods specified in the synchronous model, and the preset end-to-end latency constrains without consideration of semantic preservation. If the changes of task periods in software synthesis are significant, the mapping step may be suboptimal. In this case, we can either iterate between these two steps, or add the timing constraints to the mapping formulation and solve everything together (with a trade-off between optimality and complexity). However in real systems, it is common that the clock drifts, jitters and message latencies are all considerably smaller than the sampling periods, therefore the changes on periods in this step will not be too significant.

5.3 Case Study of Communication Interface Synthesis

We conducted experiments on the room temperature control example to demonstrate the idea of communication interface synthesis. We first model a mapped system in LabVIEW with the synthesized communication interfaces, then compare it through simulation to the functional specification in LabVIEW, which is obtained from IF translation as shown in Figure 5.

Specifically, the mapped model in LabVIEW is shown in Figure 9. Each of the PIDs and LQR is mapped to a different processor, abstracted in LabVIEW as a simulation loop that has its own local clock. Each actuating task is also mapped to a separate simulation loop. For simplicity, all sensing tasks are captured in the same simulation loop and assumed to have the same clock. The plant model is also described in this simulation loop and provides data to the sensing tasks. Control mechanism for skipping rounds as in [25] is added to all tasks except the sensing tasks.

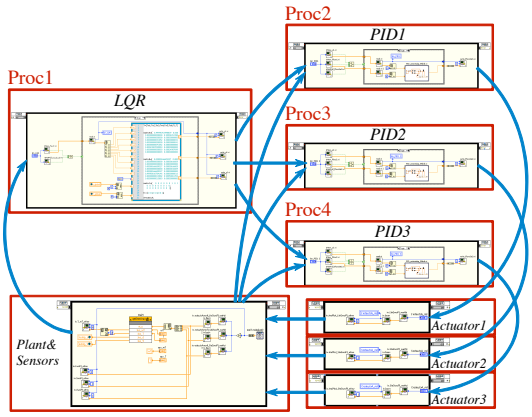

Figure 9. Mapped LabVIEW model for the temperature control system

The local clocks are set to have no clock drifts. In this case, the constraint described in Equation (30) for avoiding data loss can be simplified to Equation (33), given the fact that $l^m_{mg} \geq 0$, and for any task τ_i, $J^m_{\tau_i} \geq 0$ and $J^M_{\tau_i} = r_{\tau_i}$.

$$T_{\tau_r} + r_{\tau_r} < T_{\tau_w} - r_{\tau_w} - l^M_{mg} \tag{33}$$

The communication between processors are through shared variables, therefore $l^M_{mg} = 0$. The response time r_{τ_w} and r_{τ_r} are randomized but smaller than 0.05 second. When we set the periods of sensing tasks, LQR, PIDs and actuating tasks to be 1, 0.5, 0.2, 0.1 second respectively, constraint (33) and (32) hold, and the simulation result of the mapped model is the same as the functional specification. When we gradually reduce all the periods by the same factor, constraint (33) does not always hold, and the simulation results of the two models become different as shown in Table 4. The difference is acceptable though for temperature control system. This shows that for some applications, stream equivalence can be relaxed.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.

Table 4. Comparison of mapped and synchronous models

Sensing Period (s)	1.0	0.5	0.2	0.1
Avg. Differences of Temperature (^oC)	0	2.28 $\times 10^{-4}$	6.83 $\times 10^{-4}$	8.14 $\times 10^{-4}$

6 Conclusions

We proposed a correct-by-construction design flow for automatic deployment of building automation and control systems on distributed platforms that leverages vendor tool chains for code generation. This approach differs significantly from the *ad-hoc* flow used today where control contractors interpret standard sequences of operations and code them directly in a low level language.

In the future, we plan to include more general ERMs and EMs in a library of schedulers to cover a larger set of input specifications. Further, we plan to improve the MILP formulation in the mapping step by including the estimation of communication costs. We also plan to study semantic preservation with fault tolerance.

Acknowledgment

The authors would like to thank Mehdi Maasoumy for providing the functional model of the room temperature control system, Guang Yang and Hugo Andrade for their help on LabVIEW, and Philip Haves for his input to the IF library of HVAC systems.

References

[1] ARCNET. `http://www.arcnet.com`.
[2] ASHRAE. `http://www.ashrae.org`.
[3] EIKON-LogicBuilder for WebCTRL. `http://www.automatedlogic.com`.
[4] ILOG CPLEX Optimizer. `http://www.ilog.com/products/cplex`.
[5] Modelica. `http://www.modelica.org`.
[6] NI LabVIEW. `http://www.ni.com/labview`.
[7] Simulink. `http://www.mathworks.com`.
[8] Metasys GPL Programmer's Mannual, 2004.
[9] User's Guide EIKON for WebCTRL, 2005.
[10] Honeywell Spyder User's Guide, 2007.
[11] A. Benveniste, P. Caspi, M. di Natale, et al. Loosely time-triggered architectures based on communication-by-sampling. In *Proc. of the 7th international conference on embedded software*, pages 231–239, 2007.
[12] A. Benveniste, P. Caspi, P. L. Guernic, et al. A protocol for loosely time-triggered architectures. In *Proc. of the 2nd international conference on embedded software*, pages 252–265, 2002.
[13] A. Davare, Q. Zhu, M. Di Natale, et al. Period optimization for hard real-time distributed automotive systems. In *Proc. of the 44th Design Automation Conference*, pages 278–283, June 2007.
[14] M. Di Natale, A. Benveniste, P. Caspi, et al. Applying ltta to guarantee flow of data requirements in distributed systems using controller area networks. *Proc. of the Design, Automation and Test in Europe Workshop Dependable Software Systems*, 2008.
[15] E. A. Lee and T. Parks. Dataflow process networks. In *Proc. of the IEEE*, pages 773–799, 1995.
[16] J. Lygeros, C. Tomlin, and S. Sastry. Controllers for reachability specifications for hybrid systems. *Automatica*, 35:349–370, 1999.
[17] M. Natale, W. Zheng, C. Pinello, et al. Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems. In *Proc. of the 13th Real Time and Embedded Technology and Applications Symposium*, pages 293–302, April 2007.
[18] National Science and Technology Council, Committee on Technology. Federal research and development agenda for net-zero energy, high-performance green buildings. 2008.
[19] A. Pinto, L. Carloni, and A. Sangiovanni-Vincentelli. A communication synthesis infrastructure for heterogeneous networked control systems and its application to building automation and control. In *Proc. of the 7th international conference on embedded software*, 2007.
[20] A. Pinto, L. P. Carloni, R. Passerone, et al. Interchange format for hybrid systems: Abstract semantics. In *Proc. of Hybrid Systems: Computation and Control, 9th International Workshop*, pages 491–506, 2006.
[21] A. Pinto, L. P. Carloni, and A. L. S. Vincentelli. Cosi: A framework for the design of interconnection networks. *IEEE Design and Test of Computers*, 25(5), 2008.
[22] A. Pinto, M. D'Angelo, C. Fischione, et al. Synthesis of embedded networks for building automation and control. In *Proc. of American Control Conference*, 2008.
[23] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, et al. Interchange formats for hybrid systems: Review and proposal. In *Proc. of Hybrid Systems: Computation and Control, 8th International Workshop*, pages 526–541, 2005.
[24] P. Torcellini, S. Pless, and M. Deru. Zero energy buildings: A critical look at the definition. *ACEEE Summer Study on Energy Efficiency in Buildings*, June 2006.
[25] S. Tripakis, C. Pinello, A. Benveniste, et al. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. Comput.*, 57(10):1300–1314, 2008.
[26] M. Wetter and P. Haves. Modelica library for building hvac and control systems. https://gaia.lbl.gov/bir.
[27] F. Xia, F. Hao, I. Clark, et al. Buffered asynchronous communication mechanisms. *Fundam. Inf.*, 70(1):155–170, 2005.
[28] F. Xia, A. V. Yakovlev, I. G. Clark, et al. Data communication in systems with heterogeneous timing. *IEEE Micro*, 22(6):58–69, 2002.
[29] Q. Zhu, Y. Yang, E. Scholte, et al. Optimizing extensibility in hard real-time distributed systems. In *Proc. of the 15th Real-Time and Embedded Technology and Applications Symposium*, pages 275–284, 2009.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on August 09,2022 at 09:34:08 UTC from IEEE Xplore. Restrictions apply.