Software Synthesis for Distributed Embedded Systems

Yang Yang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-60
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-60.html

May 4, 2012

Software Synthesis for Distributed Embedded Systems

by

Yang Yang

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Spring 2012

Software Synthesis for Distributed Embedded Systems

Abstract

Software Synthesis for Distributed Embedded Systems

by

Yang Yang

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

The amount and complexity of software in embedded control systems is increasing rapidly. This factor, together with the wide use of distributed platforms and the tight design requirements, raises great challenges to software design and development in these systems. However, the current design practice is largely manual and ad-hoc, especially at the system level, which produces suboptimal and unreliable systems. In this dissertation, we propose a systematic software synthesis flow to address some of the pressing issues in software design, in particular the heterogenity of the design inputs, the complexity of the design space, and the semantic difference between the functional specification and the implementation platform.

The flow consists of a front-end that translates heterogeneous input specification into a unified representation, and a back-end that conducts automatic design space exploration and code generation. We define an intermediate format (IF) as the unified representation, and develop translators from input models to IF and from IF to output code. We design algorithms to explore the design space during mapping from the functional specification to the architectural platform, with respect to design metrics such as cost, latency and extensibility. We also propose approaches to synthesize the communication interfaces between software tasks to guarantee the semantic equivalence of the distributed implementation with respect to the synchronous specification.

The applicability of the synthesis flow is illustrated with case studies from the building automation and automotive domains. The results showed that the flow can be effectively applied to widely different applications in different domains.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to first thank my advisor Prof. Alberto Sangiovanni-Vincentelli for his guidance and support throughout my PhD study. Alberto is a great mentor with incredible knowledge and insights on many research areas and industries. During my research, he is always a great source of new ideas and motivations. Many of the ideas in this dissertation came from the discussions with him. His emphasis on both rigorous theoretical foundation and potential practical applications has greatly influenced my research, and will continue to guide me in my future work. Alberto also cares deeply for his students. Over the years, he has been very supportive in both my study and my life. For this, I will be forever grateful.

I would like to thank Prof. Sanjit Seshia and Prof. Francesco Borrelli for taking time to review my dissertation. I also want to thank them and Prof. Jan Rabaey for being on my qualifying exam committee. Their acute comments and suggestions help shaping and improving this work. I had the opportunities to work with a group of wonderful researchers from academia and industry, and I would like to thank their help and mentorship. They include: Alessandro Pinto, Eelco Scholte, Guido Poncia, Hugo Andrade, Marco Di Natale, Mehdi Maasoumy, Michael Wetter, Philip Haves, Qi Zhu and Stavros Tripakis. In particular, Stavros Tripakis, Alessandro Pinto and Qi Zhu were heavily involved in the research work presented in this dissertation. Their contributions are instrumental and I have learned a lot from them through the collaboration.

During my study at Berkeley, I have encountered many great professors and taken many excellent courses. In particular, I would like to thank Prof. Kurt Keutzer for reading my Masters report and providing valuable feedback. I would like to thank Prof. Andreas Kuehlmann for his logic synthesis class. It was one of the hardest courses that I took and also one of the most satisfying courses after I completed it.

I am also fortunate to interact with a group of talented colleagues and friends. They include, but not limited to: Bryan Brady, Bryan Catanzaro, Donald Chai, Satrajit Chatterjee, Minghua Chen, Jike Chong, Abhijit Davare, Douglas Densmore, Yitao Duan, Thomas Feng, Shanna-Shaye Forbes, Liangpeng Guo, Dan Holcomb, Dan Huang, Ling Huang, Shinjiro Kakita, Nathan Kitchen, Yanmei Li, Wenchao Li, Chung-Wei Lin, Cong Liu, Kelvin Lwin, Mark McKelvin, Pierluigi Nuzzo, Trevor Meyerowitz, Alberto Puggelli, Kaushik Ravindran, Alena Samalatsar, Baruch Sterin, Jimmy Su, Xuening Sun, Guoqiang Wang, Lynn Wang, Zile Wei, Tobias Welp, Wei Xu, Guang Yang, Jing Yang, Haibo Zeng, Wei Zheng, Feng Zhou, Li Zhuang, and Jia Zou.

Last but not least, I would like to thank my parents, my husband and my daughter. Without them, I would not have accomplished my goal. My parents love me with all their hearts, and always support me unconditionally. I will forever be in their debt. My daughter Olivia is the sweetest girl in the world and the best thing happened to me. She makes every day exciting and bright. Even before she could speak, she was already one of my biggest supporters. At many difficult times, she gives me the comfort and courage that I need. My husband Qi is the most important person in my life. He is so kind, caring, loving and generous. He is always there with me for every important decision I have made, and supports me through my ups and downs. He sits down and brainstorms with me whenever I meet difficulties in my research and life. Now he is pursing his goals in academia, I wish him the best.

Chapter 1

Introduction

Embedded systems have become ubiquitous in everyday life and are fastly growing, with wide applications in domains such as consumer electronics, automotive, aerospace, civil infrastructure, medical devices and industrial automation. According to International Data Corporation (IDC), the market for embedded systems will double in size over the next four years, with estimately more than 4 billion units shipped and $2 trillion in revenue [32]. As the system complexity increasing rapidly in terms of both scale and functionality, many of the modern embedded systems are deployed on spatially-distributed and networked platforms. These distributed embedded systems consist of a network of embedded processors (ranging from tens to hundreds or even more) connected through wired buses or wireless communication. For instance, regular modern vehicles typically employ 50 to 70 electronic control units (ECUs, luxury cars may have up to 100 ECUs) and a number of buses.

Another major trend is the rapid growing of software in embedded systems, in terms of both amount and complexity. Using automotive domain as an example, an average car in 2000 contains one million lines of code, with software taking 2% of the total value of the car; while an average car in 2010 has 100 millions of code and software takes 13% of the car value. It is predicted that more than 80% of car innovations in future will come from computer systems, and software will become major contributor of value [66].

In addition, many complex distributed embedded systems with time and (possibly) reliability constraints are today the result of the integration of components or subsystems provided by various suppliers in large quantities. For example, production quantities for automotive subsystems are in the range of hundreds of thousands. Many avionics systems, home automation systems (HVAC controls, fire and security), and other control systems (for example elevators and industrial refrigeration systems) share similar characteristics and models of supply chain. All these systems are characterized by the need of a careful design and deployment of system-level functions, given the need to satisfy real-time constraints and cope with tight resource requirements because of cost constraints.

The above factors – increasing software complexity, employment of highly-distributed platform, integration of heterogeneous components or subsystems, and tight design constraints – propose major challenges to the design of distributed embedded systems. In current practice, much of the design process is manual and ad-hoc, with different subsystems designed in isolation. This leads to suboptimal and unreliable systems. In automotive domain, vehicle recalls related to electronic systems have tripled in past 30 years [24], and now 50% of the warranty costs are related to electronics and software [66]. In avionics domain, software complexity and integration challenge are major contributors to the production delay of Boeing 787 Dreamliner [23].

In this dissertation, we focus on the problem of *implementing a given embedded application on a distributed platform with respect to a set of design objectives and constraints.* We target typical embedded control systems in domains such as automotive, avionics and building automation, where the embedded applications are control algorithms representing the system functionality and the distributed platforms consist of sensors, embedded processors, actuators and communication buses. During the design process, the control algorithms that are initially described by high-level modeling languages will be implemented as software code on the distributed embedded platform. To address the design challenges mentioned above, we propose a systematic software synthesis flow that focuses on two main aspects – *integration* and *automation*. The flow bridges the gap between a desirable design entry point – at a high abstraction level using model-based design tools such as Simulink [7] and Modelica [6]– and the available back-end tools able to generate low-level code. The flow enables the *integration* of heterogeneous input models from different high-level languages, allowing the interaction between domain experts and designers of different subsystems. It also *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. This automation of design space exploration and code synthesis makes it possible to cope with the complexity from increasing software content, highly distributed platform and tight design requirements.

1.1 A Systematic Software Synthesis Flow

Our proposed software synthesis flow consists of a front-end and a back-end, as shown in Figure 1.1. 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 front-end and the back-end exchange models using an intermediate format (IF). The introduction of this intermediate layer is essential for the integration of heterogeneous inputs, the leverage of back-end tools, and automatic design space exploration. It enables building a software synthesis flow that is general with respect to the user input (e.g. Simulink and Modelica), and to the output implementation code (e.g. C and domain-specific language such as EIKON [5] for building automation and control). 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 by allowing them exchange models

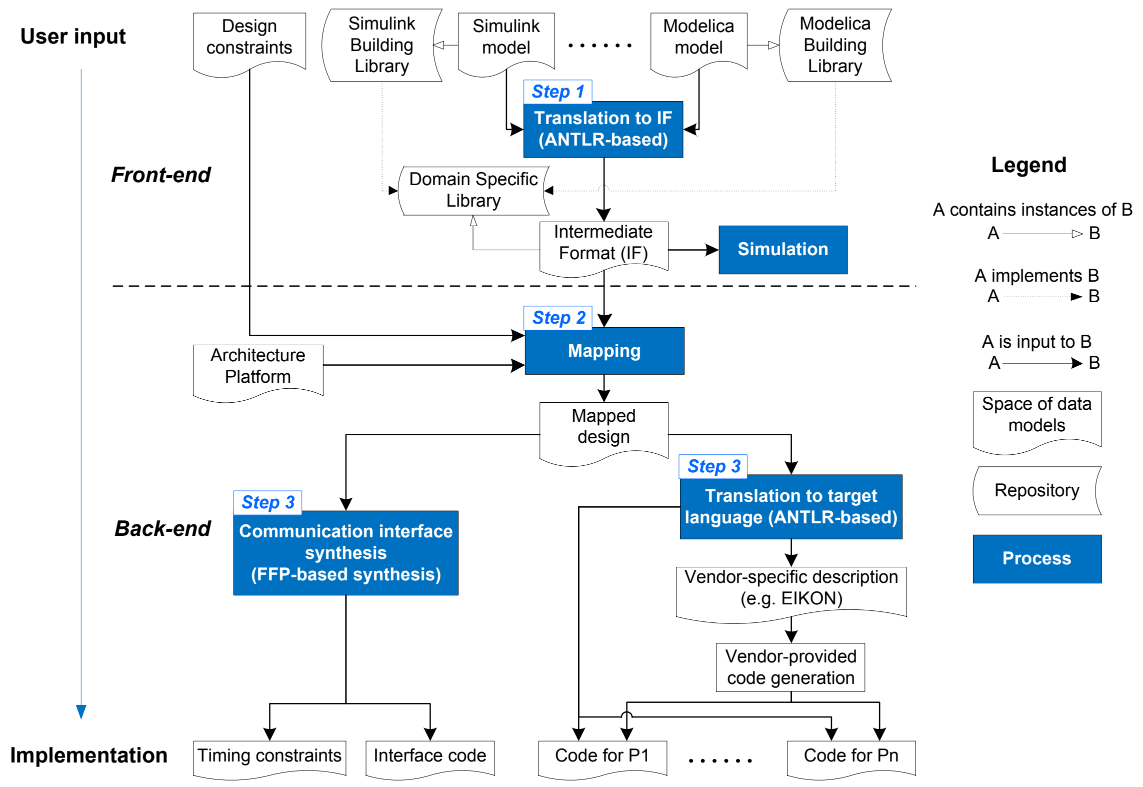


Figure 1.1: Software synthesis flow for distributed embedded systems

and evaluate designs taking into account the interactions with other subsystems. The intermediate level also allows targeting multiple implementation platforms. Embedded 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 IF into the vendor specific language. Compared to providing customized software synthesis flows from each high-level language to each architecture specific language, using IF reduces the number of translators needed from a quadratic number to a linear number. We define IF based on the Metropolis Meta Model (MMM) semantics and the nomenclature introduced in [56]. The denotational IF representation can be further translated into an executable model in the METROPOLISII framework and simulated.

In *Step 1* of our software synthesis flow, input functional models are translated into the IF representation through automatic or manual translation. As a proof of concept, we developed a translator based on ANTLR (ANother Tool for Language Recognition) [1] for automatically translating Modelica models into IF. 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 domain-specific *libraries* of primitives at the intermediate level designed to capture a large class of control algorithms in corresponding domain and that can be extended by users. The domain-specific libraries are then mirrored by equivalent libraries defined in the source languages. The set of models that can be translated into the IF is the one obtained as composition of the library elements. This architecture simplifies the translation process and will be described later in the context of building automation and control systems.

Besides serving as the intermediate representation between input models and target implementation, IF also provides the functional abstraction for automatic design space exploration. In *Step 2*, the back-end automatically *maps* the functional model described in the IF to the architectural model that captures the implementation platform. The part of the functional model to be mapped is the 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, resource utilization 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 impact of the surrounding physical environment and the related mechanical components is abstracted into a set of physical constraints imposed on the system. There may also be other types of design constrains based on functional requirements or architectural limitation. The mapping problem is then 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. *Step 3* of the software synthesis flow conducts synthesis starting from the mapped design. The synthesis process includes code generation for individual processors 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 or a vendor specific language. As a demonstration, in our case study in building automation and control domain, we developed a translator for translating IF into the EIKON language based on ANTLR. 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, we propose a communication interface synthesis approach to ensure the preservation of synchronous functionality on asynchronous platforms. The approach includes two main aspects: interface synthesis to guarantee stream equivalence on distributed electronic systems while optimizing communication load, and adding timing constraints to preserve the semantics with consideration of the interaction with physical environment.

1.2 Application Domains

We believe our software synthesis flow can be applied to a variety of application domains. In this work, we choose automotive domain and building automation and control domain as our focus. Automotive systems is an epitome of complex distributed embedded systems, and as shown earlier, it is in great need of more formal and automatic methodologies. Advances in automotive domain is also crucial to a successful manufacturing industry, which in turn is a key part (and fundamental part) of the economy.

Building automation and control (BAC) is another important area that is critical to the economy and environment. The building stock in the US accounts for 40% of total energy consumption and 70% of electricity consumption [51]. 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 [3]. 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 [67], 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 BAC 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. It is essential to develop new design methodologies for such complex systems, and our proposed software synthesis flow is designed to tackle these challenges.

Our case studies in automotive domain mostly focus on the mapping between functional model and architectural platform (Step 2 in the software synthesis flow), with consideration of two important design objectives – reducing the end-to-end latency along functional paths and improving the system extensibility. Our case studies in BAC systems follow the entire software synthesis flow, including the translation to IF (Step 1), the mapping between functional and architectural model (Step 2, with explicit consideration of physical constraints), and the code generation with semantic preservation (Step 3).

1.3 Related Work

The proposed software synthesis flow is based on the fundamental paradigm of platform-based design (PBD) [41, 26, 64], where the functionality of the design (what the system is supposed to do) and the architectural platform (how the system is implemented) are initially captured separately, and then brought together through the mapping process. By separating functionality and architecture and applying mapping across multiple abstraction levels during the design process, platform-based design facilitates design reuse and reduces design complexity. Step 1 of our synthesis process can in fact be viewed as a mapping process from the initial heterogeneous input models to the unified IF. Step 2 solves the mapping from the functional model in IF to an architectural model at the *system level*. In our work, this is the level that we conduct most of our design space exploration, which determines the system performance and cost. Finally, step 3 can also be viewed as a mapping process but at lower levels of abstraction – it maps the software model on each individual processor to the final code implementation. For the sake of brevity, the term mapping in the rest of the dissertation refers to mapping at the step 2.

For each of the steps in our synthesis flow, there has been extensive work in the literature. In step 1, the intermediate format is closely related to the concept of interchange formats, which have been the backbone of the EDA industry for several years. They enable the development of design flows that integrate foreign tools using formats with different syntax and semantics. In the U.S. the DARPA MoBIES project had the importance of an interchange format very clear and supported the development of HSIF [8]. However, limitations to its semantics make the data interchange between foreign tools difficult. For example, HSIF does not support some of the features of Simulink models. In our opinion, HSIF is an excellent model for supporting clean design of hybrid systems but not yet a true interchange format. Simulink internal format could be a de facto standard however it is not open, nor it has features that favor easy import and export. Modelica has full support of hierarchy and of general semantics that subsumes most if not all existing languages and tools. As such, it is an excellent candidate but it is not open. In addition, all of the above solutions have not been developed with the goal of supporting heterogeneous implementations. The intermediate format based on the Metropolis metamodel (MMM) we plan to use was proposed in [56, 54]. It has abstract semantics and can accommodate the translation to and from various formats of the foreign tools. Besides, due to the Metropolis metamodel, it has generality and can be used to represent a very wide class of models of computation.

For step 2, the mapping problems have been studied in CAN-bus based systems used for automotive and avionics applications, with latency as the main design objective. Because of its low cost and high reliability, the CAN bus is a quite popular solution for automotive systems, and also used in avionics systems as an auxiliary sensor and actuator bus, as well as used in home automation, refrigeration systems and elevators. For distributed systems with end-to-end latency deadlines, the optimization problem was partially addressed in [62], where genetic algorithms were used for optimizing priority and period assignments with respect to a number of constraints, including end-to-end deadlines and jitter. In [60] a design optimization heuristics-based algorithm for mixed time-triggered and event-triggered systems was proposed. The algorithm, however, assumed that nodes are synchronized. In [48], a SAT-based approach for task and message placement was proposed. The method provided optimal solutions to the placement and priority assignment. However, it did not consider signal packing. In [33], task and message periods are explored in an algorithm based on geometric programming to satisfy latency constraints. In [50], the trade-offs between the purely periodic and the data-driven activation models are leveraged to meet the latency requirements of distributed vehicle functions. In [75, 79], task allocation, signal packing and message allocation, as well as task and message priority are explored to optimize the end-to-end path latencies.

In our work for automotive systems, besides the traditional design objectives such as latency and utilization, we also optimize the metric of task extensibility, which measures how much the task execution time may be increased without violating design constraints. The literature on extensibility is rich. Sensitivity analysis was studied for priority-based scheduled distributed systems [62], with respect to end-to-end deadlines. The evaluation of extensibility with respect to changes in task execution times, when the system is characterized by end-to-end deadlines, was studied in [74]. The notion of robustness under reduced system load was defined and analyzed in [49], for both preemptive and non-preemptive systems. The paper highlights possible anomalies (increased response times for shorter task execution times) that would make evaluation of extensibility quite complex. These papers do not explicitly address system optimization. Task allocation, priorities, and message configuration, are assumed as given. Also, it is worth mentioning that time anomalies such as those in [49] and other described in several other papers on multiprocessor and distributed scheduling do not occur for the scheduling and information propagation model we consider. This is because we assume local scheduling by preemption, the passing of information by periodic sampling and the periodic (not event-based) activation of each task and message. This decouples the scheduling of each task and message from predecessors and successors as well as from scheduling on other resources and avoids anomalies. In [15], task allocation and priority assignment are defined with the purpose of optimizing the extensibility with respect to changes in task computation times. The proposed solution is based on simulated annealing, and the maximum amount of change that can be tolerated in the task execution times without missing end-to-end deadlines is computed by scaling all task times by a constant factor. A model of event-based activation for task and messages is assumed. In [37, 39, 38], a generalized definition of extensibility on multiple dimensions (including changes in the execution times of tasks as in our definition, but also period speed-ups and possibly other metrics) is presented. A randomized optimization procedure based on a genetic algorithm is proposed to solve the optimization problem. These papers focus on the multi-parameter Pareto optimization, and how to discriminate the set of optimal solutions. The main limitation of this approach is complexity and expected running time of the genetic optimization algorithm. In addition, randomized optimization algorithms are difficult to control and give no guarantee on the quality of the obtained solution. Indeed, in the cited papers, the use of genetic optimization is only demonstrated for small sample cases. In [39], the experiments show the optimization of a sample system with 9 tasks and 6 messages. The search space consists of the priority assignments on all processors and on the interconnecting bus. Hence, task allocation (possibly the most complex step) and signal to message packing are not subject to optimization. Yet, a complete robustness optimization takes approximately 900 and 3000 seconds for the two-dimensional and three-dimensional case, respectively. In general, the computation time required by randomized optimization approaches for large and complex problems may easily be an issue. In [37] a larger set of "20 tasks and message" is considered, with only priority assignment subject to optimization. These results albeit important in their own right, exhibit a running time that is clearly infeasible for an effective design space exploration. This observation motivated us to develop a two-stage "deterministic" algorithm that has running times over an order of magnitude faster than the ones proposed so far in the literature, as explained later in Chapter 3.

For step 3, the main challenge is to preserve the semantics during code generation with minimal cost. There is a large body of research on distribution of synchronous models, and in particular synchronous languages [20]. For instance, a dynamic buffering protocol is proposed in [65] for preservation of semantics when distributing a synchronous model to multiple tasks running under preemptive scheduling. The buffering protocol can be used for inter task communication inside a processor. A mechanism to distribute a synchronous model to Loosely Time Triggered Architecture(LTTA) is proposed and proved to guarantee semantics preservation in [69]. Other than synchronous programs, there has also been research on code generation for other models of computation. Software synthesis method proposed in [13] focuses on distributing a global asynchronous local synchronous (GALS) network of CFSMs and proposes a method to generate RTOS for communication among CFSMs. In [80], a general execution strategy is defined for executing discrete event (DE) semantics on distributed platform, by ensuring each actor processes input events in time-stamp order.

In our work, we focus on code generation for synchronous models. We leverage the work from [69], which defines an intermediate layer called Finite FIFO Platform (FFP) to facilitate the distribution of synchronous models on LTTA platforms. The FFP platform makes no assumptions on clock synchronization. This has the advantage of providing implementations that are robust to various types of timing uncertainties such as clock drifts and network delays. Similar techniques are used in the design of digital circuits, in particular, latency-insensitive or elastic circuits [27, 28]. On the other hand, knowledge about the timing characteristics of the execution platform may sometimes be available, e.g. bounds on clock drifts and network delays. Implementation techniques that leverage such type of knowledge can be found in [63, 29, 17]. There are also studies that target synchronous distributed execution platforms such as the Time-Triggered Architecture [43]. In that case, one of the main challenges is to synthesize time-triggered communication schedules so that semantics is preserved [30]. We further develop an approach to optimize the communication load while conducting code generation for a particular set of synchronous models – the Triggered Synchronous Block Diagrams (SBDs). The model of Triggered SBDs is directly inspired by tools such as Simulink and SCADE. SCADE can be seen as a subclass of Lustre [31]. Since we only target a restricted class of synchronous models, we avoid many of the difficulties encountered when considering more general models, such as the full Lustre, Signal or Esterel synchronous languages, for which there exists a wealth of techniques [36, 18, 63, 61, 16]. Furthermore, for this communication optimization work, we assume a one-to-one mapping between blocks of the synchronous model and processes of the distributed architecture. This simplifies the problem and allows focusing on semantical preservation. How to allocate functional blocks to processes is an important and difficult problem in embedded control systems, that often involves multi-criteria optimization and tradeoffs, e.g. see [68, 59, 58, 75].

Compared to the above related work, this dissertation makes novel contributions in the following areas.

- We propose a *complete* software synthesis flow for distributed embedded systems, and apply it to the building automation and control systems. Starting from a high level model-based input specification, our flow generates a unified IF representation for facilitating the following synthesis steps, conducts design space exploration through mapping algorithm, and performs code generation on distributed platforms with semantic preservation and communication optimization. As far as we know, this is the first attempt in developing a synthesis flow from model-based specification to embedded implementation for building control systems.

- We develop an automatic translator from Modelica input model to IF at the front-end and an automatic translator from IF to EIKON embedded language at the back-end. Both translators are implemented using the ANTLR framework. The mapping step and the code generation step are automated by algorithms. Together they demonstrate, as a proof-of-concept, an automated flow from input to implementation.

- In the mapping step, we extend the traditional concept of mapping a given functional model to a given architectural platform to include the exploration of the architectural platform. This in principle provides more optimization opportunities. We also define a novel metric and algorithm for optimizing the system extensibility in mapping.

- In the code generation step, we extend the previous work on semantic-preserving synthesis for LTTA to include the consideration of physical environment (therefore additional timing constraints) and the optimization of communication load for Triggered SBDs.

- We leverage the METROPOLISII framework as an executable IF for simulation-based analysis and validation, in particular we apply it to a heterogeneous input model for building control systems.

In rest of the dissertation, the three steps in the software synthesis flow will be introduced in Chapter 2, Chapter 3 and Chapter 4, respectively. Case studies are presented in each of the chapters, including temperature control systems in building automation and control as well as automotive safety and control systems . Chapter 5 concludes our current work and discusses the future directions.

Chapter 2

Intermediate Format

In Step 1 of our software synthesis flow, models capturing the specification of the control algorithms and of the environment are translated into an intermediate format (IF) that is defined to facilitate the other steps in the synthesis flow, namely mapping and code generation. Because the type of specifications that we are interested in are in general hybrid systems [46] with multiple semantics, the IF representation may become very complex [56, 54], 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 our work that is a first step towards the ideal scenario, we restrict the IF to dataflow semantics [44] which is amenable to efficient mapping and code generation. We base our IF definition on the Metropolis Meta Model (MMM) semantics and retain the nomenclature introduced in [56] as we plan to extend this work to more general intermediate representations. In particular, *processes* (also called *actors*) are the basic entities for specification. They are categorized into continuous processes and discrete processes. Each process is defined by a set of *parameters*, *ports* and *equations*. Parameters are set for configuring the process. Ports constitute the communication interface of the process and can be either input or output port. Equations capture the behavior of the process in the form of an input-output function. Multiple processes may be connected through *channels* to form a *netlist* at the higher level and eventually build the entire system. During execution, the equations in the processes 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).

2.1 IF Translation and IF library

The translation process may be done manually or through automatic translators. We developed a translator for Modelica based on the ANTLR framework, a parser generator that uses *LL(*)* parsing. We chose ANTLR because it provides comprehensive support and consistent syntax for specifying lexers, parsers and tree parsers, and supports generating code in common languages such as C, Java, Ada and Objective-C. The translator we developed includes a lexer and a parser for parsing the Modelica language (currently without full support of inheritance and algorithm), and a code generator for generating IF. The following sample code snippet shows part of the ANTLR input for generating the lexer, parser and generator for the *process* entity.

```
class_specifier
  : id=IDENT
  {
    if(classKind.equals("class") || classKind.equals("model")) {
      if(ifAtomic) {
        /// transform to a process in IF
        System.out.println("process " + $id.text + " {");
      } else {
        /// transform to a netlist in IF
        System.out.println("netlist " + $id.text + " {");
      }
      $class_definition::mdl = new Model();
      modelLib.put($id.text, $class_definition::mdl);
      $class_definition::connectionlist_lib
        = new HashMap<String, Set<ConnectionTriple>>();
      $class_definition::connectionlist_lib_keytodelete
        = new ArrayList<String>();
    }

    if(classKind.equals("connector")) {
      /// transform to a class in IF
      System.out.println("class " + $id.text + " {");
      $class_definition::con = new Connector();
      $class_definition::con.id = $id.text;
      $class_definition::con.hasMember = true;
      connectorLib.put($id.text, $class_definition::con);
    }
  }
  composition
  'end'
  {
    if($class_definition::connectionlist_lib_keytodelete != null) {
      Iterator<String> itr1
        = $class_definition::connectionlist_lib_keytodelete.iterator();
      while(itr1.hasNext()) {
        String element = itr1.next();
        $class_definition::connectionlist_lib.put(element, null);
        $class_definition::connectionlist_lib.remove(element);
      }
    }

    /// generate equations for flow memebers from the connection lists
    if($class_definition::connectionlist_lib!= null) {
      System.out.println("equations{ ");
      Iterator iter
        = $class_definition::connectionlist_lib.keySet().iterator();
      while (iter.hasNext()) {
        String key = iter.next().toString();
        Set<ConnectionTriple> connectionlist
          = $class_definition::connectionlist_lib.get(key);
        Iterator iter_cl;
        if(connectionlist.size() == 1) {
          iter_cl = connectionlist.iterator();
          if(iter_cl.hasNext()) {
            ConnectionTriple ctl = (ConnectionTriple)iter_cl.next();
            if(ctl.ifInside == false) {
              continue;
            }
          }
        }
        iter_cl = connectionlist.iterator();
        List<String> flow_member_list = null;
        if(iter_cl.hasNext()) {
          ConnectionTriple ctl = (ConnectionTriple)iter_cl.next();
          flow_member_list = ctl.coj.con.flow_members;
        }
        if(flow_member_list != null) {
          Iterator iter_fml = flow_member_list.iterator();
          while(iter_fml.hasNext()) {
            String flow_member = (String)iter_fml.next();
            iter_cl = connectionlist.iterator();
            if(iter_cl.hasNext()) {
              ConnectionTriple ctl = (ConnectionTriple)iter_cl.next();
              System.out.print(ctl.coj.obj + "." + flow_member);
            }
            while(iter_cl.hasNext()) {
              ConnectionTriple ctl = (ConnectionTriple)iter_cl.next();
              System.out.print(" + " + ctl.coj.obj + "." + flow_member);
            }
            System.out.println(" = 0;");
          }
        }
      }
      System.out.println("}");
    }
    System.out.println("}");
    System.out.println();
  }
  IDENT
  | IDENT '=' base_prefix name ( array_subscripts )?
              ( class_modification )?
  {
    if(classKind.equals("type")) {
      Type t = new Type();
      t.prefix = $base_prefix.text;
      t.specifier = $name.text;
      typeLib.put($IDENT.text, t);
    }
    if(classKind.equals("connector")) {
      System.out.println("class " + $IDENT.text + " extend "
                         + $base_prefix.text + " " + $name.text +" { }");
      $class_definition::con = new Connector();
      $class_definition::con.id = $id.text;
      $class_definition::con.hasMember = false;
      if($base_prefix.text.equals("flow")) {
        $class_definition::con.ifFlow = true;
      } else {
        $class_definition::con.ifFlow = false;
      }
      connectorLib.put($id.text, $class_definition::con);
    }
  }
;
```

Domain specific libraries can be used to enable fast translations to (and from) IF. As an example, we define a domain specific IF library for HVAC control systems in buildings, and we export the library to different specification languages. We reviewed 71 HVAC-related component models in the GPL language from Johnson Controls [9], 70 in Automated Logic EIKON language [10], 42 in Honeywell Spyder [11], and 59 in the HVAC library defined by the Lawrence Berkeley National Laboratory [70]. 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:

```
process PID extends CTProcess{
  parameter real kp, ki, kd, kc;
  parameter real lb, ub;
  parameter real eps;
  input port CTInterface real uset, u;
  output port CTInterface real out;
  equations{
    err = uset - u;
    sum = kp*err + ki*(intg(err))
      + kc*(out-sum)) + kd*deri(err);
    out = (sum>=lb+eps && sum<=ub-eps)*sum
      + (sum<lb+eps)*lb
      + (sum>ub-eps)*ub;
  }
}
```

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. The input ports u and uset indicate process variable and the desired setpoint, and the output port out is the controller output. The tuning parameters kp, ki, kd, kc indicate the proportional gain, integral gain, derivative gain, and anti-windup compensation respectively. The parameters lb and ub are the lower and upper bounds of the controller output.

A component may have slightly different implementations in different languages. 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 may 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.

The above IF representation is denotational. We may further translate it to an executable IF for simulation-based model validation and exploration. In particular, we choose

[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.

the MetropolisII framework [14] for modeling and simulating the executable IF, because its semantics also derives from the Metropolis Meta Model (same as the denotational IF) and it provides strong support on modeling heterogeneous systems. The translation from denotational IF to MetropolisII model is straightforward because of the similarity of their semantics. Processes in IF are translated to *components* in MetropolisII, with equations translated to *constraints*. The ERM and EMs in IF are translated to *constraint solvers* and *schedulers*, which govern the resolution and scheduling of constraints in a three-phase execution semantics.

2.2 Case Study

We conducted a case study on a room temperature control system example to illustrate the application of our software synthesis flow in building automation and control systems. This example will be used throughout the dissertation for each of the three steps in the flow. In this chapter, we will first show how the design input, a heterogeneous functional model of the temperature control system, is translated into an IF representation.

The functional model captures a two-level control algorithm as shown in Figure 2.1. The higher level LQR (linear-quadratic regulator) controller determines the set points for lower level PID controllers. The LQR coordinates among multiple rooms (three rooms in the example) to optimize the total energy consumption while maintain a certain comfort level. The PIDs track the set points and interact with the physical environment. The inputs to the plant model include the air mass flow into each thermal zone, and the outputs are the temperature of each thermal zone and the temperature of walls. More details on the model and control can be found in [47]. As a *heterogeneous design input*, the controller (including PIDs and LQR) is modeled in Simulink, while the plant is modeled in Modelica.

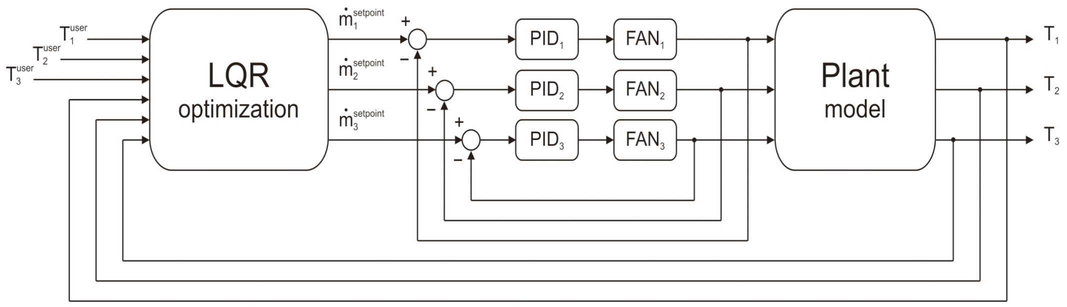
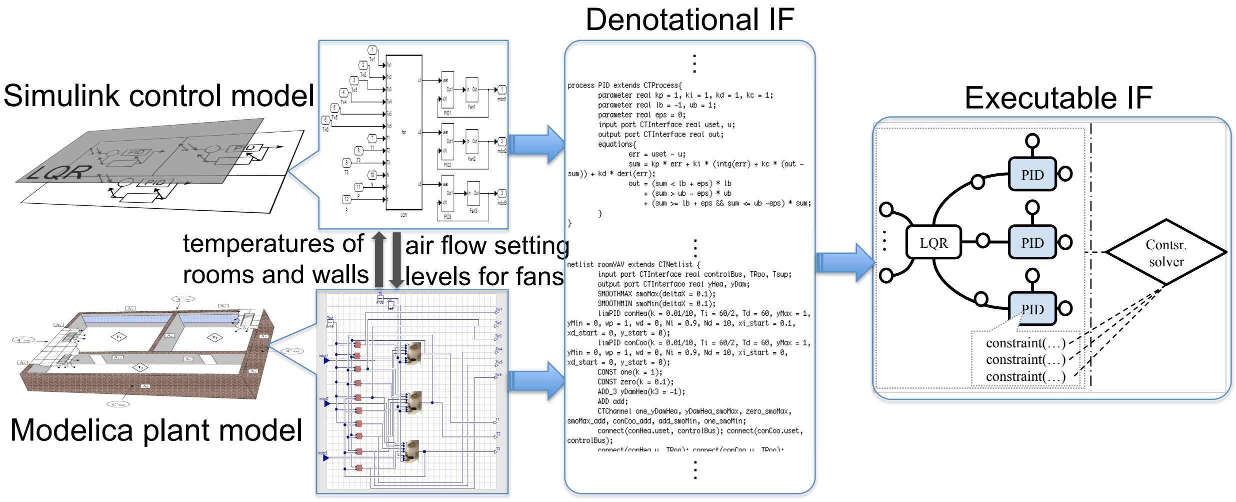


Figure 2.1: Room temperature control system

2.2.1 Translation to Executable IF and IF Simulation

In the first step of our software synthesis flow, the heterogeneous input model is translated into a uniformed IF representation, as shown in Figure 2.2. The Simulink controller model is translated into IF manually, with one-to-one correspondence between the components in Simulink and the processes in the IF representation. The resulted IF includes one LQR process and three PID processes. The scheduling in Simulink is based on the causality relation between components, which is translated to the ERM scheduling in the IF. The Runge-Kutta ODE solver used in Simulink employs a fixed time step and is translated to the scheduling in ERM and EMs in IF. The Modelica plant model is translated into IF through the automatic ANTLR-based translator. Basic models (classes) in Modelica are translated into corresponding processes in IF, with their equations translated to IF equations.



```
                         ⋮
process PID extends CTProcess{
        parameter real kp = 1, ki = 1, kd = 1, kc = 1;
        parameter real lb = -1, ub = 1;
        parameter real eps = 0;
        input port CTInterface real uset, u;
        output port CTInterface real out;
        equations{
                err = uset - u;
                sum = kp * err + ki * (intg(err) + kc * (out -
sum)) + kd * deri(err);
                out = (sum < lb + eps) * lb
                    + (sum > ub - eps) * ub
                    + (sum >= lb + eps && sum <= ub -eps) * sum;
        }
}
                         ⋮
netlist roomVAV extends CTNetlist {
        input port CTInterface real controlBus, TRoo, Tsup;
        output port CTInterface real yHea, yDam;
        SMOOTHMAX smoMax(deltaX = 0.1);
        SMOOTHMIN smoMin(deltaX = 0.1);
        limPID conHea(k = 0.01/10, Ti = 60/2, Td = 60, yMax = 1,
yMin = 0, wp = 1, wd = 0, Ni = 0.9, Nd = 10, xi_start = 0.1,
xd_start = 0, y_start = 0);
        limPID conCoo(k = 0.01/10, Ti = 60/2, Td = 60, yMax = 1,
yMin = 0, wp = 1, wd = 0, Ni = 0.9, Nd = 10, xi_start = 0,
xd_start = 0, y_start = 0);
        CONST one(k = 1);
        CONST zero(k = 0.1);
        ADD_3 yDamHea(k3 = -1);
        ADD add;
        CTChannel one_yDamHea, yDamHea_smoMax, zero_smoMax,
smoMax_add, conCoo_add, add_smoMin, one_smoMin;
        connect(conHea.uset, controlBus); connect(conCoo.uset,
controlBus);
        connect(conHea.u, TRoo); connect(conCoo.u, TRoo);
                         ⋮
```

Figure 2.2: IF translation for room temperature control system

In order to validate the accuracy of our IF translation, we further translate the denotational IF to an executable model in the MetropolisII framework, and compare the simulation in MetropolisII versus the simulation of original input model. The translation from denotational IF to MetropolisII model was explained in Section 2.1.

For the simulation of original heterogeneous input model, first the Modelica plant model is imported into Simulink through the Dymola-Simulink interface (Dymola [4] is a modeling and simulation environment for Modelica language). In this case, the entire plant is imported into Simulink as a DymolaBlock, which wraps an S-function MEX block that contains the C-code generated by Dymola for the Modelica model. The DymolaBlock may be configured by setting parameters and start values of the block. Once the DymolaBlock is imported, the heterogeneous input model can be simulated in Simulink.

The comparison of MetropolisII simulation versus the heterogeneous input model is shown in Figure 2.3 and Table 2.1. Figure 2.3 shows the temperature of Room 1 from the simulation of two models over an entire day. The other rooms have similar plots. Table 2.1 summarizes the average and maximum temperature differences for all three rooms. Overall, the simulation results of two models are close, which shows the accuracy of our IF translation.

We believe the remaining differences are mostly caused by the difference in implementing ODE solvers.

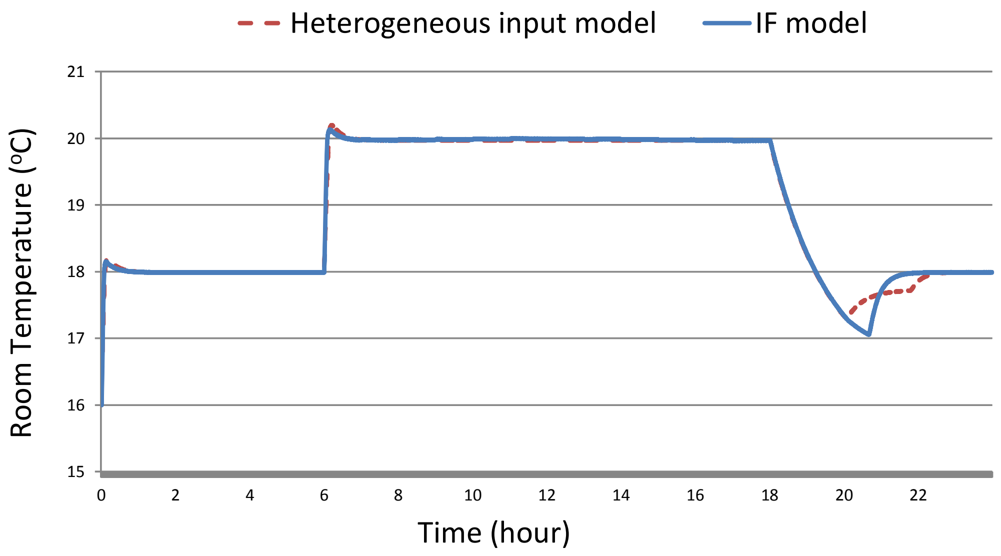


Figure 2.3: Comparison of heterogeneous input model in Simulink/Modelica and IF model in MetropolisII

Table 2.1: Comparison of all room temperatures

	Room1	Room2	Room3
Avg. differences (^oC)	0.033	0.034	0.125
Max. differences (^oC)	0.56	0.60	1.13

2.2.2 IF Translation at Different Abstraction Levels

As we mentioned earlier in Section 2.1, one important aspect of IF translation is that it can be conducted at different abstraction levels. In this part of the case study, we demonstrate this feature by translating the Simulink controller in the input model to an output language *through IF at two different levels of abstraction.* We choose the *G* language from National Instruments (NI) as the output as NI provides a comprehensive tool chain assocaited with G in the LabVIEW environment, including both simulation and code generation tools. The plant model is imported to LabVIEW and is the same for both abstraction levels.

We first translate the Simulink controller through IF to G in LabVIEW at the *PID level.* The translation from Simulink to IF is the same as shown earlier in Figure 2.2. The one LQR and three PID components in the Simulink model are translated one-to-one into processes in IF. The translation from IF to G in LabVIEW is similarly based on the correspondence at the component level. As mentioned before, the scheduling in Simulink for the dataflow type semantics is based on the causality relation between components. This is translated to ERM scheduling in IF, and further translated to the component scheduling in LabVIEW, which is also based on causality relations. The Runge-Kutta ODE solver used in Simulink is translated to ERM and EM scheduling in IF, and then translated to the Runge-Kutta solver available in LabVIEW.

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 is imported to LabVIEW to provide a fair comparison of the control system part. In Figure 2.4, 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.

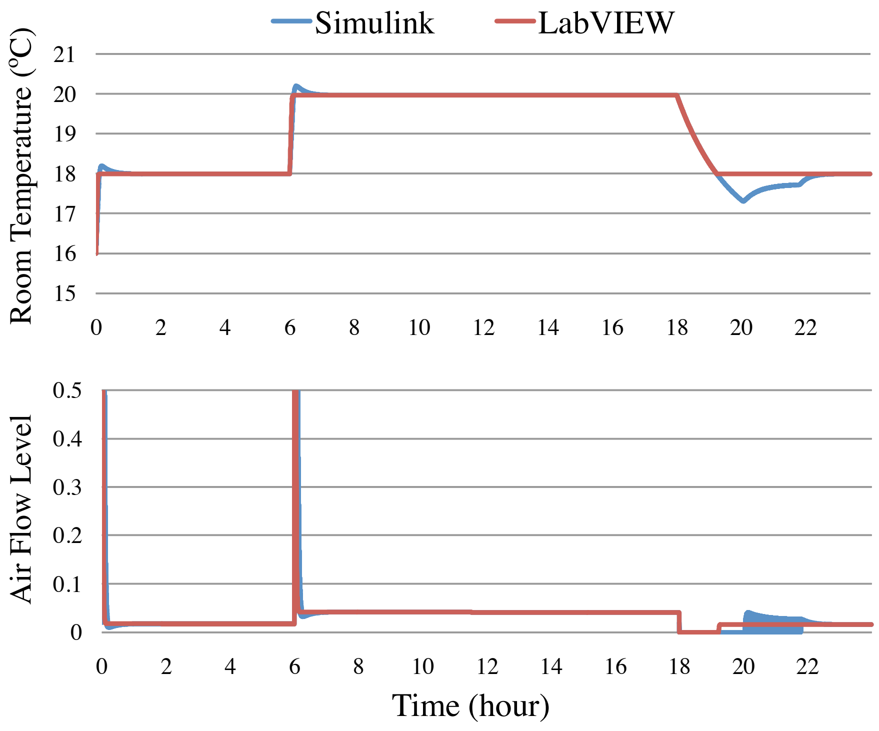


Figure 2.4: Comparison of Simulink and LabVIEW models at PID level

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 G in 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, and translate them from Simulink through IF to G in LabVIEW at this *sub-PID level*. We then assemble those lower level LabVIEW components to construct a PID in LabVIEW. This process is shown in Figure 2.5.

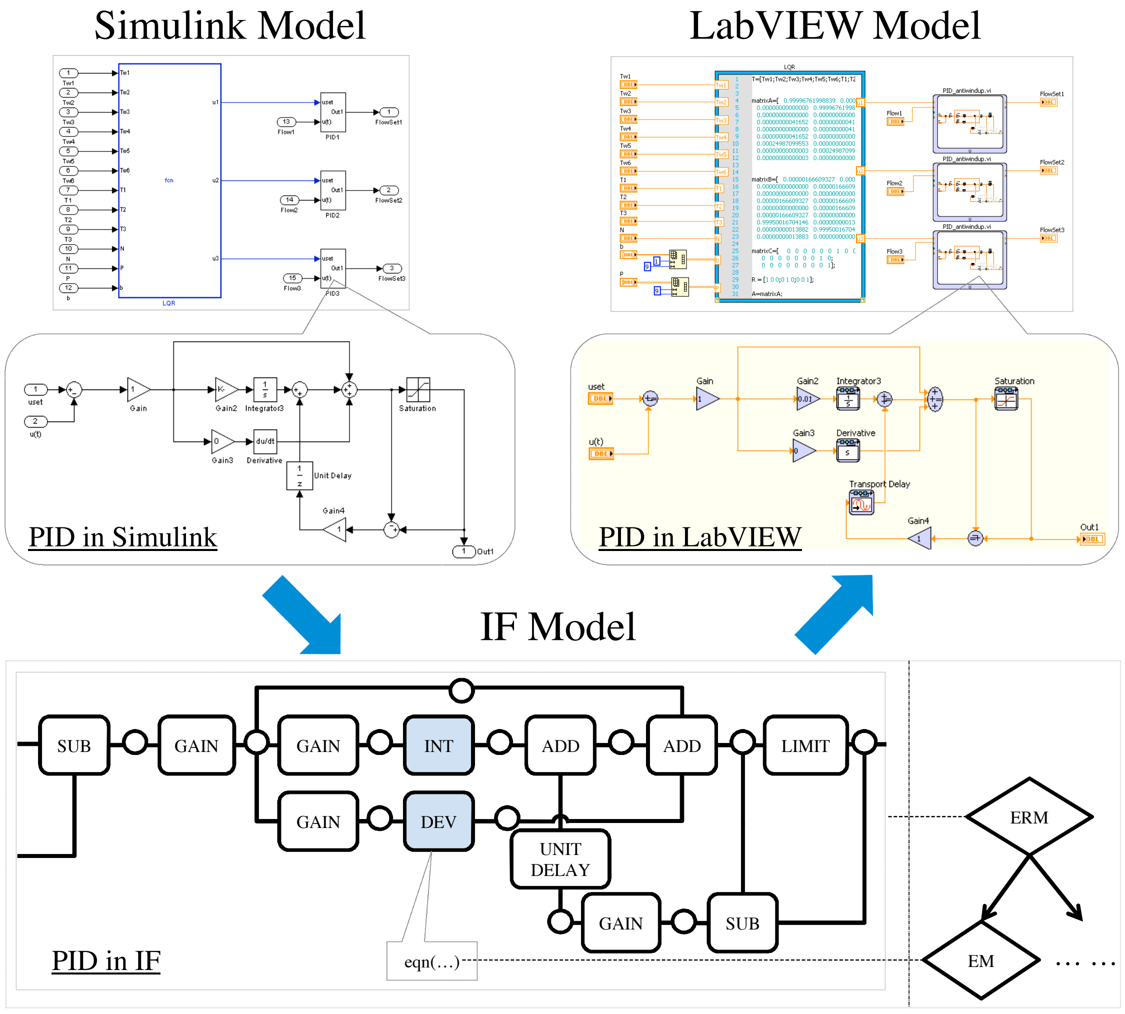


Figure 2.5: IF translation at sub-PID level

The comparison of the translations at different abstraction levels is shown below. Table 2.2 shows the absolute differences of the room temperatures between Simulink and LabVIEW simulations at two different abstraction levels. Table 2.3 shows the relative differences of the cumulative air flow levels. From these two tables, we can see that the differences are reduced by 10^1 to 10^3 times when using the lower level of abstraction. Of course this is at the expense of more modeling complexity and translation effort.

Table 2.2: Comparison of room temperature at different abstraction levels

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

Table 2.3: Comparison of cumulative air flow at different abstraction levels

	Level	Room1	Room2	Room3
Cumulative air flow level differences (%)	PID	1.29	1.29	1.55
	sub-PID	6.26×10^{-3}	6.26×10^{-3}	8.15×10^{-4}

Chapter 3

Mapping

In Step 2 of the software synthesis flow, mapping is conducted to explore the design space, including the selection of computation resources, the allocation of control functions onto processors, and the synthesis of communication network. Note that traditionally, the mapping step focuses on bridging a given functional model and a given architectural model. In this work, we extend its scope to include the exploration of the architecture platform (including selection of the computation and communication resources). We propose a general mapping flow as shown in Figure 3.1. The inputs include a functional model that is derived from the IF, an architecture platform that captures the computation and communication resources for realizing the functional specification, and a set of design constraints and objectives.

3.1 General Mapping Flow

The functional model represented in IF includes processes and channels. Through automatic extraction based on ANTLR, processes and channels are abstracted to tasks and signals, by hiding their internal implementation while 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 $\mathcal{F} = (\mathcal{T}, \mathcal{S})$. $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ is the set of tasks that perform the computations. $\mathcal{S} = \{s_1, s_2, ..., s_m\}$ is the set of signals that are exchanged between task pairs.

The architecture platform 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 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, extensibility, 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, and constraints from the physical environment and resources (for example, in BAC systems, the building floorplan and geometry impose constraints on the locations of sensors, actuators and processing units, and wire layout).

Conceptually, there are three steps in the general mapping flow. In the first step, a set of computation components $\mathcal{P}_U$ is selected from the architecture platform and connected by virtual communication components $\mathcal{L}_U$. This constitutes an architectural model $\mathcal{A}_U$ 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. Besides, the signals between tasks are packed into messages, and 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. 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 this 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.

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 the following part of this chapter, we present two mapping algorithms: one is to optimize the total cost of a BAC system while satisfying the real-time constraints and the constraints imposed by building floorplan and geometry; the other is to optimize the extensibility of a CAN-bus based system while satisfying the hard end-to-end latency constraints.

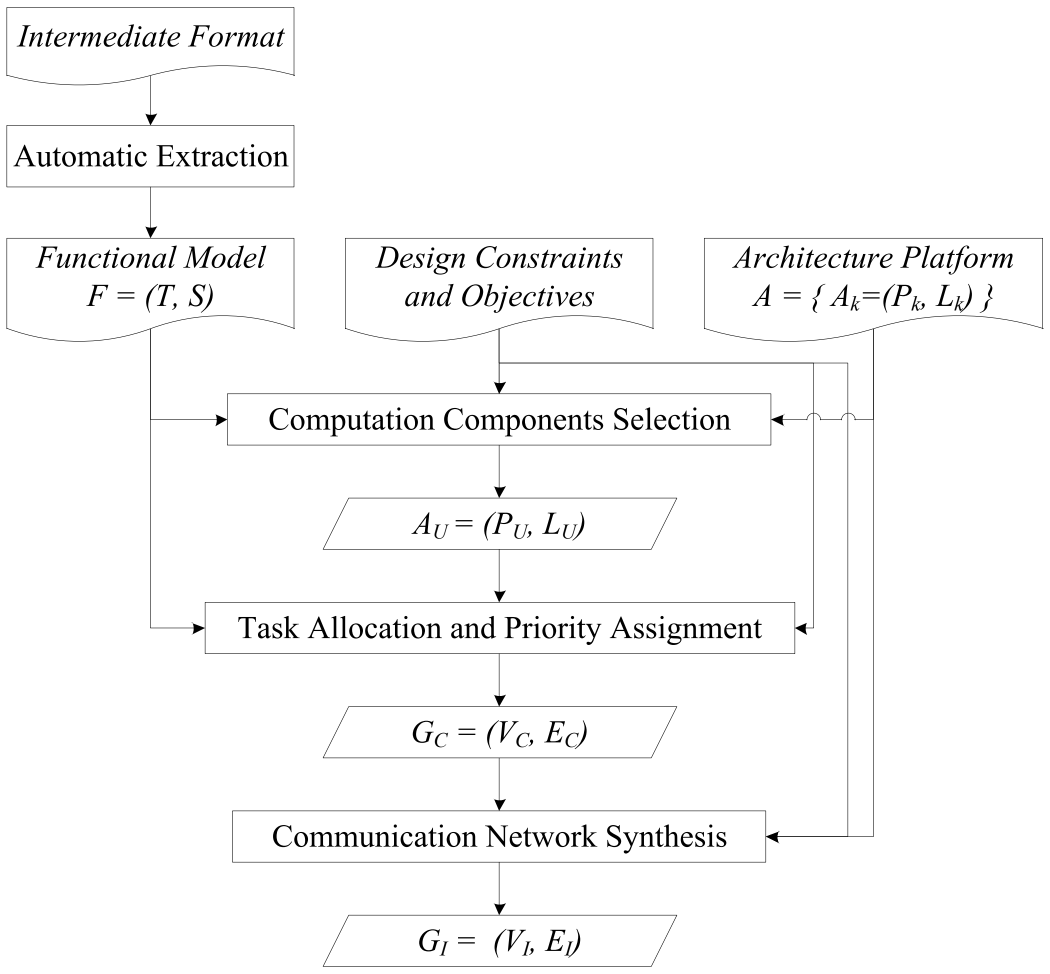


Figure 3.1: Mapping flow

3.2 Cost Optimization Mapping Algorithm for BAC Systems

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 $d$esign 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 $d$esign space that consists of the selection of computation components, allocation of tasks to embedded processors, assignment of task priorities, and communication network, to minimize the $s$ystem cost, which includes the prices of the components and the installation cost.

For this specific problem, we combine the first and second step of the general mapping flow in Figure 3.1 to explore the selection of computation components together with the allocation and priority assignment of the tasks. We then perform communication network synthesis as in the third step of the mapping flow. The details are explained in below.

3.2.1 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 2.1, 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 periodically activated with period T_{τ_i}. Strictly speaking, since the architecture we consider is loosely time-triggered, the periodical tasks are following the local clocks that may have drifts and jitters. To reduce the problem complexity, we assume a fixed period for each task in mapping, and leave the consideration of clock drifts and jitters to the code generation step, as shown later in Chapter 4.

Tasks are scheduled with preemption according to their priorities, and a total order exists among the task priorities on each node. We use o_{τ_i,τ_j} to denote the priority relation between task τ_i and τ_j, i.e. o_{τ_i,τ_j} is 1 if τ_j has a higher priority than τ_i, 0 otherwise. Computational nodes can be heterogeneous, and tasks can have different execution times on different nodes. We use C_{τ_i,p_j} to denote 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.

For a signal s_i, src_{s_i} and $\{dst_{s_i,j}\}$ denote the source task and the set of destination tasks of signal s_i, respectively (communication may be of multicast type). The computational nodes to which the source task src_{s_i} and the destination task $dst_{s_i,j}$ are allocated are called source and destination nodes, respectively. If the source node is the same as all the destination nodes, the signal is local. Otherwise, it is global and must be packed into a message transmitted on the network between the source node and all its destination nodes. In this mapping problem, we assume each signal s_i is packed into its own message, denoted by m_i (in Section 3.3, we will explore signal packing for CAN-bus based systems). $\mathcal{M} = \{m_1, m_2, ..., m_l\}$ denotes the set of messages communicated between tasks. Similarly as for signals, src_{m_i} and dst_{m_i} denote the source task and destination task of message m_i. 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 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.

3.2.1.1 End-to-End Latency

A path ρ on the application graph $\mathcal{G}$ is an ordered interleaving sequence of tasks and signals, defined as $\rho = [\tau_{r_1}, s_{r_1}, \tau_{r_2}, s_{r_2}, ..., s_{r_{k-1}}, \tau_{r_k}]$. $src(\rho) = \tau_{r_1}$ is the path's source task and $snk(\rho) = \tau_{r_k}$ is its sink task. Source tasks are activated by external events, while sink tasks activate actuators. A typical path for BAC systems would start from a sensing task, pass through tasks running control algorithms, and end at an actuating task. Multiple paths may exist between each source-sink pair. The worst case end-to-end latency incurred when traveling a path ρ is denoted as l_ρ. The path deadline for ρ, denoted by d_ρ, is an application requirement that may be imposed on selected paths.

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. Let l_{s_i} denote the worst case transmission time of a signal s_i. In this problem, s_i is mapped to its own message m_i, therefore $l_{s_i} = l_{m_i}$, and $m_i \in \rho$ if and only if $s_i \in \rho$. The worst case end-to-end latency of a path can be computed as follows.

$$l_\rho = \sum_{\tau_i \in \rho} r_{\tau_i} + \sum_{m_i \in \rho} l_{m_i} + \sum_{m_i \in \rho \wedge m_i \in GS} T_{dst_{m_i}} \tag{3.1}$$

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 [33, 78], 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.

3.2.1.2 Task Response Time Analysis

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 [33, 50].

$$r_{\tau_i} = C_{\tau_i} + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{r_{\tau_i}}{T_{\tau_j}} \right\rceil C_{\tau_j} \tag{3.2}$$

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

3.2.1.3 MILP Formulation of the Optimization Problem

A mixed-integer linear programming (MILP) formulation of the optimization problem is in below, with a summary of symbols in the following table.

a_{τ_i,p_j}	binary variable, whether task τ_i is allocated on component p_j
s_{p_j}	binary variable, whether p_j is selected
C_{τ_i,p_j}, M_{τ_i,p_j}	parameters, WCET and memory consumption of τ_i on p_j
U_{p_j}, R_{p_j}	parameters, utilization and memory uppder bound of p_j
h_{τ_i,τ_j}	binary variable, whether τ_i and τ_j are on the same component
g_{m_i}	binary variable, whether message m_i is a global message
l_{m_i}	real variable, latency of message m_i
$f_{\tau_i,p_k,\tau_j,p_q}$	auxiliary variable for message latency computation
x_{τ_i,τ_j,p_k}, y_{τ_i,τ_j,p_k}, z_{τ_i,τ_j,p_k}	auxiliary variables for task response time
o_{τ_i,τ_j}	binary variable, whether τ_i has higher priority than τ_j

Table 3.1: Summary of symbols in MILP

Allocation constraints:

$$\forall \tau_i \in \mathcal{T}, \quad \sum_{p_j \in \mathcal{P}_{\tau_i}} a_{\tau_i,p_j} = 1 \tag{3.3}$$

$$\forall \tau_i \in \mathcal{T}, p_j \notin \mathcal{P}_{\tau_i}, \quad a_{\tau_i,p_j} = 0 \tag{3.4}$$

Equation (3.3) and (3.4) enforce that each task should be mapped to one computation component.

Selection constraints:

$$\forall p_j \in \mathcal{P}', \quad \sum_{\tau_i \in \mathcal{T}'} a_{\tau_i,p_j} \geq s_{p_j} \tag{3.5}$$

$$\forall \tau_i \in \mathcal{T}', p_j \in \mathcal{P}', \quad a_{\tau_i,p_j} \leq s_{p_j} \tag{3.6}$$

Equation (3.5) and (3.6) define the selection of processors.

Resource constraints:

$$\forall p_j \in \mathcal{P}', \quad \sum_{\tau_i \in \mathcal{T}'} a_{\tau_i,p_j} * C_{\tau_i,p_j}/T_{\tau_i} \leq U_{p_j} \tag{3.7}$$

$$\forall p_j \in \mathcal{P}', \quad \sum_{\tau_i \in \mathcal{T}'} a_{\tau_i,p_j} * M_{\tau_i,p_j} \leq R_{p_j} \tag{3.8}$$

Equation (3.7) and (3.8) 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.

Global message:

$$\forall p_k \in \mathcal{P}, \quad a_{\tau_i,p_k} + a_{\tau_j,p_k} - 1 \leq h_{\tau_i,\tau_j} \tag{3.9}$$

$$\forall p_k, p_q \in \mathcal{P}, \quad 2 - a_{\tau_i,p_k} - a_{\tau_j,p_q} \geq h_{\tau_i,\tau_j} \tag{3.10}$$

$$\forall m_i \in \mathcal{M}, \quad 1 - h_{src_{m_i},dst_{m_i}} = g_{m_i} \tag{3.11}$$

Equations (3.9) to (3.11) define whether a message is a global message.

Message latency:

$$\sum_{p_k,p_q \in \mathcal{P}} f_{src_{m_i},p_k,dst_{m_i},p_q} * L_{p_k,p_q} = l_{m_i} \tag{3.12}$$

$$a_{\tau_i,p_k} + a_{\tau_j,p_q} - 1 \leq f_{\tau_i,p_k,\tau_j,p_q} \tag{3.13}$$

$$a_{\tau_i,p_k} \geq f_{\tau_i,p_k,\tau_j,p_q} \tag{3.14}$$

$$a_{\tau_j,p_q} \geq f_{\tau_i,p_k,\tau_j,p_q} \tag{3.15}$$

Equations (3.12) to (3.15) compute the message latency.

Task response time:

$$\sum_{\tau_j \in \mathcal{T}} \sum_{p_k \in \mathcal{P}} z_{\tau_i,\tau_j,p_k} * C_{\tau_j,p_k} + \sum_{p_j \in \mathcal{P}} a_{\tau_i,p_j} * C_{\tau_i,p_j} = r_{\tau_i} \tag{3.16}$$

$$y_{\tau_i,\tau_j,p_k} - M * (1 - o_{\tau_i,\tau_j}) \leq z_{\tau_i,\tau_j,p_k} \leq y_{\tau_i,\tau_j,p_k} \tag{3.17}$$

$$z_{\tau_i,\tau_j,p_k} \leq M * o_{\tau_i,\tau_j} \tag{3.18}$$

$$x_{\tau_i,\tau_j,p_k} - M * (1 - a_{\tau_i,p_k}) \leq y_{\tau_i,\tau_j,p_k} \leq x_{\tau_i,\tau_j,p_k} \tag{3.19}$$

$$y_{\tau_i,\tau_j,p_k} \leq M * a_{\tau_i,p_k} \tag{3.20}$$

$$u_{\tau_i,\tau_j} - M * (1 - a_{\tau_j,p_k}) \leq x_{\tau_i,\tau_j,p_k} \leq u_{\tau_i,\tau_j} \tag{3.21}$$

$$x_{\tau_i,\tau_j,p_k} \leq M * a_{\tau_j,p_k} \tag{3.22}$$

$$0 \leq u_{\tau_i,\tau_j} - r_{\tau_i}/T_{\tau_j} < 1 \tag{3.23}$$

$$r_{\tau_i} \leq T_{\tau_i} \tag{3.24}$$

Equations (3.16) to (3.24) 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 (3.17) or (3.18) 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 since there is only one task on the computation node (sensor or actuator). Similarly, the computation of g_{m_i} and l_{m_i} are simple if the source or destination task of m_i is either a sensing or actuating task.

End-to-end latency:

$$\forall \rho_k, \quad l_{\rho_k} \leq d_{\rho_k} \tag{3.25}$$

$$\sum_{\tau_i \in \rho_k} r_{\tau_i} + \sum_{m_i \in \rho_k} (l_{m_i} + g_{m_i} * T_{dst_{m_i}}) = l_{\rho_k} \tag{3.26}$$

Equations (3.25) and (3.26) set up the end-to-end latency constraints on paths.

Priorty constraints:

$$\forall \tau_i, \tau_j \in \mathcal{T}', \quad o_{\tau_i,\tau_j} + o_{\tau_j,\tau_i} = 1 \tag{3.27}$$

$$\forall \tau_i, \tau_j, \tau_k \in \mathcal{T}', \quad o_{\tau_i,\tau_j} + o_{\tau_j,\tau_k} - 1 \leq o_{\tau_i,\tau_k} \tag{3.28}$$

Equations (3.27) and (3.28) assure the correct assignment of priorities. We only explore the priorities for tasks mapped onto processors since sensing and actuating tasks are mapped one-to-one to sensors or actuators.

Objective function:

$$\min \sum_{p_j \in \mathcal{P}'} s_{p_j} * V_{p_j} \tag{3.29}$$

Finally, Equation (3.29) 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.

3.2.2 Communication Network Synthesis

As shown in Figure 3.1, 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) [57][1] 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.

3.2.3 Case Study

We applied our mapping formulation and algorithm to the room temperature control example shown in Figure 2.1. 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 3.2, part of which is the same as in [53]. We use ARCNET [2] daisy chain buses as communication library.

The MILP problem is solved using CPLEX 11.0 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 3.2. The LQR task is mapped to the only

[1]For more details of COSI formulation and algorithms, please refer to [57, 53, 55].

Table 3.2: 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

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 3.3 shows the final result after communication network synthesis.

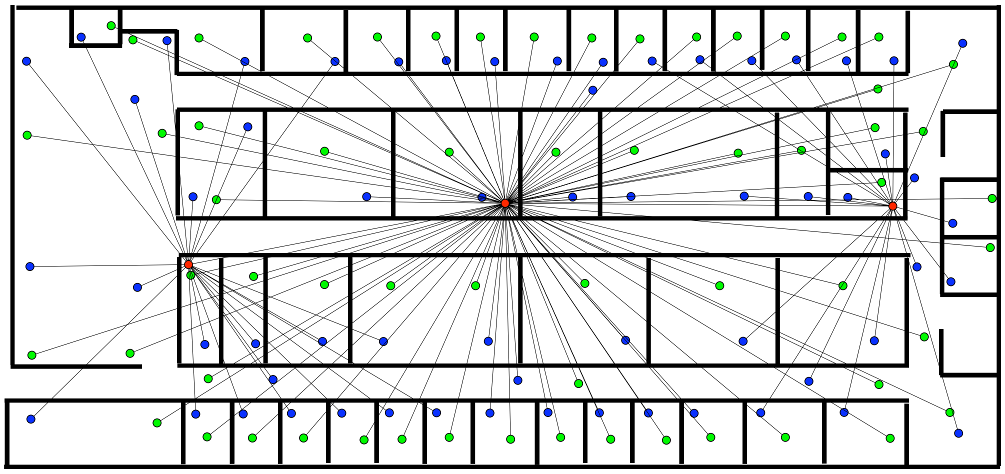


Figure 3.2: Mapping result after MILP

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 will increase to \$20196.22 (the final layout

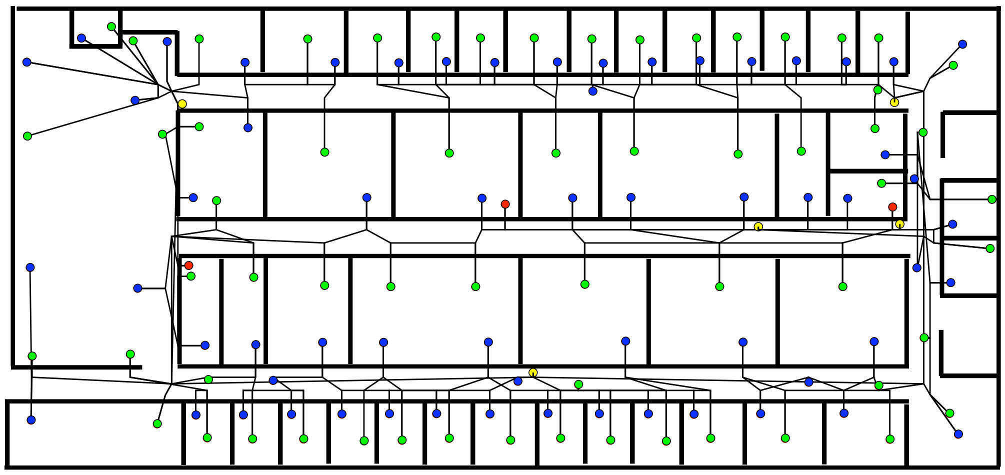


Figure 3.3: Final mapping result

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.

3.3 Extensibility Optimization Mapping Algorithm for CAN-bus based Systems

We proposed another mapping algorithm that optimizes extensibility of hard real-time distributed systems. Extensibility is defined as the amount by which the execution time of tasks can be increased without changing the system configuration while meeting the design constraints. Optimizing extensibility is particularly important for systems with large production quantities and long lifetime, such as automotive systems, avionics systems and BAC systems. These systems must accommodate function updates or additions for new features and error fixes over a multi-year product lifetime. Any major change in the software or hardware architecture that requires the replacement of one or more subsystems means huge losses because of the large quantities involved and the backlogs in the production of these units. Being able to upgrade or adjust the software design incrementally, without undergoing a major re-design cycle is imperative for competitive advantage. Extensibility optimization addresses this problem by finding the design that is as robust as possible with respect to modifying the existing tasks. With this definition, a design that is optimized for extensibility not only allows adding future functionality with minimum changes, but is also more robust with respect to errors in the estimation of task execution times.

The hard real-time systems we consider collect data from a set of sensors, perform computations in a distributed fashion and based on the results, send commands to a set of actuators. We focus on systems that are based on *priority-based scheduling* of periodic tasks and messages. Each input data (generated by a sensor, for instance) is available at one of the system's computational nodes. A periodically activated task on this node reads the input data, computes intermediate results, and writes them to the output buffer from where they can be read by another task or used for assembling the data content of a message. Messages - also periodically activated - transmit the data from the output buffer on the current node over the bus to an input buffer on a remote node. Local clocks on different nodes are not synchronized. Tasks may have multiple fan-ins and messages can be multi-cast. Eventually, task outputs are sent to the system's output devices or actuators.

The architecture platform is given in this case, which consists of a set of ECUs connected by CAN buses. The design space includes task allocation and priority assignment, signal packing and message priority assignment. Following the idea of the general mapping flow discussed in Section 3.1, we designed a two-stage algorithm for the mapping problem. The first stage of the algorithm is based on mixed integer linear programming (MILP), where task allocation (the most important variable with respect to extensibility) is optimized within deadline and utilization constraints. The second stage features three heuristic steps, which iteratively pack signals to messages, assign priorities to tasks and messages, and explore task re-allocation. This algorithm runs much faster than randomized optimization approaches (a 20x reduction with respect to simulated annealing as shown in [77]). Hence, it is applicable to industrial-size systems as shown by the experimental case studies in Section 3.3.3, addressing the typical case of the deployment of additional functionalities in a commercial car. The shorter running time of the proposed algorithm allows using the method not only for the optimization of a given system configuration, but also for *architecture exploration*, where the number of system configurations to be evaluated and subject to optimization can be large.

A further advantage of an MILP formulation (even if used only for the first stage) with respect to randomized optimization, is the possibility of leveraging mature technology in solvers, the capability of detecting the actual optimum (when found in reasonable time), or, when the running time is excessive, to compute at any time a lower bound on the cost of the optimum solution, which allows to evaluate the quality of the best solution obtained up to that point.

3.3.1 Representation

The problem representation has some similarities with the mapping problem introduced in Section 3.2, but also has its uniqueness due to the consideration of gateways, signal packing and most importantly the extensibility metric.

The application is represented as a directed graph $\mathcal{G} = (\mathcal{T}, \mathcal{S})$ as discussed in Section 3.1. The application is mapped onto an architecture that consists of a set of computational nodes, denoted as $\mathcal{E} = \{e_1, e_2, ..., e_p\}$, connected through a set of CAN buses $\mathcal{B} = \{b_1, b_2, ..., b_q\}$.

τ_i is periodically activated with period t_{τ_i}, and executed with priority p_{τ_i}. The periods of communicating tasks are assumed to be harmonic, which is almost always true in practical designs. Tasks are scheduled with preemption according to their priorities, and a total order exists among the task priorities on the same node. Tasks can have different execution times on different nodes. We use $c_{\tau_i,e}$ to denote the execution time of task τ_i on node e. In the following, the e subscript is dropped whenever the formula refers to tasks on a given node, and e is implicitly defined, or when the task allocation is (at least temporarily) defined, and the node to which the computation time (or its extensibility Δc) refers, is known. Finally, r_{τ_i} denotes the worst case response time.

A global signal s_i, whose source node is different than the destination node, must be packed into a message transmitted on the buses between the source node and all its destination nodes. Only signals with the same period, same source node and same communication bus can be packed into the same message. For message m_i, t_{m_i} denotes its period, p_{m_i} denotes its priority, and c_{m_i} denotes its worst case transmission time on a bus with unit speed. The worst transmission time on bus b_j is $c_{m_i}/speed_{b_j}$, where $speed_{b_j}$ is the transmission speed of b_j. r_{m_i} is the worst case response time on a bus with unit speed.

In addition, in complex systems the source and destination tasks may not reside on computation nodes that share the same bus. In this case, a signal exchanged among them will have to go through a gateway node and be forwarded by a gateway task. We include the gateway concept in our model with a number of restrictive (but hopefully realistic) assumptions.

- any communication between two computation nodes is never going to need more than one gateway hop (every bus is connected to all the others through one gateway computation node). This assumption, realistic for small systems, could probably be removed at the price of additional complexity.
- a single gateway node connects any two buses.
- a single task is responsible for signal forwarding on each gateway node. This task is fully determined (there might be other tasks running on the gateway node.)

A path p on the application graph $\mathcal{G}$ is an ordered interleaving sequence of tasks and signals, defined as $p = [\tau_{r_1}, s_{r_1}, \tau_{r_2}, s_{r_2}, ..., s_{r_{k-1}}, \tau_{r_k}]$, where $src(p) = \tau_{r_1}$ is the path's source and $snk(p) = \tau_{r_k}$ is its sink. Multiple paths may exist between each source-sink pair. The worst case end-to-end latency of path p is denoted as l_p, and the path deadline of p is denoted by d_p.

It may be argued that today, it is industrial practice (at least in the automotive domain) to allocate resources between suppliers before implementation parameters (such as worst-case execution times) are known. This is however only partly true. A major architecture and functional redesign is often characterized by a significant reuse (or carry-over) of pre-existing functionality (60% to 70% are typical figures), for which these estimates could be available. In addition, rapid prototyping techniques and the increased use of automatic code generation tools should ease the availability of these implementation related parameters (or at least estimates) even for newly-designed functions.

3.3.1.1 Design Space and Extensibility Metric

The design problem can be defined as follows. Given a set of design constraints including:

- end-to-end deadlines on selected paths
- utilization bounds on nodes and buses
- maximum message sizes

explore the design space that includes:

- allocation of tasks to computational nodes
- packing of signals and allocation of messages to buses
- assignment of priorities to tasks and messages

to maximize *task extensibility.*

Different definitions can be provided for task extensibility. The main definition used in this work is as the weighted sum of each task's execution time slack over its period:

$$max. \quad S = \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} \frac{\Delta c_{\tau_i}}{t_{\tau_i}} \tag{3.30}$$

where a task's execution time slack Δc_{τ_i} is defined as the maximum possible increase of its execution time c_{τ_i} without violating the design constraints, assuming the execution times of other tasks are not changed.

w_{τ_i} is a preassigned weight that indicates how likely and how much the task's execution time will be increased in future functionality extensions. In practice, however, because of functional dependencies, execution time increases in tasks belonging to a set might need to be considered jointly. This can be done in several ways. One possible way is in the assignment of the w_{τ_i} weights as follows.

1. Identify a set of update scenarios $u_1, u_2, ...u_n$. Each scenario u_k includes a group of tasks $\mathcal{T}_k$ to be extended, and is assigned a likelihood probability p_k.

2. For each update scenario u_k and $\tau_i \in \mathcal{T}_k$, assign a weight w_{i_k} to represent how much the task's execution time will be increased in this scenario.

3. The final weight w_{τ_i} of a task is computed as $w_{\tau_i} = \sum_{k:\tau_i \in \mathcal{T}_k} p_k * w_{i_k}$.

A more explicit way is to identify groups of tasks that are functionally related so that their execution times increases are related (in a way expressed by a simple linear formulation).

We identify a set of task groups $\mathcal{T}_\mathcal{G} = \{\mathcal{T}_{g_1}, \mathcal{T}_{g_2}, ...\mathcal{T}_{g_n}\}$ (each $g_i \in \mathcal{G}$ representing an update scenario). Execution times of tasks belonging to the same group are bound to increase together in each update scenario. For each task $\tau_j \in \mathcal{T}_{g_i}$, we model the possible additional execution time as

$$\Delta c_{\tau_j,g_i} = A_{\tau_j,g_i} * \Delta c_{g_i} \tag{3.31}$$

where A_{τ_j,g_i} is a constant. Equation (3.31) represents a simple extensibility dependency among tasks belonging to a functional group (more complex relationships can be represented at the price of higher complexity.)

Based on Equation (3.31), we define this alternative extensibility metric as follows.

$$max. \quad S = \sum_{g_i \in \mathcal{G}} w_{g_i} * \Delta c_{g_i} \tag{3.32}$$

Finally, another formulation is to use execution time slack over original execution time, i.e. $\Delta c_{\tau_i}/c_{\tau_i}$, instead of using execution time slack over period in Equation (3.30).

The metric function (3.30) is used in the following discussion of the optimization algorithm. The required changes for the adoption of the metric in (3.32) are discussed further in Section 3.3.2.6. The comparison of the metric function (3.30) with the metric that uses slack times relative to the original execution time can be found in [77].

3.3.1.2 End-to-End Latency

After tasks are allocated, some signals are local, and their transmission time is assumed to be zero. Others are global, and need to be transmitted on the buses through messages. The time needed to transmit a global signal is equal to the transmission time of the corresponding message. Let r_{s_i} denote the worst case response time of a global signal s_i, and assume its corresponding message is m_j, then $r_{s_i} = r_{m_j}$.

The worst case end-to-end latency can be computed for each path by adding the worst case response times of all the tasks and global signals on the path, as well as the periods of all the global signals and their destination tasks on the path.

$$l_p = \sum_{\tau_i \in p} r_{\tau_i} + \sum_{s_i \in p \wedge s_i \in GS} (r_{s_i} + t_{s_i} + t_{dst_{s_i}}) \tag{3.33}$$

where GS is the set of all global signals. Of course, in the case that gateways are used across buses, the signals to and from possible gateway tasks, as well as the response time of the gateway task itself and the associated sampling delays must be included in the analysis.

We need to include periods of global signals and their destination tasks because of the asynchronous sampling of communication data. In the worst case, the input global signal arrives immediately after the completion of the first instance of task τ_i. The event data will be read by the task on its next instance and the result will be produced after its worst case response time, that is, $t_{\tau_i} + r_{\tau_i}$ time units after the arrival of the input signal. The same reasoning applies to the execution of all tasks that are the destinations of global signals, and applies to global signals themselves. However, for local signals, the destination task can be activated with a phase (offset) equal to the worst-case response time of the source task, under our assumption that their periods are harmonic. In this case, we only need to add the response time of the destination task. When a task has more than one local predecessor on a time-critical path, its activation phase (offset) will have to be set to the largest among the completion times of its predecessors. This case could be dealt with in the previous Equation (3.33) and in the following formulations, by replacing the response time contribution of local sender tasks with the maximum among the response times of all the senders for a given task in the path. Similarly, it is sometimes possible to synchronize the queuing of a message for transmission with the execution of the source tasks of the signals present in that message. This would reduce the worst case sampling period for the message transmission and decrease the latency in Equation (3.33). In this work, we do not consider these possible optimizations and leave them to future extensions.

Task Response Times

The analysis and calculation of task response times is the same as in the previous mapping problem for BAC systems (Equation (3.2)).

$$r_{\tau_i} = C_{\tau_i} + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{r_{\tau_i}}{T_{\tau_j}} \right\rceil C_{\tau_j} \tag{3.34}$$

Message Response Times

Worst case message response times are calculated similarly to task response times. The main difference is that message transmissions on the CAN bus are not preemptable. Therefore, a message m_i may have to wait for a blocking time B_{max}, which is the longest transmission time of any frame in the system. Likewise, the message itself is not subject to preemption from higher priority messages during its own transmission time c_{m_i}. The response time can therefore be calculated with the following recurrence relation, in the case of $r_{m_i} \leq t_{m_i}$:

$$r_{m_i} = c_{m_i} + B_{max} + \sum_{m_j \in hp(m_i)} \left\lceil \frac{r_{m_i} - c_{m_i}}{t_{m_j}} \right\rceil c_{m_j} \tag{3.35}$$

3.3.1.3 Formulation

Based on the formulas for computing end-to-end latencies and response times, we construct a mathematical formulation that contains all the design variables. Part of the formulation is similar to the one in [76]: both explore the same set of design variables - task allocation, signal packing and message allocation, as well as task and message priorities. In [76], the problem was formulated as mixed integer linear programming (MILP). To reduce the complexity, the problem was divided into sub-problems and solved by a two-step approach.

However, in [76], the objective is to minimize end-to-end latencies, while in this work, we optimize task extensibility. The formulation of task extensibility with respect to end-to-end deadline constraints is a quite challenging task. In general, inverting the function that computes response times as a function of the task execution times is of exponential complexity in the simple case of single-CPU scheduling [22]. When dealing with end-to-end constraints, the problem is definitely more complex. A possible approach consists of a very simple (but possibly time-expensive) bisection algorithm that finds the sensitivity of end-to-end response times with respect to increases in task execution times (this is the solution used for performing sensitivity analysis in [62]).

Formally, if Δr_{ij} denotes the increase of task τ_j's response time r_{τ_j} when task τ_i's computation time c_{τ_i} is increased by Δc_{τ_i}, the end-to-end latency constraints and utilization constraints are expressed as follows:

$$\sum_{\tau_j \in p \wedge \tau_j \in (lp(\tau_i) \cup \{\tau_i\})} \Delta r_{ij} \leq d_p - l_p \qquad \forall p, \forall \tau_i \in \mathcal{T} \tag{3.36}$$

$$\frac{\Delta c_{\tau_i}}{t_{\tau_i}} + \sum_{\tau_j \in \mathcal{T}(e)} \left(\frac{c_{\tau_j}}{t_{\tau_j}}\right) \leq u_e \qquad \forall e, \forall \tau_i \in \mathcal{T}(e) \tag{3.37}$$

where $lp(\tau_i)$ refers to the set of tasks with priority lower than p_{τ_i} and executed on the same node as τ_i, $\mathcal{T}(e)$ denotes the set of the tasks on computational node e, and u_e denotes the maximum utilization allowed on e.

The relation between Δr_{ij} and Δc_{τ_i} can be derived from Equation (3.2), as follows.

$$\begin{aligned}\Delta r_{ij} = &\sum_{\tau_k \in hp(\tau_j)} \left(\left\lceil \frac{r_{\tau_j} + \Delta r_{ij}}{t_{\tau_k}} \right\rceil - \left\lceil \frac{r_{\tau_j}}{t_{\tau_k}} \right\rceil\right) c_{\tau_k} \\ &+ \left\lceil \frac{r_{\tau_j} + \Delta r_{ij}}{t_{\tau_i}} \right\rceil \Delta c_{\tau_i} \qquad \forall \tau_j \in lp(\tau_i)\end{aligned} \tag{3.38}$$

$$\Delta r_{ii} = \Delta c_{\tau_i} + \sum_{\tau_k \in hp(\tau_i)} \left(\left\lceil \frac{r_{\tau_i} + \Delta r_{ii}}{t_{\tau_k}} \right\rceil - \left\lceil \frac{r_{\tau_i}}{t_{\tau_k}} \right\rceil\right) c_{\tau_k} \tag{3.39}$$

For brevity, above formulas do not model task allocation and priority assignment as variables. In the complete formulation, they were expanded to include those variables.

Contrary to the problem in [76], in our case the formulation cannot be linearized because of the second term in Equation (3.38). It could be solved by nonlinear solvers but the complexity is in general too high for industrial size applications. Therefore, we propose an algorithm that defines two stages to decompose the complexity: one in which mathematical programming (MILP) is used, and one refinement stage that consists of several steps based on heuristics.

3.3.2 Optimization Algorithm

The flow of our algorithm is shown in Figure 3.4. First, we decide the allocation of tasks, since the choices of other design variables are restricted by task allocation. In the initial allocation stage, the problem is formulated as MILP and solved by an MILP solver. Then

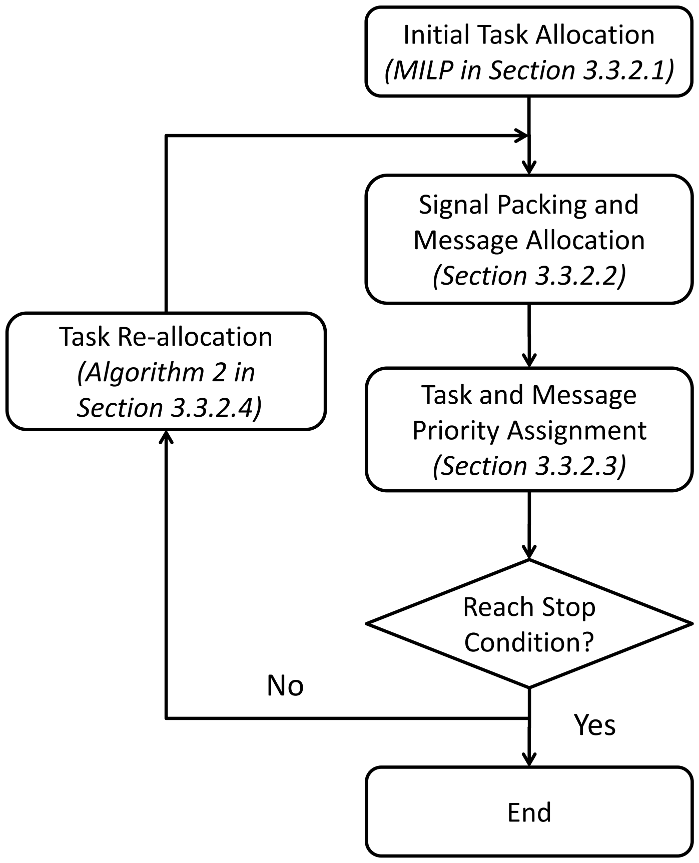

Initial Task Allocation
(MILP in Section 3.3.2.1)
Signal Packing and Message Allocation
(Section 3.3.2.2)
Task and Message Priority Assignment
(Section 3.3.2.3)
Reach Stop Condition?
Task Re-allocation
(Algorithm 2 in Section 3.3.2.4)
No
Yes
End

Figure 3.4: Algorithm flow for task extensibility optimization

a series of heuristics is used in the refinement stage: in the signal packing and message allocation step, a heuristic is used to decide signal-to-message packing and message-to-bus allocation. In the task and message priority assignment step, an iterative method is designed to assign the priorities of tasks and messages. After these steps are completed, if the design constraints cannot be satisfied or if we want to further improve extensibility, the tasks can be re-allocated and the process repeated. Because of the complexity of the MILP formulation, we designed a heuristic for task re-allocation, based on the extensibility and end-to-end latency values obtained in the previous steps.

3.3.2.1 Initial Task Allocation

In the initial task allocation stage, tasks are mapped onto nodes while meeting the utilization and end-to-end latency constraints. Utilization constraints are considered in place of the true extensibility metric to allow a linear formulation. In this stage, we also allocate signals to messages and buses assuming each message contains one signal only. The initial tasks and message priority assignment is assumed as given. In case the procedure is used to optimize an existing configuration, priorities are already defined. In case of new designs, any suitable policy, such as Rate Monotonic, can be used.

The MILP problem formulation includes the following variables and constraints:

Allocation constraints

$$\sum_{e \in E(\tau_i)} a_{\tau_i,e} = 1 \tag{3.40}$$

$$a_{\tau_i,e} + a_{\tau_j,e} - 1 \leq h_{\tau_i,\tau_j,e} \tag{3.41}$$

$$h_{\tau_i,\tau_j,e} \leq a_{\tau_i,e} \tag{3.42}$$

$$h_{\tau_i,\tau_j,e} \leq a_{\tau_j,e} \tag{3.43}$$

$$\forall b_r \in B(e_m), e_m \neq e_p \quad a_{\tau_{s_i},e_m} + a_{\tau_j,e_p} - 1 \leq a_{s_{i,j,0},b_r} \tag{3.44}$$

$$\forall B(e_m) \cap B(e_p) = \emptyset \quad a_{\tau_{s_i},e_m} + a_{\tau_j,e_p} - 1 \leq a_{s_{i,j,k},b_r} \tag{3.45}$$

$$\forall b_r \quad g_{s_{i,j,0}} \geq a_{s_{i,j,0},b_r} \tag{3.46}$$

$$g_{s_{i,j,0}} \leq \sum_{b_r} a_{s_{i,j,0},b_r} \tag{3.47}$$

$$\forall b_r \quad g_{s_{i,j,k}} \geq a_{s_{i,j,k},b_r} \tag{3.48}$$

$$g_{s_{i,j,k}} \leq \sum_{b_r} a_{s_{i,j,k},b_r} \tag{3.49}$$

$$\forall s_{i,j,k} \quad g_{s_{i,j,k}} \leq g_{s_{i,j,0}} \tag{3.50}$$

$$\forall \tau_j, k \quad a_{s_{i,j,k},b_r} \leq a_{s_i,b_r} \tag{3.51}$$

$$\forall \tau_j \quad a_{s_{i,j,0},b_r} \leq a_{s_i,b_r} \tag{3.52}$$

$$a_{s_i,b_r} \leq \sum_{j,k \cup 0} a_{s_{i,j,k},b_r} \tag{3.53}$$

$$a_{s_{i,j,0},b_r} + a_{s_m,b_r} - 1 \leq h_{s_{i,j,0},s_m,b_r} \tag{3.54}$$

$$h_{s_{i,j,0},s_m,b_r} \leq a_{s_{i,j,0},b_r} \tag{3.55}$$

$$h_{s_{i,j,0},s_m,b_r} \leq a_{s_m,b_r} \tag{3.56}$$

$$a_{s_{i,j,k},b_r} + a_{s_m,b_r} - 1 \leq h_{s_{i,j,k},s_m,b_r} \tag{3.57}$$

$$h_{s_{i,j,k},s_m,b_r} \leq a_{s_{i,j,k},b_r} \tag{3.58}$$

$$h_{s_{i,j,k},s_m,b_r} \leq a_{s_m,b_r} \tag{3.59}$$

Gatewaying of signals requires additional definitions and a modification of the signal set to accommodate replicated signals that are sent by the gateway tasks. Gateway tasks are preallocated, with known period and priority subject to optimization.

For each signal s_i in the task communication model we use $s_{i,j,0}$ to represent the signal originating from the source task and directed to the destination τ_j or (if needed) to the

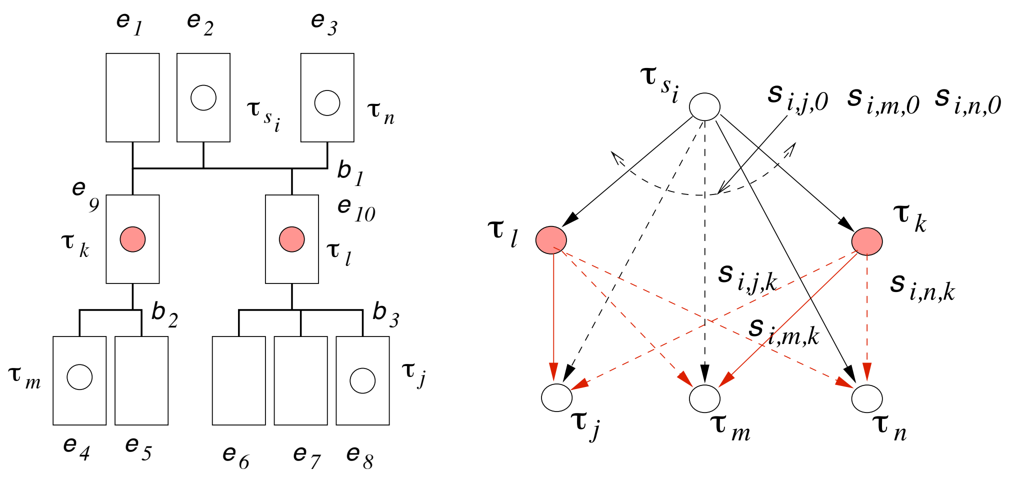


Figure 3.5: Signal forwarding by the gateways

gateway task with final destination τ_j. In addition, for each possible gateway τ_k, there is an additional possible signal, labeled $s_{i,j,k}$ representing the signal from the gateway task τ_k to the destination τ_j (allocated on a computation node that can be reached with gateway τ_k, Figure 3.5).

In case the source task and the destination task τ_j are on the same node, the signal $s_{i,j,0}$ and all the $s_{i,j,k}$ may be disregarded since they do not contribute to the latency and they will not need a message to be transmitted. In case the source and destination task are connected by a single bus, $s_{i,j,0}$ represents the signal between them and all the $s_{i,j,k}$ should be disregarded (accomplished by treating them as local signals).

For each s_i there is one set $s_{i,j,0}$ with as many signals as the number of receivers and one set $s_{i,j,k}$ with cardinality equal to the product of the number of possible gateways by the number of receivers. All gateway signals have the same period and data length of the signals from which they originate.

$E(\tau_i)$ is the set of nodes that τ_i can be allocated to. $B(e)$ represents the set of buses to which e is connected.

The Boolean variable $a_{\tau_i,e}$ indicates whether task τ_i is mapped onto node e, and $h_{\tau_i,\tau_j,e}$ defines whether τ_i and τ_j are on the same node e. Constraint (3.40) ensures that each task is mapped to one node and only one, and the set of constraints (3.41, 3.42 , 3.43) ensures the consistency of the definitions of the h and a variables.

The Boolean variable $a_{s_{i,j,k},b_r}$ is 1 if signal $s_{i,j,k}$ is mapped onto bus b_r and 0 otherwise, similarly for the definition of $a_{s_{i,j,0},b_r}$. To define these set of variables, we need to consider all computation node pairs for each signal from its source to all its destinations τ_j. In the following, for simplicity, we will label as τ_{s_i} the source task for signal s_i. The set of constraints defined by (3.44) for all possible sets (source τ_{s_i}, destination τ_j, source node e_m that is on bus b_r, destination node e_p that communicates with e_m through b_r) forces $a_{s_{i,j,0},b_r}$ to 1 for the bus b_r from e_m to e_p, or from e_m to the destination of the gateway task τ_k between e_m and

e_p. The following set (3.45) sets $a_{s_{i,j,k},b_r}$ to 1 (if necessary) for the bus b_r from the gateway to the destination node e_p when gatewaying is needed (in this set of constraints, e_p is on b_r while e_m is not). The variables $a_{s_{i,j,k},b_r}$ have a positive contribution to the cost function, hence they will be set to 0 by the optimization engine, unless forced at 1 by the constraints.

To give an example of these constraints, in Figure 3.5 the condition for the outgoing signal $s_{i,j,0}$ from τ_{s_i} to τ_j to be on bus b_1, expressed as

$$a_{\tau_{s_i},e_m} + a_{\tau_j,e_p} - 1 \leq a_{s_{i,j,0},b_1}$$

needs to be defined for each computation node pair (m,p) where $m \in \{1,2,3\}$ and $p \neq m$, or $m \in \{9\}$ and $p \in \{1,2,3,10\}$, or $m \in \{10\}$ and $p \in \{1,2,3,9\}$. Similar sets of conditions will then need to be defined for b_2 and b_3.

As an example of gatewaying, the condition for the mapping on b_2 of the (possible) signal forwarded by gateway τ_k as part of the communication from τ_{s_i} to τ_j in the figure is expressed by the set.

$$a_{\tau_{s_i},e_m} + a_{\tau_j,e_p} - 1 \leq a_{s_{i,j,m},b_2}$$

defined for all the computation node pairs (m,p) where $m \in \{1,2,3,10\}$ and $p \in \{4,5\}$.

The value of the Boolean variable $g_{s_{i,j,0}}$ is 1 if $s_{i,j,0}$ is a global signal (i.e., being transferred on bus), and 0 otherwise. Similarly, $g_{s_{i,j,k}}$ is 1 if the signal $s_{i,j,k}$ (s_i forwarded by gateway τ_k) is global. The definition of $g_{s_{i,j,0}}$ and $g_{s_{i,j,k}}$ is provided by constraints (3.46-3.47) and (3.48-3.49), respectively. Finally, $g_{s_{i,j,0}}$ must be 1 if at least one $g_{s_{i,j,k}}$ is 1, as in constraint (3.50).

The Boolean variable a_{s_i,b_r} is 1 if signal s_i needs to be transmitted on bus b_r (needs a message on b_r) and 0 otherwise. constraints (3.51-3.53) encode these conditions.

The Boolean variables $h_{s_{i,j,0},s_m,b_r}$ and $h_{s_{i,j,k},s_m,b_r}$define whether $s_{i,j,0}$ and $s_{i,j,k}$ share the same bus b_r with s_m, respectively. Constraints (3.54-3.59) enforce consistency in the definition of the $h_{s_{i,j,0},s_m,b_r}$ and $h_{s_{i,j,k},s_m,b_r}$ with respect to the signal-to-bus allocation variables.

Utilization constraints

$$z_{\tau_i,e} + \sum_{\tau_j \in \mathcal{T}} a_{\tau_j,e} * c_{\tau_j,e}/t_{\tau_j} \leq u_e \tag{3.60}$$

$$\Delta c_{\tau_i}/t_{\tau_i} - M * (1 - a_{\tau_i,e}) \leq z_{\tau_i,e} \tag{3.61}$$

$$z_{\tau_i,e} \leq \Delta c_{\tau_i}/t_{\tau_i} \tag{3.62}$$

$$z_{\tau_i,e} \leq M * a_{\tau_i,e} \tag{3.63}$$

$$\sum_{s_i \in \mathcal{S}} a_{s_i,b} * c_{s_i}/t_{s_i}/speed_b \leq u_b \tag{3.64}$$

The above constraints enforce the utilization bounds on all nodes and buses considering the load of the current tasks (summation on the left-hand side of Equation (3.60) and the additional load caused by extensions of the execution times ($z_{\tau_i,e}$, on the left-hand side of the equation). u_e and u_b are the utilization bounds on computational node e and bus b, respectively. The additional load caused by the extension Δc_{τ_i} must be considered only if the task is allocated to the node for which the bound is computed. This is represented by using an additional variable $z_{\tau_i,e}$, and the typical "big M" formulation in use in MILP programming for conditional constraints, where M is a large constant.

In our formulation, tasks can have different execution times depending on their allocation, and $c_{\tau_i,e}$ denotes the worst-case execution time of task τ_i on node e. Also, buses can have different speeds. c_{s_i} denotes the transmission time of the message that carries signal s_i on a bus with unit speed. At this stage, we assume each message will only contain one signal. The transmission time of that message on a bus with speed $speed_b$ is $c_{s_i}/speed_b$.

End-to-end latency constraints

$$l_p \le d_p \tag{3.65}$$

$$\begin{aligned}\sum_{\tau_i \in p} r_{\tau_i} + \sum_{s_i \in p} (r_{s_{i,j,0}} + t_{s_i} * g_{s_{i,j,0}} + t_{\tau_j} * g_{s_{i,j,0}} \\ + \sum_k (r_{s_{i,j,k}} + t_{\tau_k} * g_{s_{i,j,k}} + c_{\tau_k} * g_{s_{i,j,k}} \\ + t_{s_i} * g_{s_{i,j,k}})) = l_p\end{aligned} \tag{3.66}$$

$$\sum_{e \in \mathcal{E}} a_{\tau_i,e} * c_{\tau_i,e} + \sum_{\tau_j \in \mathcal{T}} \sum_{e \in \mathcal{E}} c_{\tau_j,e} * p_{\tau_i,\tau_j} * y_{\tau_i,\tau_j,e} = r_{\tau_i} \tag{3.67}$$

$$x_{\tau_i,\tau_j} - M * (1 - h_{\tau_i,\tau_j,e}) \le y_{\tau_i,\tau_j,e} \tag{3.68}$$

$$y_{\tau_i,\tau_j,e} \le x_{\tau_i,\tau_j} \tag{3.69}$$

$$y_{\tau_i,\tau_j,e} \le M * h_{\tau_i,\tau_j,e} \tag{3.70}$$

$$0 \le x_{\tau_i,\tau_j} - r_{\tau_i}/t_{\tau_j} < 1 \tag{3.71}$$

$$r_{\tau_i} \le t_{\tau_i} \tag{3.72}$$

$$\begin{aligned}\sum_{b \in \mathcal{B}} (c_{s_i} + B_{max}) * a_{s_{i,j,0},b}/speed_b \\ + \sum_{s_l \in \mathcal{S}} \sum_{b \in \mathcal{B}} c_{s_l} * p_{s_i,s_l} * y_{s_{i,j,0},s_l,b}/speed_b = r_{s_{i,j,0}}\end{aligned} \tag{3.73}$$

$$x_{s_{i,j,0},s_l} - M * (1 - h_{s_{i,j,0},s_l,b}) \le y_{s_{i,j,0},s_l,b} \tag{3.74}$$

$$y_{s_{i,j,0},s_l,b} \le x_{s_{i,j,0},s_l} \tag{3.75}$$

$$y_{s_{i,j,0},s_l,b} \le M * h_{s_{i,j,0},s_l,b} \tag{3.76}$$

$$\begin{aligned}0 \le x_{s_{i,j,0},s_l} - (r_{s_{i,j,0}} \\ - \sum_{b \in \mathcal{B}} c_{s_i} * a_{s_{i,j,0},b}/speed_b)/t_{s_l} < 1\end{aligned} \tag{3.77}$$

$$r_{s_{i,j,0}} \le t_{s_i} \tag{3.78}$$

$$\sum_{b \in \mathcal{B}} (c_{s_i} + B_{max}) * a_{s_{i,j,k},b}/speed_b$$

$$+ \sum_{s_l \in \mathcal{S}} \sum_{b \in \mathcal{B}} c_{s_l} * p_{s_i,s_l} * y_{s_{i,j,k},s_l,b}/speed_b = r_{s_{i,j,k}} \tag{3.79}$$

$$x_{s_{i,j,k},s_l} - M * (1 - h_{s_{i,j,k},s_l,b}) \leq y_{s_{i,j,k},s_l,b} \tag{3.80}$$

$$y_{s_{i,j,k},s_l,b} \leq x_{s_{i,j,k},s_l} \tag{3.81}$$

$$y_{s_{i,j,k},s_l,b} \leq M * h_{s_{i,j,k},s_l,b} \tag{3.82}$$

$$0 \leq x_{s_{i,j,k},s_l} - (r_{s_{i,j,k}}$$

$$- \sum_{b \in \mathcal{B}} c_{s_i} * a_{s_{i,j,k},b}/speed_b)/t_{s_l} < 1 \tag{3.83}$$

$$r_{s_{i,j,k}} \leq t_{s_i} \tag{3.84}$$

Latency constraints are derived from Equations (3.33), (3.2) and (3.35). Equation (3.33) shows the calculation of end-to-end latency for path p. For each signal s_i on path p, we know its destination task and denote it as τ_j. If two tasks are on computation nodes connected to different buses, they will communicate through a gateway task and the corresponding additional latencies need to be considered. The calculation of end-to-end latency is shown in constraint (3.66). We assume the response time for gateway task τ_k is c_{τ_k} (i.e. a gateway task has the highest priority on its node).

r_{τ_i} is the response time of task τ_i, defined by constraints (3.67-3.72), x_{τ_i,τ_j} represents the number of possible interference from τ_j to τ_i and p_{τ_i,τ_j} is a parameter that denotes whether task τ_j has higher priority than task τ_j. A large constant M is used to linearize the relation $y_{\tau_i,\tau_j,e} = x_{\tau_i,\tau_j} * h_{\tau_i,\tau_j,e}$, which defines the number y of actual interferences by higher priority tasks, similarly as in the utilization constraints. Constraint (3.72) is used to enforce the task's response time is no larger than its period, which is the assumption for our response time calculation in Equation (3.2).

$r_{s_{i,j,0}}$ and $r_{s_{i,j,k}}$ represent the response times of the messages that carry signals $s_{i,j,0}$ and $s_{i,j,k}$, respectively. Their definitions are in constraints (3.73-3.84). If signals are local, the corresponding response times will be 0. p_{s_i,s_j}, $y_{s_{i,j,0},s_j,b}$ and $y_{s_{i,j,k},s_j,b}$ are similarly defined as in the definition of task response time. Constraints (3.78) and (3.84) enforce the assumption that message response times should not exceed their periods.

Objective function

$$max. \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} * \Delta c_{\tau_i}/t_{\tau_i} \tag{3.85}$$

We recall here the objective function in (3.30), which represents the task extensibility. An alternative objective function can also include the optimization of latency, as shown in (3.86). K is the parameter used to explore the trade-off between task extensibility and latencies. The special case $K = 0$ is the original objective function (3.85).

$$max. \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} * \Delta c_{\tau_i}/t_{\tau_i} - K * \sum_{p \in \mathcal{P}} l_p/d_p \tag{3.86}$$

In Section 3.3.3, we will report the experimental results with various values of K, to show the relationship between task extensibility and path latencies.

An alternative to the MILP optimization for initial task allocation is to use heuristics. We designed a greedy heuristic algorithm as shown in [77] and compared it with the MILP optimization. Although the heuristic algorithm is more efficient, the results are much worse. Of course, in principle it is possible to design a better heuristic, but this task is expected to be quite difficult considering the need to balance trade-offs between feasibility and extensibility and the need to cope with gatewaying (for which the MILP formulation provides intuitive solutions). For more details, please refer to [77].

3.3.2.2 Signal Packing and Message Allocation

After the allocation of tasks is chosen, we use a simple heuristic to determine the signal packing and message allocation. The steps are shown below.

1. Group the signals with the same source node and period as packing candidates.

2. Within each group, order the signals based on their priorities, then pack them according to the message size constraints (priorities are assumed given from an existing configuration or some suitable policy, as in the initial task allocation). The priority of a message is set to the highest priority of the signals that are mapped into it.

3. Assign a weight w_{m_i} to each message m_i based on its priority, transmission time and period. In our algorithm, we set $w_{m_i} = k_1/p_{m_i} + k_2 * c_{m_i}/t_{m_i}$, where p_{m_i}, c_{m_i} and t_{m_i} are priority, transmission time on bus with unit speed and period of the message. k_1 and k_2 are constants, whose values are tuned in case studies (both set to 1 in our experiments). When multiple buses are available between the source and destination nodes, we allocate messages to buses according to their weights. Messages with larger weights are assigned first to faster buses.

Other more sophisticated heuristics or mathematical programming solutions have been considered. For instance, signal packing can be formulated as MILP as in [76]. However, from preliminary experiments, there is no significant improvement that can outweigh the speed of this simple strategy.

3.3.2.3 Priority Assignment

In this step, we assign priorities to tasks and messages, given the task allocation, signal packing and message allocation obtained from previous steps.

This priority assignment problem is proven to be NP-complete [25]. Finding an optimal solution is generally not feasible for industrial-sized problems. Therefore, we propose an iterative heuristic to solve the problem.

The flow of this heuristic is shown in Figure 3.6. The basic idea is to define the local deadlines of tasks and messages over iteration steps, then assign priorities based on the deadlines. Intuitively, shorter deadlines require higher priorities and longer local deadlines can afford lower priorities.

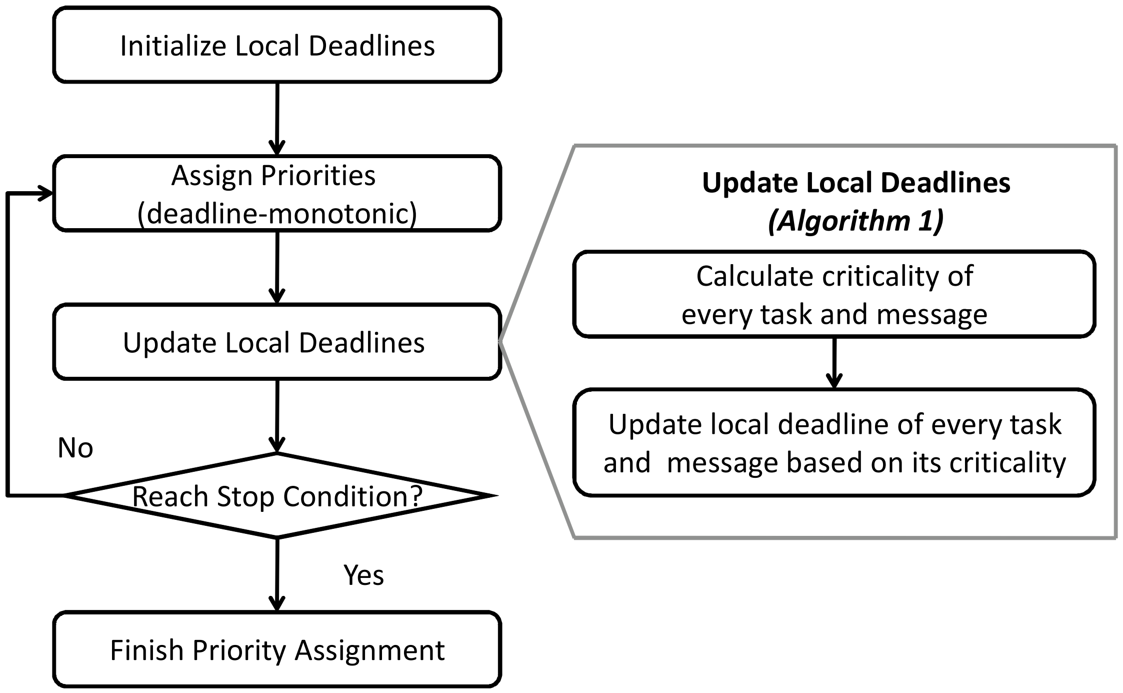

Initialize Local Deadlines
Assign Priorities (deadline-monotonic)
Update Local Deadlines
No
Reach Stop Condition?
Yes
Finish Priority Assignment
Update Local Deadlines (Algorithm 1)
Calculate criticality of every task and message
Update local deadline of every task and message based on its criticality

Figure 3.6: Iterative priority assignment algorithm

Initially, the deadlines of tasks and messages are the same as their periods. Then, deadlines are modified, and priorities are assigned using the deadline-monotonic (DM) approach [12]. Of course, there is no guarantee that the DM policy is optimal in this case as for any system with non-preemptable resources (the CAN bus), but there is no optimal counterpart that can be used here, and DM is a sensible choice in the context of our heuristics.

During the iterations, deadlines are changed based on task and message *criticality*, as shown in Algorithm 1 and explained below.

The criticality of a task or message, reflects how much the response times along the paths to which it belongs are affected by extensions in the execution times of other tasks. Tasks and messages with higher criticality are assigned higher priorities. To define the criticality ϵ of a task or a message, we increase the execution time of each task τ_i, by $UB(\Delta c_{\tau_i})$, the maximum amount allowed by utilization constraints and an upper bound of task execution time slack, as shown in line 3 and 4 in Algorithm 1. Then, the response time of τ_i and of

Algorithm 1 Update Local Deadline (K_1)

```
1: Initialize the criticality $\epsilon$ of every task and message to 0
2: for all task $\tau_i$ do
3:   $UB(\Delta c_{\tau_i}) = t_{\tau_i} * (u_e - \sum_{\tau_j \in \mathcal{T}(e)} c_{\tau_j}/t_{\tau_j})$
4:   $c_{\tau_i} = c_{\tau_i} + UB(\Delta c_{\tau_i})$
5:   for all task $\tau_j \in (lp(\tau_i) \cup \{\tau_i\})$ do
6:     update $r_{\tau_j}$
7:   for all path $p$ whose latency is changed do
8:     if $l_p > d_p$ then
9:       for all tasks and messages $o_j$ on $p$ do
10:        $\epsilon_{o_j} = \epsilon_{o_j} + w_{\tau_i} * (l_p - d_p)/t_{o_j}$
11:  reset all $c_{\tau_i}$, $r_{\tau_i}$, $l_p$ to the values before the iteration
12: for all task $\tau_i$ do
13:   $\epsilon^N_{\tau_i} = \epsilon_{\tau_i}/max_{\tau_i \in \mathcal{T}}\{\epsilon_{\tau_i}\}$
14:   $d_{\tau_i} = d_{\tau_i} * (1 - K_1 * \epsilon^N_{\tau_i})$
15: for all message $m_i$ do
16:   $\epsilon^N_{m_i} = \epsilon_{m_i}/max_{m_i \in \mathcal{M}}\{\epsilon_{m_i}\}$
17:   $d_{m_i} = d_{m_i} * (1 - K_1 * \epsilon^N_{m_i})$
```

lower priority tasks on the same node as τ_i is recomputed (line 5 and 6). The criticality of the affected task τ_j or message m_j (both generically denoted as object o_j) is defined by adding up a term $w_{\tau_i} * (l_p - d_p)/t_{o_j}$ for each path p whose end-to-end latency exceeds the deadline after the increase $UB(\Delta c_{\tau_i})$, where w_{τ_i} is the weight of τ_i (line 7 to 10). After repeating this operation for every task, the criticality of all tasks and messages is computed, denoted by ϵ_{o_j}. Criticality values are normalized, obtaining a value ϵ^N for each task and message and finally, local deadlines are computed as $d = d * (1 - K_1 * \epsilon^N)$ (line 11 to 16). The procedure is shown in Algorithm 1. The parameter K_1 is initially set to 1, then adjusted in the later iteration steps using a strategy that takes into account the number of iteration steps, the number of times the current best solution is found, and the number of times the priority assignment remains unchanged.

As shown in Figure 3.6, after local deadlines are updated, the stop condition for priority assignment is checked. If the number of iterations reaches its limit, or the upper bound of task extensibility is reached, the priority assignment will finish, otherwise we keep iterating.

The strategy of changing priorities based on local deadlines can also be found in [35]. Different from our algorithm, the goal is only to meet end-to-end latency constraints, therefore deadlines are updated based on the slack time of tasks or messages which indicate how much the local deadlines can be increased without violating latency constraints.

3.3.2.4 Task Re-allocation

As shown in Figure 3.4, after all the design variables are decided, we calculate the value of the objective function in Formula 3.30, and check the stop condition for our entire algorithm. If the results are not good enough and the iteration limit has not been exceeded, we re-allocate the tasks and repeat the signal packing, message allocation and priority assignment.

We could use the same MILP based method for re-allocating tasks, with additional constraints to exclude the allocations that have been considered. However, solving the MILP is time consuming. To speed up the algorithm, we designed a local optimization heuristic that leverages the results of previous iterations for the task re-allocation step in Figure 3.4. The details of this heuristic are shown in Algorithm 2.

Algorithm 2 Task Re-allocation (K_2)

```
Let $\Phi(M) = \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} * UB(\Delta c_{\tau_i})/t_{\tau_i} - K_2 * \sum_{p \in \mathcal{P}} l_p/d_p$ for a mapping $M$
 1: **if** current solution does not satisfy latency constraints **then**
 2:     $K_2 += K_C$
 3: $\Delta_{best} = MIN$
 4: **for all** task $\tau_i$ and node $e$ that $\tau_i$ is not on $e$ **do**
 5:     $\Delta_{\tau_i,e} = \Phi(M') - \Phi(M)$ {where $M$ is the original mapping, $M'$ is the new mapping after
        moving $\tau_i$ to $e$}
 6:     **if** $\Delta_{\tau_i,e} > \Delta_{best}$ **then**
 7:         *best_move* = $\tau_i$ *moves to* $e$
 8:         $\Delta_{best} = \Delta_{\tau_i,e}$
 9: **for all** task $\tau_i$, $\tau_j$ that are not on the same node **do**
10:     $\Delta_{\tau_i,\tau_j} = \Phi(M') - \Phi(M)$ {$M$, $M'$ similarly defined as above}
11:     **if** $\Delta_{\tau_i,\tau_j} > \Delta_{best}$ **then**
12:         *best_move* = *switch* $\tau_i$ *and* $\tau_j$
13:         $\Delta_{best} = \Delta_{\tau_i,\tau_j}$
14: *Execute best_move*
```

Two operators are considered for generating new configurations: moving one task to a different node, or switching two tasks on different nodes. For each possible application of the operators on each task or task pair, that satisfies the utilization constraints, we compute the corresponding change of the performance function Φ of Equation (3.86), which includes the consideration of task extensibility and end-to-end latencies. In the case of multiple buses, the changes might lead to signal forwarding through gateway tasks, and this is taken into account in the calculation. Finally the change that provides the largest increase of the performance function is selected.

Parameter K_2 in cost function Φ provides the trade-off between task extensibility and end-to-end latencies. Initially, it is set to the same value as parameter K in Equation (3.86), which is used in the initial task allocation. If the current solution does not satisfy the end-to-end deadlines, we increase K_2 by a constant K_C to emphasize the optimization of latencies (K_C was tuned to 0.05 in our experiments).

3.3.2.5 Algorithm Complexity

The algorithm shown in Figure 3.4 is polynomial except for the MILP based initial task allocation, which can be regarded as a preprocessing stage since we use heuristics for task re-allocation in following iterations.

Finding the *optimal* initial task allocation by MILP is a NP-hard problem. In practice, we set a timeout and use the best feasible solution. For the following steps, let $N_{\mathcal{S}}$ denote the number of signals, $N_{\mathcal{T}}$ denote the number of tasks, $N_{\mathcal{E}}$ denote the number of computational nodes, $N_{\mathcal{B}}$ denote the number of buses, and $N_{\mathcal{P}}$ denote the number of paths. The complexity of the signal packing and message allocation stage is $O(N_{\mathcal{S}} * log(N_{\mathcal{S}}) + N_{\mathcal{S}} * N_{\mathcal{B}})$. The complexity of the priority assignment is $O(N_{\mathcal{T}}*N_{\mathcal{P}}*(N_{\mathcal{T}}+N_{\mathcal{S}})+N_{\mathcal{S}}*log(N_{\mathcal{S}})+N_{\mathcal{T}}*log(N_{\mathcal{T}}))$ assuming the number of iterations in Figure 3.6 is within a constant (as stated in Section 3.3.2.3, there is a preset limit of number of iterations when checking the end condition). And the complexity of heuristic task re-allocation stage is $O(N_{\mathcal{E}} * N_{\mathcal{T}} * N_{\mathcal{P}} * (N_{\mathcal{T}} + N_{\mathcal{S}}) + N_{\mathcal{T}} * N_{\mathcal{T}} * N_{\mathcal{P}} * (N_{\mathcal{T}} + N_{\mathcal{S}}))$. This is the dominant stage.

If we assume $N_{\mathcal{S}} \in O(N_{\mathcal{T}}^2)$, $N_{\mathcal{T}} \in O(N_{\mathcal{S}})$ and $N_{\mathcal{B}} \in O(N_{\mathcal{E}})$, which is usually the case in practice, we can simplify the complexity of the entire algorithm (excluding the MILP based preprocessing stage) as $O(N_{\mathcal{E}} * N_{\mathcal{T}} * N_{\mathcal{P}} * N_{\mathcal{S}} + N_{\mathcal{T}} * N_{\mathcal{T}} * N_{\mathcal{P}} * N_{\mathcal{S}})$, assuming the number of iterations in Figure 3.4 is within a constant.

3.3.2.6 Extensibility Metric for Multiple Tasks

When using the extensibility metric for task groups defined in Formula (3.32), the optimization algorithm introduced in the previous sections needs to be modified as follows.

In the MILP formulation, the utilization constraints from formula (3.60) to (3.61) should be modified to:

$$\sum_{\tau_j \in \mathcal{T}_{g_i}} z_{\tau_j,g_i,e} + \sum_{\tau_j \in \mathcal{T}} a_{\tau_j,e} * c_{\tau_j,e}/t_{\tau_j} \leq u_e \tag{3.87}$$

$$\Delta c_{\tau_j,g_i}/t_{\tau_j} - M * (1 - a_{\tau_j,e}) \leq z_{\tau_j,g_i,e} \tag{3.88}$$

$$z_{\tau_j,g_i,e} \leq \Delta c_{\tau_j,g_i}/t_{\tau_j} \tag{3.89}$$

$$z_{\tau_j,g_i,e} \leq M * a_{\tau_j,e} \tag{3.90}$$

Equation (3.31) needs to be added to the MILP formulation. Objective function (3.85) should be replaced with the new objective (3.32), and objective (3.86) should be replaced with

$$max. \sum_{g_i \in \mathcal{G}} w_{g_i} * \Delta c_{g_i} - K * \sum_{p \in \mathcal{P}} l_p/d_p \tag{3.91}$$

The allocation and the end-to-end latency constraints in the MILP formulation do not change since they are only related to the original execution times.

In Algorithm 1, the criticality calculation from line 2 to 11 needs to be adjusted as follows.

```
1: for all task groups $g_i$ do
2:    $UB(\Delta c_{g_i}) = min_e\{\frac{u_e - \sum_{\tau_j \in \mathcal{T}(e)} c_{\tau_j}/t_{\tau_j}}{\sum_{\tau_j \in \mathcal{T}(e) \wedge \mathcal{T}_{g_i}} A_{\tau_j,g_i}/t_{\tau_j}}\}$
3:    for all task $\tau_j \in \mathcal{T}_{g_i}$ do
4:       $c_{\tau_j} = c_{\tau_j} + A_{\tau_j,g_i} * UB(\Delta c_{g_i})$
5:    for all task $\tau_i \in \bigcup_{\tau_j \in \mathcal{T}_{g_i}} (lp(\tau_j) \cup \{\tau_j\})$ do
6:       update $r_{\tau_i}$
7:    for all path $p$ whose latency is changed do
8:       if $l_p > d_p$ then
9:          for all tasks and messages $o_j$ on $p$ do
10:            $\epsilon_{o_j} = \epsilon_{o_j} + w_{g_i} * (l_p - d_p)/t_{o_j}$
11:   reset all $c_{\tau_i}$, $r_{\tau_i}$, $l_p$ to the values before the iteration
```

where $UB(\Delta c_{g_i})$ is an upper bound of Δc_{g_i} considering only the utilization constraints. For each g_i, we compute $UB(\Delta c_{g_i})$ (line 2), and increase the execution times of all the tasks in g_i (line 3 and 4). Then, the response times of all the tasks being affected are updated (line 5 and 6), and the criticality of each task or message is updated (line 7 to 10).

Also, in Algorithm 2, the definition of $\Phi(M)$ needs to be changed to $\Phi(M) = \sum_{g_i \in \mathcal{G}} w_{g_i} * UB(\Delta c_{u_i}) - K_2 * \sum_{p \in \mathcal{P}} l_p/d_p$ to reflect the change of the objective function.

Finally, in the calculation of eventual objective value, both utilization and end-to-end latency constraints need to be considered to calculate Δc_{g_i}. A bisection algorithm is used for approximating Δc_{g_i} considering all tasks in g_i (a bisection algorithm is also used for approximating Δc_{τ_i} in the original objective function).

The experimental results for this metric based on task groups are shown in [77].

3.3.3 Case Studies

The effectiveness of the methodology and algorithm is validated in this section with three industrial case studies. The first two cases focus on improving the extensibility for two automotive architecture options, whereas the third case study investigates the impact of additional resources on the optimality of the design of a truck control system.

3.3.3.1 Active Safety Vehicle

In the following two case studies, we apply our algorithm to an experimental vehicle that incorporates advanced active safety functions. This is the same example studied in [76].

We considered two architecture platform options with different number of buses. Both options consist of 9 ECUs (computational nodes). In the first configuration, they are connected through a single CAN bus; in the second by two CAN buses, with one ECU functioning as gateway between the two buses. The transmission speed is 500kb/s. The vehicle supports advanced distributed functions with end-to-end computations collecting data from 360° sensors to the actuators, consisting of the throttle, brake and steering subsystems and of advanced HMI (Human-Machine Interface) devices.

The subsystem that we considered consists of a total of 41 tasks executed on the ECUs, and 83 signals exchanged between them. Worst-case execution time (WCET) estimates have been obtained for all tasks. In our formulation shown before, the WCET of a task on each ECU is distinguished (denoted by $c_{\tau_i,e}$). For the purpose of our algorithm evaluation, we assumed that all ECUs have the same computational power, so that the worst case execution time of tasks does not depend on their allocation. This simplification does not affect the complexity or the running time of the optimization algorithm and is only motivated by the lack of WCET data for the tasks on all possible ECUs. The bit length of the signals is between 1 (for binary information) and 64 (full CAN message). The utilization upper bound of each ECU and bus has been set to 70%. The task weights w_{τ_i} in Equation (3.30) are set to 1 for all tasks.

End-to-end deadlines are placed over 10 pairs of source-sink tasks in the system. Most of the intermediate stages on the paths are shared among the tasks. Therefore, despite the small number of source-sink pairs, there are 171 unique paths among them. The deadline is set at 300ms for 8 source-sink pairs and 100ms for the other two.

The experiments are run on a 1.7-GHz processor with 1GB RAM. CPLEX [52] is used as the MILP solver for the initial task allocation. The timeout limit is set to 1000 seconds. The parameter K in the MILP formulation is used to explore the trade-off between task extensibility and end-to-end latencies during initial task allocation. We test our algorithm with several different K values, and compare them with a system configuration produced manually. Results are shown in Figure 3.7.

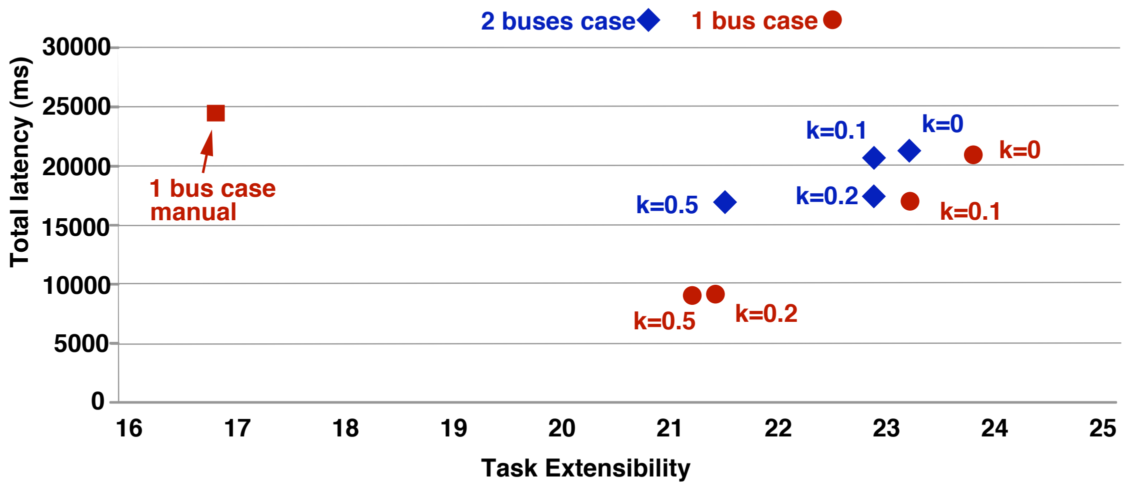


Figure 3.7: Comparison of manual and optimized designs for the two architecture options

A manual design is available for the single-bus configuration and consists of the configuration of the tasks and messages provided by its designers. This initial configuration is not optimized. The total latencies of all paths is 24528.1ms and the task extensibility is 16.9113.

For the single-bus case, in any of the four automatically optimized designs, all paths meet their deadlines. Different K values provide the trade-off between task extensibility and end-to-end latencies. When $K = 0$, we have the largest task extensibility at 23.8038, which is a 41% improvement over manual design. When $K = 0.5$, we have the shortest total end-to-end latency at 9075.46ms, which is 63% less than manual design. If a balanced design between extensibility and end-to-end latency is needed, intermediate values may be used. For $K = 0.1$, we obtain 37% improvement on task extensibility and 31% improvement on end-to-end latencies.

For the two-buses case, again all optimized designs satisfy the end-to-end latency constraints. When $K = 0$, the largest task extensibility obtained after optimization is 23.1347. When $K = 0.5$, we have the shortest total end-to-end latency at 16948.1ms. If a balanced design is needed, intermediate values may be used, with the results shown in Figure 3.7.

Comparing single-bus and two-buses case, the results of two-buses case have longer latencies in general. This is because of the additional time taken on gateway tasks and signals. Also, the two-buses results span across a smaller range of extensibility and latency than the single-bus results. This is because the configurations for two-buses case are less flexible due to the constraints from allocation and gatewaying.

For both configurations, after the initial task allocation, each outer iteration of the signal packing and message allocation, priority assignment and task re-allocation takes less than 30 seconds, and the optimization converges within 30 iterations for the K values we tested. Figure 3.8 shows the current best task extensibility over 30 iterations for $K = 0$ for the two architecture options. Iteration 0 is the task extensibility after initial task allocation. The running time is 732 seconds for 30 iterations in the case of single bus, and 545 seconds for 30 iterations in the case of two buses.

To evaluate the quality of above results, we compared our algorithm performance with a simulated annealing algorithm as shown in [77]. The maximum task extensibility values obtained from the optimization algorithm (when $K = 0$), and from the simulated annealing algorithm are extremely close. This fact, together with the way both algorithms converge to their final result, suggests that the obtained values and configurations are very close to the true optimum (although final proof cannot be obtained unless all combinations are evaluated, which is clearly impossible in a feasible time). Furthermore, as mentioned earlier, our algorithm is much faster than the simulated annealing algorithm. More detalis can be found in [77].

3.3.3.2 Distributed Control System

In addition to the active safety vehicle application, another example is presented: a safety-critical distributed control system deployed within a small truck. This is a CAN based system that implements a distributed closed-loop control. The key features of this system are the integration of slow and very fast (power electronics) control loops using the same communication network. In this example, we are interested in redesigning an existing sys-

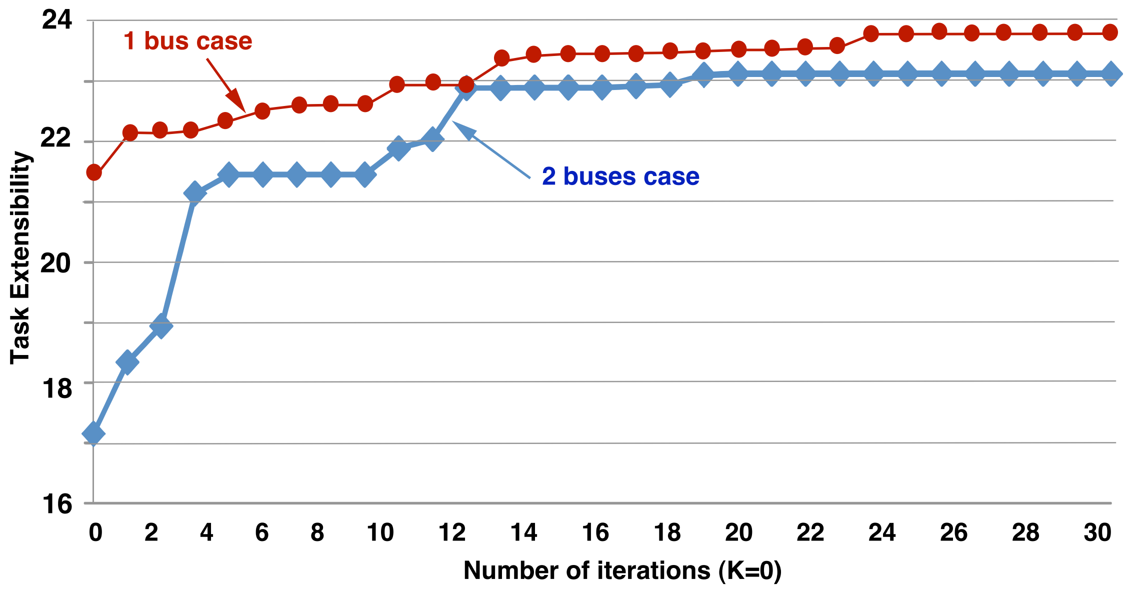


Figure 3.8: Task extensibility over iterations

tem to understand the effects of adding communication and computational resources to the system.

The system implements several control loops, such as the power electronics control, and diagnostic features. To protect sensitive confidential data obtained by a major automobile manufacturer, the system is abstracted as a set of tasks with aggregate information. Table 3.3 summarizes information about the test case. Task periods range from 10 to 1000 ms.

# of tasks	# of signals	# of paths	# of nodes	# of buses
43	68	15	7 - 8	1

Table 3.3: Description of distributed control system

The example system is evaluated for an initial system configuration consisting of 7 nodes, and a derived system where one additional node is provided for additional flexibility. The optimization algorithm must define a new allocation of tasks to maximize extensibility on the new architecture. The average task utilization is 0.05. In the initial configuration with 7 nodes, the average CPU utilization is 0.307, with a maximum of 0.45 and a minimum of 0.25. Results are shown in Figure 3.9. Solid lines indicate the mapping of tasks (indicated by T#) to the 7 nodes in the initial configuration, whereas dotted lines indicate the mapping computed by the algorithm for the extended configuration. In the optimized configuration with 8 nodes, utilization is 0.269, with a maximum of 0.30 and a minimum of 0.20. The task extensibility values for the two systems are 12.15 and 13.97, respectively. The timeout limits of the MILP for initial task allocation in the two cases are both set to 1000 seconds. The running time of the rest of flow shown in Figure 3.4, which includes 30 iterations of signal packing and message allocation, priority assignment and task-reallocation, is 298 seconds for the first case (7 nodes) and 346 seconds for the second case (8 nodes).

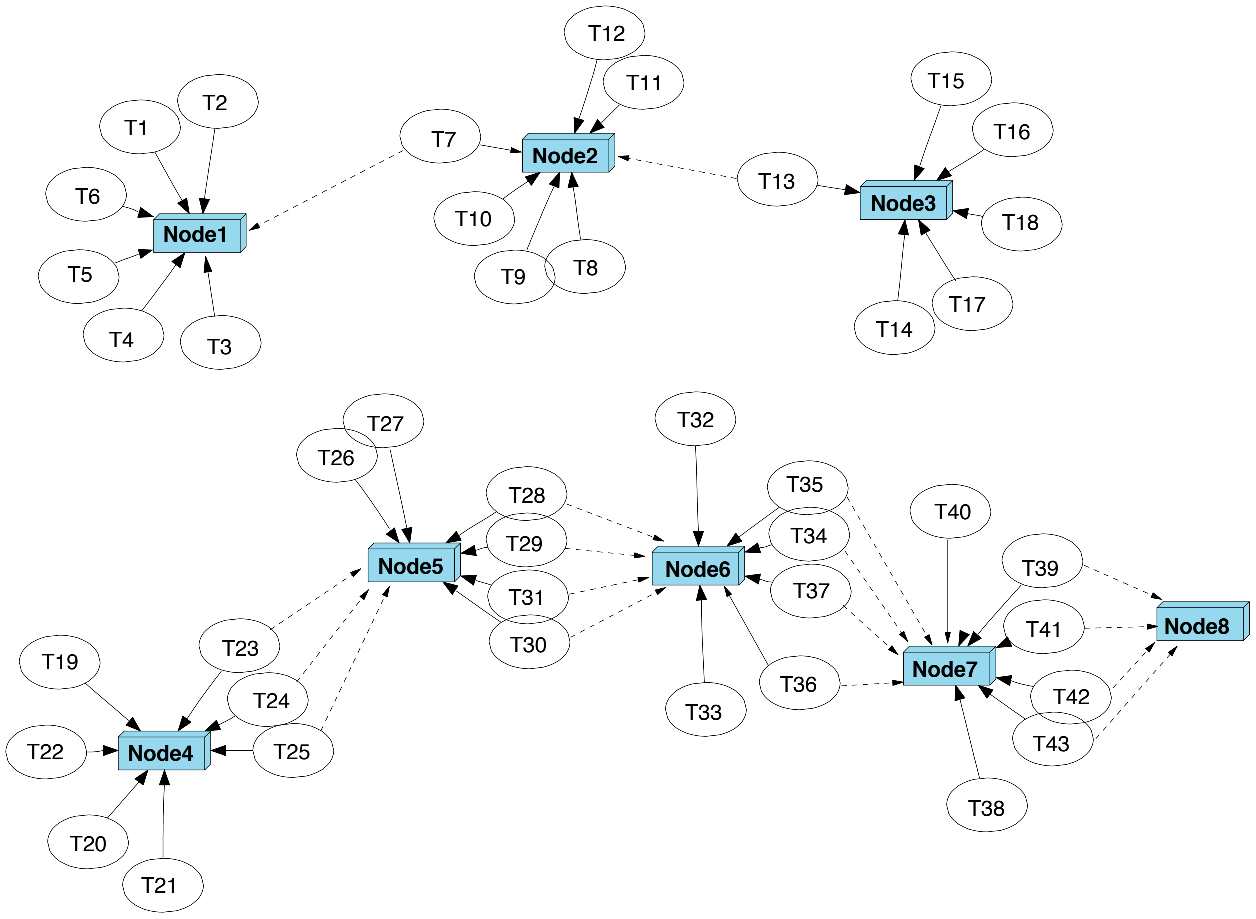


Figure 3.9: Reallocation of tasks for increased computational resources

Chapter 4

Code Generation and Communication Interface Synthesis

After the functional model is mapped onto the architecture platform, the Step 3 in our software synthesis flow includes code generation for individual functional tasks and the synthesis of communication interfaces between tasks.

4.1 Code Generation

Code generation translates the processes (corresponding to functional tasks) in IF representation to code in target languages. Based on the allocation result from mapping, the target language may be generic C code or vendor specific representation for the mapped embedded processor. Translating into vendor specific languages enables leveraging vendor tools for analysis, debugging and simulation. As a proof of concept, we developed a translator in ANTLR for translating IF to EIKON, a language for modeling BAC systems. The translator includes a lexer, a parser and a code generator. The lexer and parser parse the designs described in IF to an abstract syntax tree (AST), from which the code generator generates the target EIKON description. EIKON provides a library of microblocks (control functions) for developing various control sequences. A process will be translated into the microblock that implements the same functionality. In the case that a process does not have corresponding microblock in the EIKON library, the translator implements its functionality in OCL (Operator's Control Language) defined in EIKON. OCL provides a number of mathematical and logical functions in the syntax. For the set of equations in a process, the translator constructs an OCL block by mapping the equations to a set of functions provided in OCL.

As an example, the following code snippet shows part of the ANTLR input for generating lexer, parser and generator for translating the *process* entity in IF into EIKON.

```
proc
    : 'process' ID 'extends' proc_type '{' proc_body '}'
```

```
      -> ^(PROCDEF ID proc_type proc_body)
    ;

proc_type
    : 'DTProcess' | 'CTProcess'
    ;

proc_body
    : input_decl* output_decl*
      para_decl*
      local_var_decl*
      para_var_assign*
      init* eqn*
    ;

......

proc
@init{
    proc_inLib = false;
    in_proc = true;
}
@after{
    proc_inLib = false;
    in_proc = false;
    System.out.println();
}
    : ^(PROCDEF ID
      {
        if (eikonLib.get($ID.text) != null) {
          System.out.println("Choose microblock \""
            + eikonLib.get($ID.text) + "\" from EIKON library");
          proc_inLib = true;
        }
        else {
          System.out.println("No microblock can be directly used for "
            + $ID.text + ". build OCL:");
          System.out.println("TITLE " + $ID.text);
          ......
          proc_inLib = false;
       }
      }
      proc_type proc_body)
    ;
```

```
input_decl
    : ^(INPUTDECL intf_type intf_data_type ID)
      {
        if (in_proc == true){
          if (proc_inLib == true) {
            if ($intf_type.text.equals("DTInterface")) {
              System.out.print("digital input");
              System.out.print(" " + $ID.text);
              System.out.println(";");
            }
            else if ($intf_type.text.equals("CTInterface")) {
              System.out.print("analog input");
              System.out.print(" " + $ID.text);
              System.out.println(";");
            }
            else
              System.out.println("unrecognized type");
          }
          else {
            if ($intf_type.text.equals("DTInterface")) {
              System.out.print("DINPUT ");
              System.out.println($ID.text);
            }
            else if ($intf_type.text.equals("CTInterface")) {
              System.out.print("AINPUT ");
              System.out.println($ID.text);
            }
            else
              System.out.println("unrecognized type");
          }
        }
      }
    ;
```

4.2 Communication Interface Synthesis

Communication interface synthesis focuses on the communication mechanism between tasks. The goal is to preserve the semantics of the input functional model when the architecture platform does not directly support it. A typical case is that the functional model is synchronous, which eases the design by orthogonalizing functionality and timing, while the architecture platform is distributed and asynchronous.

A method is proposed in [69] to implement synchronous functional models on a Loosely

Time Triggered Architecture (LTTA) [19] 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 distribution method in [69] 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. Both reads and writes are non-blocking in an FFP and the processes have the responsibility for checking that the queue is non-empty before doing a read, and that the queue is non-full before doing a write. 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.

Note that the method proposed in [69] only applies to close systems. For open systems, which take into account the physical environment, more constraints are needed to guarantee stream equivalence. For example, a room temperature control system that interacting with the environment as shown in Figure 2.1 is an open system, and only applying the method proposed in [69] cannot guarantee stream equivalence. To cope with open systems, we extend the method by enforcing additional timing constraints. The detail is discussed in Section 4.2.1.

The second extension focuses on optimizing communication for Triggered Synchronous Block Diagrams (Triggered SBDs). The semantics-preserving distribution method proposed in [69] applies to "pure" SBDs, where all blocks fire at every synchronous step. The method can be adapted to Triggered SBDs by using trigger elimination [45], where triggers are transformed to standard inputs, however this often results in unnecessary communication overhead. In our work discussed in Section 4.2.2, we present a distribution method that eliminates this overhead by exploring trigger information, and therefore minimize the message load while still preserving the semantics. We consider both general Triggered SBDs where the values of triggers are dynamically computed and are thus not known a-priori, as well as Timed SBDs where triggers are statically known, usually specified by (period, initial phase) pairs.

4.2.1 Communication Interface Synthesis for Open Systems

We observed that the assumption that every process (or task after mapped to LTTA) can freely skip a round does not hold if we want to preserve stream equivalence for open systems that interact with physical environment. 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 longer equivalent to the synchronous model. 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.

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 [69] in the implementation of $n$on-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 [72, 71]. 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 [34]. 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.

4.2.1.1 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 (4.2) and (4.1), similarly as in [34].

$$t_{\tau_i}(n) \in [\hat{t}_{\tau_i}(n),\ \hat{t}_{\tau_i}(n) + J_{\tau_i}] \tag{4.1}$$

$$\hat{t}_{\tau_i}(n+1) - \hat{t}_{\tau_i}(n) \in [T^m_{\tau_i},\ T^M_{\tau_i}],$$
$$T^m_{\tau_i} = T_{\tau_i}(1-\delta^m_{\tau_i}),\ T^M_{\tau_i} = T_{\tau_i}(1+\delta^M_{\tau_i}) \tag{4.2}$$

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 [21, 34]. Specifically, for a pair of source task τ_w and destination task τ_r communicating through global messages on a communication link, Formula (4.3) 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.

$$T^M_{\tau_r} + J^M_{\tau_r} < T^m_{\tau_w} + (J^m_{\tau_w} + l^m_{mg}) - (J^M_{\tau_w} + l^M_{mg}) \tag{4.3}$$

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 (4.3) 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 (4.3) 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 (4.3) becomes trivial.

The control mechanism from [69] and timing constraint (4.3) 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,

$$\forall n, \quad l_{\tau_i \to \tau_k} < t_{\tau_j}(n+1) - t_{\tau_i}(n) \tag{4.4}$$

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).

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 (4.5), 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 (3.26). Since communication interface synthesis is done after mapping, we will be able to have an accurate estimation of all the parts in Equation (3.26), including message latencies.

$$l_\rho < (k_\rho + 1) * T^m_{src_\rho} \tag{4.5}$$

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 (4.3).

We have assumed all sensing and actuating tasks have the same period in the functional model. If this is not the case, constraint (4.5) 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 (4.3), (4.4) and (4.5), we may need to adjust task periods and the drifts of local clocks. For instance, constraint (4.4) 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 communication interface 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.

4.2.1.2 Case Study

We conducted experiments to demonstrate the impact of timing constraints on semantic preservation for the room temperature control system shown in Figure 2.1, which is an open system interacting with the physical environment constantly. We chose LabVIEW from National Instruments (NI) as the target platform, and generate code in NI's G language, for which both simulation and C code generation are provided by LabVIEW. 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 2.5.

Specifically, the mapped model in LabVIEW is shown in Figure 4.1. 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 [69] is added to all tasks except the sensing tasks.

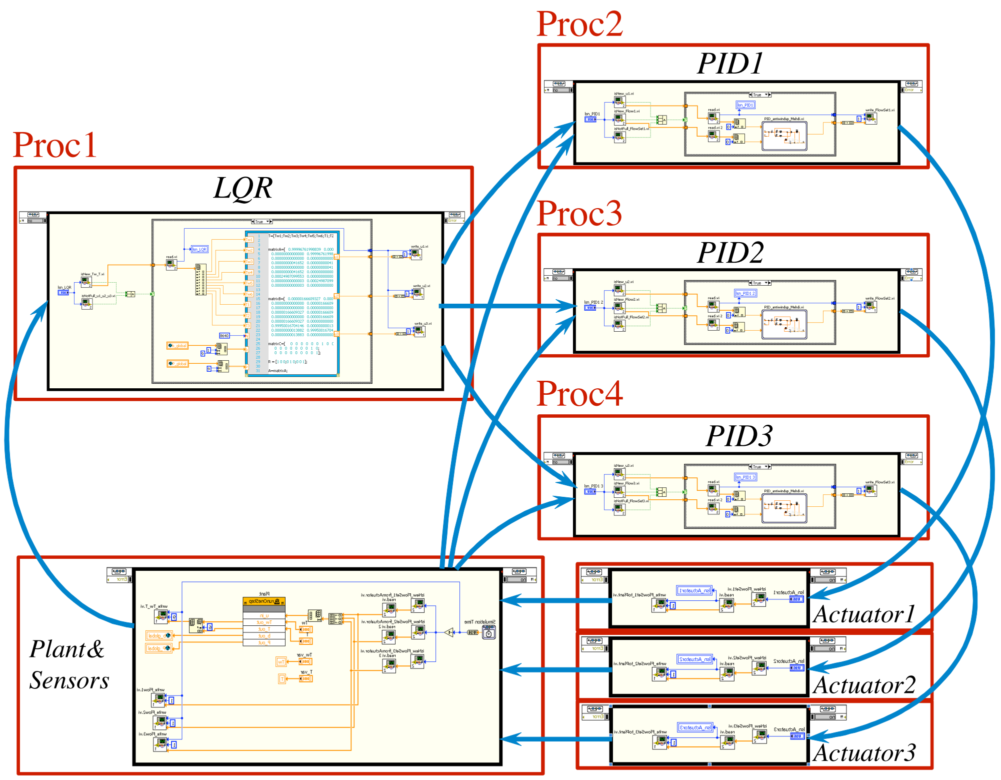


Figure 4.1: Mapped LabVIEW model for the temperature control system

The local clocks are set to have no clock drifts. In this case, the constraint described in Equation (4.3) for avoiding data loss can be simplified to Equation (4.6), given the fact that $l_{mg}^m \geq 0$, and for any task τ_i, $J_{\tau_i}^m \geq 0$ and $J_{\tau_i}^M = r_{\tau_i}$.

$$T_{\tau_r} + r_{\tau_r} < T_{\tau_w} - r_{\tau_w} - l_{mg}^M \tag{4.6}$$

The communication between processors are through shared variables, therefore $l_{mg}^M = 0$. 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 (4.6) and (4.5) 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 (4.6) does not always hold, and the simulation results of the two models become different as shown in Table 4.1. The difference is acceptable though for temperature control system. This shows that for some applications, stream equivalence can be relaxed.

Table 4.1: 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 $\times 10^{-4}$

4.2.2 Communication Interface Synthesis for Triggered and Timed Synchronous Block Diagrams

The other problem we address is semantics-preserving and communication-efficient distribution of Triggered SBDs on asynchronous execution platforms. In particular, given a design specification described as a Triggered SBD, how to map it to a distributed, asynchronous execution platform, so that the semantics of the Triggered SBD is *stream-equivalent* to the semantics of the distributed implementation, and the communication overhead between the distributed processes is reduced.

The semantics-preserving distribution problem has been studied in [69], but only for a "pure" SBD model, where all blocks fire at every synchronous step. In this work we generalize these results to the case of Triggered SBDs. We also study distribution of Timed SBDs as a special case, for which more efficient implementations can be obtained.

At the heart of many synchronous languages such as Simulink, SBDs are usually chosen as the model of computation, because they facilitate formal analysis of the system behavior and verification of the design correctness. The fundamental component in an SBD is a *block*, which can be modeled as a (not necessarily finite) state machine with inputs and outputs à la Mealy. Outputs of blocks can be connected to inputs of other blocks to form a diagram. The semantics of such diagrams are *synchronous* in the sense that all blocks proceed in *lock-step*. Provided the diagram has no cyclic dependencies (within a step), all blocks "fire" in a certain order within a synchronous step, so that the external outputs of the diagram are computed by propagating the external inputs throughout the diagram. The firing of a block corresponds to a local reaction step of the corresponding state machine: the machine reads its local inputs, computes its local outputs and updates its local state.

Triggered SBDs are an extension of SBDs where the firing of a block may be controlled by a Boolean signal called a *trigger*. At a given synchronous step, if the trigger is *true*, the block fires normally; otherwise, the block *stutters*, that is, keeps its local state and local outputs unchanged, until the next step. Triggered SBDs are useful for modeling *multi-rate* systems, where different parts of the system operate at different time scales. Notice that the triggering patterns need not be periodic. A trigger signal for a block A may be produced by another block B, or it may even be an external input of the diagram. The point is that the behavior of the triggers (i.e., at which steps they are *true* or *false*) is generally unknown. An exception is *Timed* SBDs, a special case of Triggered SBDs, where triggering patterns are known statically ("at compile time").

We follow the problem formulation of [69] where "distributed asynchronous execution platforms" are captured by so-called *finite FIFO platforms* (FFPs). An FFP is similar to a Kahn Process Network (KPN) [40], with the difference that while in a KPN queues are unbounded, in an FFP they are of fixed, finite size. Although FFPs model a specific kind of distributed systems and in particular network communication, they can themselves be mapped in a semantics-preserving way to a variety of underlying networks, such as onto the *loosely time triggered architecture* (LTTA) [69]. Therefore, FFPs represent a useful intermediate layer that can serve as a first step in distributing a model onto many different execution platforms (all platforms upon which FIFO queues can be implemented, e.g. using the TCP protocol). This can be done because FFPs make *no assumption about the relative speeds of the local clocks of distributed processes*, hence the characterization *asynchronous.*

There is a simple way to solve the distribution problem for Triggered SBDs: first, apply trigger elimination to translate the Triggered SBD into a pure SBD; then, use the mechanisms in [69] for distribution of pure SBDs. Unfortunately, this simple method often results in unnecessary communication overhead: a block always sends output messages even when its trigger is *false*. The block does not fire in this case, so the outputs have the same values as in the previous step, but they are still transmitted to downstream blocks. We present an implementation method that eliminates this overhead. This is especially critical in CPS where communication is expensive, for example, in wireless applications where the channel capacity is limited, or where energy savings are essential.

In particular, our implementation method optimizes communication along the following two directions: first, data messages are not sent to processes that are not triggered; second, a process which is not triggered need not send a full data message to its successor processes, but only a flag indicating that the data are the same as in the previous step. In addition to these optimizations that apply to general Triggered SBDs, we also present further optimizations for the case of Timed SBDs.

4.2.2.1 Triggered SBDs and Timed SBDs

A Triggered SBD consists of a set of *blocks* connected to form a *diagram.* Each block has a number of input ports (possibly zero) and a number of output ports (possibly zero). Diagrams are formed by adding connections. There are two types of connections: a *data connection* connects some output port of a block M to some input port of another block M'; a *trigger connection* connects some output port of a block M directly to another block M' (in this case we say that *M' has a trigger*). A block can have zero or one incoming trigger. An output port can be connected to more than one input ports. However an input port can only be connected to a single output.

Semantically, each block corresponds to a state machine, generally of type Mealy [42]. We say that a block is "Moore" if its output function only depends on its state, but not on the inputs. Every connection in the diagram corresponds semantically to a *stream*, that is, a function $s : \mathbb{N} \to \mathcal{U}$, where $\mathbb{N} = \{0, 1, 2, ...\}$ is the set of natural numbers, $\mathcal{U}$ is the universe of all possible data values that streams in the diagram can take, and $s(n)$ represents the value of s at the n-th synchronous step. For simplicity, we ignore typing issues, which in practice would only allow connections between ports of compatible types. However, we use terms such as "Boolean signal" for streams that only take values in a restricted subset of $\mathcal{U}$, e.g., $\{\mathit{true}, \mathit{false}\}$ for Boolean signals.

The semantics of a diagram can be given as a *composite state machine*, obtained by synchronous composition of all machines corresponding to blocks in the diagram. To define the composite state machine, we assume that the diagram is *acyclic*, that is, every dependency cycle visits at least one Moore block. We also assume that there are no "self-loops": this is not a restrictive assumption since blocks can have internal state. The state space of the composite machine is the product of the state spaces of all its component machines, plus all outputs of blocks that have triggers. These outputs become states because when a block is not triggered, its outputs maintain their previous value. The outputs of the composite machine can be defined to be all outputs in the system (including those connected to inputs).

The state of the composite machine is updated by updating the states of all individual components. The output function of the composite machine is defined by defining the value $s(n)$ of every stream s in the diagram, for a given $n \in \mathbb{N}$. Suppose s is the output of machine M. If M has no trigger, $s(n)$ is defined by the output function of M. This requires the local inputs of M to be already known. Since the diagram is acyclic, there always exists a well-defined order in which to evaluate all streams in the diagram at every step n. If M has trigger t and $t(n) = \mathit{true}$, again $s(n)$ is defined by the output function of M. If M has trigger t and $t(n) = \mathit{false}$, $s(n) = s(n-1)$ (if this happens when $n = 0$, some default value is used for $s(0)$). Notice that M having no trigger is equivalent to M having a trigger which is *true* at every step.

A *Timed SBD* is a special case of a Triggered SBD where every trigger is generated by a (period, initial phase) pair (PPP) $(\tau, \theta) \in \mathbb{N} \times \mathbb{N}$, where τ represents a period and θ represents an initial phase.[1] For example, the pair $(2, 1)$ generates the stream *false true false true* $\cdots$. Clearly, every PPP can be defined by a finite state machine, so Timed SBDs are a subclass of Triggered SBDs. The important thing about Timed SBDs is that the triggering pattern is known "at compile time". This is not the case for general Triggered SBDs. Note that the implementation methods that we present here, as well as those proposed in [69], are agnostic of the internals of blocks, that is, blocks are treated as black boxes whose internal state machines are not known.

[1] More generally, triggers in timed SBDs could be specified by *firing time automata* (FTA) [45]. Our implementation method can be directly extended to FTA, but for simplicity, we limit our discussion to PPPs.

4.2.2.2 The Distribution Problem

The distribution problem is to automatically generate from a given Triggered SBD, an FFP that is *stream-equivalent* to the Triggered SBD. Generating an FFP means synthesizing the topology of the FFP (processes and FIFO queues) and the code that each FFP process executes. The topology synthesis is straightforward, since we assume a one-to-one mapping of blocks to processes, as in [69]. In an FFP, a stream is essentially the sequence of values that are written in a given queue. In the naive implementation, stream equivalence requires that every stream s^* produced in the FFP be identical to the corresponding stream s defined by the Triggered SBD. This requirement is too strict for the optimized implementation, where redundant messages are omitted from s^*. Instead, we require only that s^* be identical to s sampled at the points in time when the consumer of s is triggered.

Note that, contrary to Triggered SBDs, streams of FFPs are not guaranteed to be infinite. This is because some processes in an FFP may "deadlock", waiting forever for messages in an input queue or space in an output queue. A proof of semantical preservation must therefore show that the resulting FFPs are deadlock-free [69].

4.2.2.3 Distribution of General Triggered SBDs

We first introduce some notations and terminology. Figure 4.2 shows the general configuration of a block M and its *surroundings*, as a part of a Triggered SBD.

- If M has a trigger t, $T(M)$ denotes the block that produces t. If M has no trigger, $T(M)$ is undefined: we examine this as a special case below.
- The set of blocks that have data connections to M is denoted as $W(M)$. [2]
- $B(M)$ denotes the set of blocks triggered by M.
- $R(M)$ denotes the set of blocks that have data connections from M, except for those blocks that are already in $B(M)$. $R(M)$ is partitioned into two disjoint subsets: $RR(M)$, containing all blocks in $R(M)$ that either have no trigger or have a trigger but are already in $W(B(M))$; and $RT(M)$, containing all the remaining blocks of $R(M)$.

Note that $W(M), R(M), B(M)$ are pairwise disjoint. Also, absence of self-loops ensures that M cannot be a member of any of these three sets. Finally, $T(M)$ cannot be an element of either $R(M)$ or $B(M)$ (this would result in cyclic diagrams), but it may be an element of $W(M)$.

[2]For a set W, $|W|$ denotes its cardinality. We use W_1, W_2, etc. to enumerate and denote its elements, so that $W(M) = \{W(M)_1, ..., W(M)_{|W(M)|}\}$. Also, we define $W(X) = \bigcup_{Q \in X} W(Q)$, for a set of processes X.

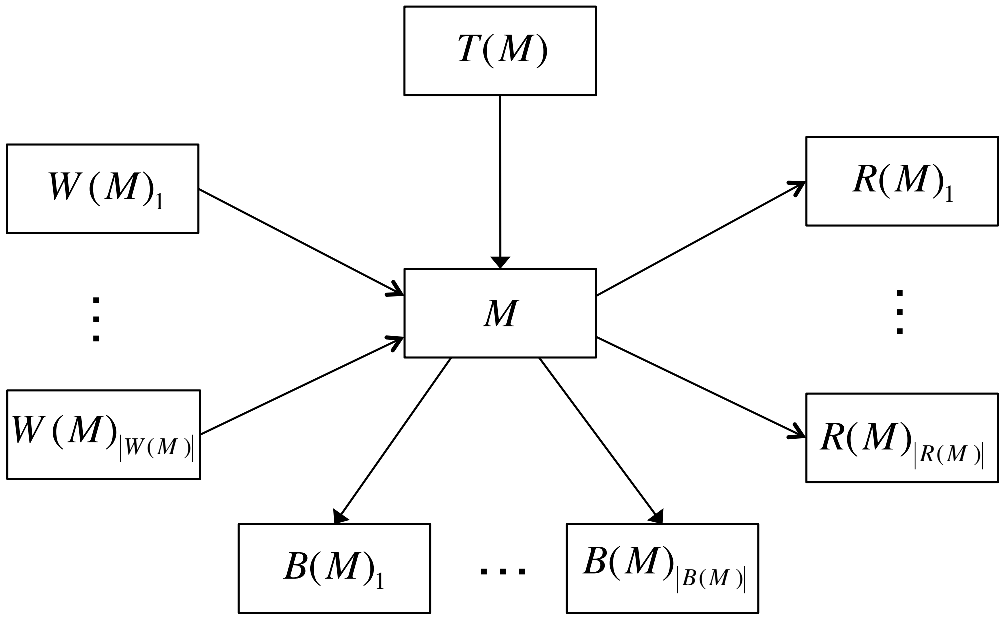


Figure 4.2: A block M and its surroundings.

Mapping Triggered SBDs on FFPs

A Triggered SBD is mapped onto an FFP in the following way. Every block M in the Triggered SBD is mapped to an FFP process P. Every link from a block M to another block M' in the Triggered SBD is mapped to a FIFO queue between the corresponding FFP processes, from P to P'. The sizes of the queues are as in [69]. In particular, if M is not Moore, a queue of size 1 is sufficient; otherwise a queue of size 2 is sufficient (this queue is initialized with a message carrying the initial output of M). Schematically, the Triggered SBD part shown in Figure 4.2 results in the FFP part shown in Figure 4.3.

Similarly to the notation $T(M)$,$W(M)$, etc. for blocks, we introduce notation $T(P)$,$W(P)$, etc. for processes. If block M is mapped to process P, $T(P)$ denotes the process corresponding to $T(M)$, $W(P)$ denotes the set of all processes P' such that P' corresponds to some block $M' \in W(M)$, etc.

As can be seen from Figure 4.3, P may have more inputs and outputs (shown in blue) than its corresponding block M. In particular, P receives additional input signals from processes in $T(RT(P))$. This is done in order to minimize data traffic: if a process $P' \in RT(P)$ is not triggered in a given step, P need not send a message to P' for that step. To know whether P' is triggered or not, P needs to receive a message from the process triggering P', that is, from $T(P')$. These additional signals are called *backward* signals and the corresponding queues are called backward queues. They are illustrated in Figure 4.4. Backward signals are sent to backward queues at every step.

Symmetrically, P itself may trigger other processes (those in $B(P)$). Therefore, P needs to notify potential writers of processes in $B(P)$ about whether the latter are triggered or

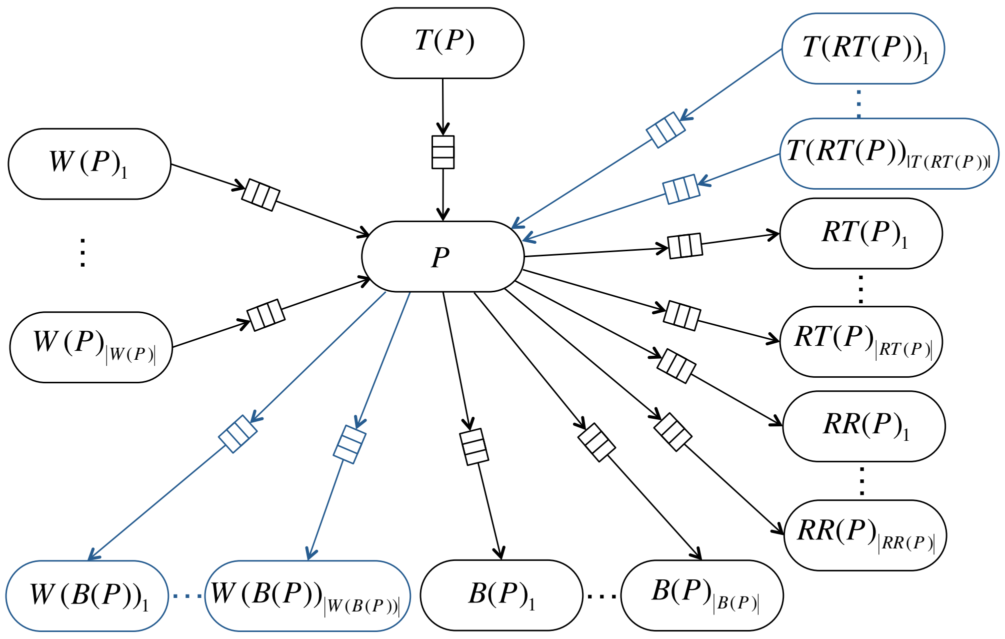


Figure 4.3: Part of an FFP generated from M and its surroundings in Figure 4.2.

not. This explains the additional output queues of P, namely, queues to the processes in $W(B(P))$.

We should note that additional queues are introduced by the optimized implementation, only if they do not exist after mapping links in the Triggered SBD to FIFO queues in the FFP. For example, the process $T(P)$ may also be in $T(RT(P))$. This is the case in Figure 4.8, where $T(P_{11}) = T(P_{12}) = P_0$. Since there is already a queue from P_0 to P_{11}, no additional queue is needed.

Additional backward queues may create apparent dependency cycles in the FFP, as illustrated in Figure 4.5. If M_1 already has a forward link to M_3, adding a backward queue from P_3 to P_1 in the FFP creates a cycle. To ensure that such cycles are not problematic, i.e. do not result in deadlocks, a process P is designed in a way such that its execution is structured in *stages*. The stages are ordered so that dependency cycles are not introduced. In the example of Figure 4.5, P_1 will transmit to P_3 without waiting for messages from the backward queue. These messages are necessary only in order for P_1 to decide whether to send a message to P_2 or not, but are not needed for P_1 to compute its outputs.

The code that each FFP process P executes is shown below. It follows the same general scheme as the implementation described in [69]: initialization of state variables, followed by execution of an infinite loop. Every iteration of the loop proceeds in a number of stages. First (Stage 0), P determines if it is triggered in the current iteration. If $T(P)$ is undefined, P is implicitly always triggered, therefore `trigger` is set to *true*. Otherwise, P needs to consume a message from the input queue `trigger` coming from process $T(P)$ and containing the value of the trigger. If the queue is empty, P needs to wait until a message arrives. At Stage 1, P

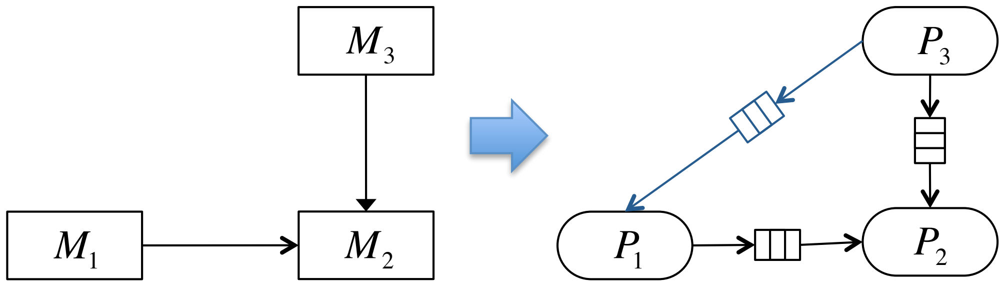


Figure 4.4: Backward queue sending trigger information about P_2 to P_1.

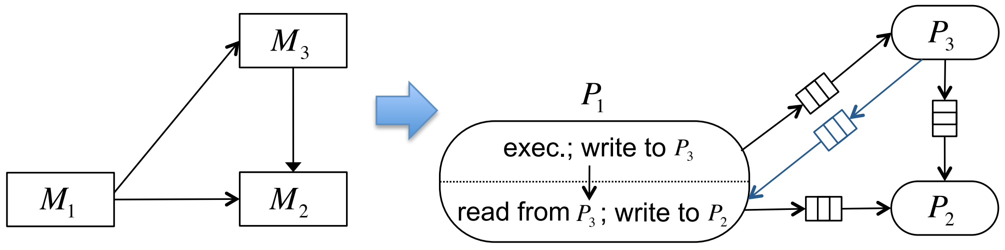


Figure 4.5: Avoiding deadlocks by structuring each process in stages.

fires if and only if the trigger is *true*, and sends messages to $RR(P) \cup B(P) \cup W(B(P))$ (the union of the sets is denoted as $RB(P)$ in the code). These messages are sent at every step, even when P is not triggered. At Stage 2, P sends messages to the processes in $RT(P)$ that are triggered: the rest need not receive data messages. This is part of the traffic optimizations that our method achieves.

FFP process for a general Triggered SBD

```
P (inputs: ins, trigger; outputs: outs)
{
  initialize state, outs, ins' and outs';
  for all i, outs'[i].fresh := true;
  while (true) {
    for all i in T(RT(P)), known[i] := false;
    // Stage 0: determine trigger
    if  (T(P) is defined) {
      wait until trigger queue is not empty;
      get_inputs(trigger);
      if(trigger.fresh = true)
        ins'[T(P)] := trigger;
      if (T(P) in T(RT(P))) known[T(P)] := true;
    }
    else
      trigger := true;
    // Stage 1: fire and send to RB(P), where
    // RB(P) := RR(P) union B(P) union W(B(P))
    wait until no queue to RB(P) is full;
    if (trigger) {
      wait until no queue from W(P) is empty;
      get_inputs(ins[W(P)\T(P)]);
      for (every i in W(P)\T(P) s.t.
           ins[i].fresh = true)
        ins'[i] := ins[i];
      for (every i in W(P)\T(P) s.t. i in T(RT(P)))
        known[i] := true;
      (state, outs) := M.step(state, ins'[W(P)]);
      for all i in (RB(P)) {
        outs'[i].fresh := true;
        outs'[i].data := outs[i];
      }
    }
    put_outputs(outs'[RB(P)]);
    for all i in (RB(P))
      outs'[i].fresh := false;
    // Stage 2: selectively send to RT(P)
    RTunproc := RT(P);
    while (RTunproc != empty) {
      pick a process rt in RTunproc;
      if (T(rt) = P) {
        known[T(rt)] := true;
        ins'[T(rt)] := outs'[rt];
      }
      if (known[T(rt)] = false and the
          queue from T(rt) is not empty)  {
        get_inputs(ins[T(rt)]);
        if (ins[T(rt)].fresh = true)
          ins'[T(rt)] := ins[T(rt)];
        known[T(rt)] := true;
      }
      if (known[T(rt)] = true)
        if (ins'[T(rt)] = false)
          remove rt from RTunproc;
        else if (the queue to rt is not full) {
          put_outputs(outs'[rt]);
          outs'[rt].fresh = false;
          remove rt from RTunproc;
        }
    }
  }
}
```

Returning to the example in Figure 4.5, P_1 will send a message to P_3 at Stage 1, since $P_3 \in RR(P_1)$. Then, P_3 can execute and send back to P_1 the trigger information about P_2 via the backward queue. Once P_1 has this information, it can decide whether a message needs to be sent to P_2. If so, this will happen at Stage 2 of P_1. One can see how this careful ordering avoids dependency cycles and deadlocks in this example. More complicated cases exist, for instance, where P_3 has a trigger and belongs not to $RR(P_1)$ but to $RT(P_1)$. The proof of semantical preservation described in Section 4.2.2.3 argues how these cases are also handled correctly by our method.

We now further explain the code of P. `trigger` denotes the trigger input queue of P, `ins` denotes the set of all the other input queues, and `outs` denotes the set of all output queues. We use notation such as `ins[i]` to denote the queue from a given process `i`. Similarly, if X is a set of processes, `ins[`X`]` denotes the set of the corresponding queues.

P maintains state variables `ins'` and `outs'`. For each input queue from a process `i`, `ins'[i]` memorizes the last data message received from the queue. This is used when a process has no "fresh" message for P (i.e. no new message since the last time P was triggered), in which case it only sends a flag to P indicating that the last data message should be used. Symmetrically, for each output queue, `outs'` memorizes the latest message that P produced for that queue. Note that `get_inputs()` and `put_outputs()` use only `ins` and `outs` and do not affect `ins'` and `outs'`.

Messages in `outs'` contain an extra Boolean flag `fresh`, indicating that the corresponding output is newly produced, as opposed to the one that has already been sent. Initially all output data are fresh: this is because the initial data may need to be sent before the first time P is triggered. When `put_outputs()` takes `outs'` as argument, it first checks the `fresh` flag of each message: if it is *true*, the whole message is sent; otherwise, only the flag is sent, indicating that the data is the same as in the last transmitted message. This reduces communication load, since data messages typically have a larger payload. Note that each message sent by `put_outputs()` contains all the information that must be transmitted from one process to another within a synchronous step. Such a message may therefore include, for example, both a trigger and a data part.

For each process `i` in $T(RT(P))$, P also maintains a Boolean flag `known[i]`. These flags are used to indicate whether the value of certain triggers is known at a given iteration. All flags are reset to *false* at the beginning of each iteration. Once messages are received, the corresponding flags are set to *true*.

`RTunproc` represents the set of all processes in $RT(P)$ that P needs to consider in Stage 2. For each $rt \in RT(P)$, P needs to determine if rt is triggered: if so, P sends a message to rt, otherwise not. P iterates over all processes in $RT(P)$ until all of them are handled. A process `rt` is selected randomly, and P checks whether the triggering status for `rt` is known. If not, P attempts to find out by checking whether the backward queue from `T(rt)` contains a message. If the trigger value for `rt` is known and it is *false*, P need not send a message to it. If the trigger is *true*, P sends a message if space is available in the corresponding queue. In those cases, `rt` is removed from `RTunproc`, marking the fact that `rt` has been handled.

Stage 2 may appear unnecessarily complicated: why not simply iterate over all processes $rt \in RT(P)$, wait for a message from $T(rt)$, proceed to decide whether rt is triggered or not, and send a message to rt if it is triggered? The reason is that a fixed order of iterating over processes in $RT(P)$ may result in deadlocks. For example, assuming $P_1, P_2 \in RT(P)$ and we decide to wait first for a message from $T(P_1)$ and then a message from $T(P_2)$, there will be a deadlock if P_1 is itself triggered by P_2, i.e. $T(P_1) = P_2$. This situation is illustrated in Figure 4.6 (the Triggered SBD is shown to the left and the corresponding FFP to the right). Links from P_3 to P and from P_2 to P are backward links. The deadlock happens because: P_2 waits at Stage 1 for a message from P; while at the same time, given the above fixed iteration order, P at Stage 2 first waits for a message from $T(P_1)$, i.e. from P_2. This happens before P can wait for a message from $T(P_2)$ to decide whether to send a message to P_2.

This deadlock is avoided in our method. Assuming P_1 is selected first in Stage 2 of P, P attempts to read the trigger signal from $T(P_1) = P_2$, but finds the backward queue from P_2 to P empty, so another process in $RT(P)$ is selected. In this way, no extra dependencies are added among processes, and P eventually handles P_2 before P_1.

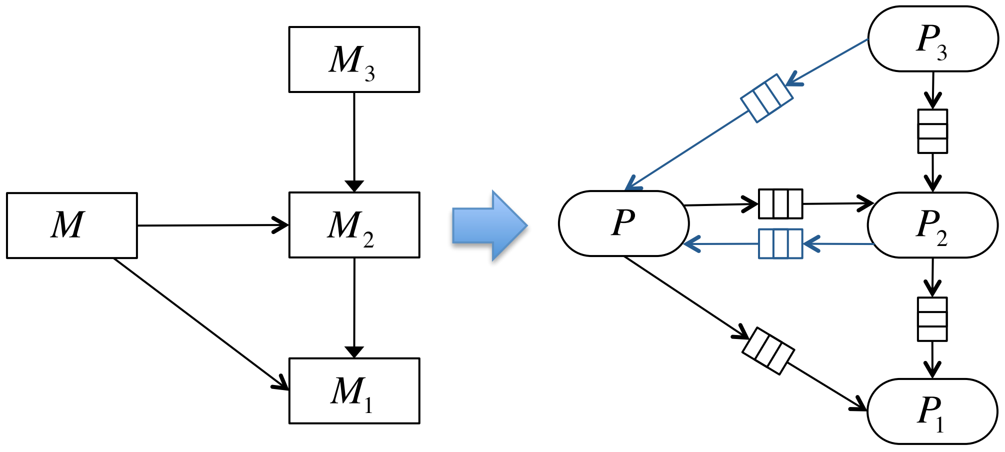


Figure 4.6: Potential deadlock with a static iteration order over $RT(P)$.

Note that the deadlock *could* be avoided with a static iteration order, where P handles P_2 before P_1. However, such a static order generally depends on the topology of the diagram. In this work, we opted for a method that guarantees absence of deadlocks while being *modular*, that is, where the code for P does not depend at all on the diagram.

Semantical Preservation

Stream equivalence between a Triggered SBD G and the FFP generated by our method, denoted as F^*, can be proven in four steps. Due to space limitation, we present only a sketch of these steps here. The full proof is shown in our technical report [73].

*S*tep 1 and 2: G is transformed to an equivalent pure SBD G_s using the trigger elimination method from [45]. G_s is then mapped to an FFP F_s, using the method proposed in [69], which guarantees stream equivalence between F_s and G_s, and therefore also between F_s and G.

*S*tep 3: We transform F_s to a new FFP, denoted as F_s', by adding backward signals (and queues if needed), and restructuring every process in F_s into three stages, as with processes in F^*. The difference between F_s' and F^* is the following: although a process in F_s' reads the backward signals, it does not use the information; instead, it always sends messages to all the output queues at every step.

We next show that F_s' is stream-equivalent to F_s. For this, it suffices to prove that no process in F_s' ever deadlocks. This is because every process P in F_s' behaves identically to the corresponding process in F_s, except that P consumes a set of additional messages that it never uses. To prove that no process in F_s' deadlocks, we use the careful structuring of the code into stages, which ensures that the additional backward queues do not create any dependency cycles.

*S*tep 4: We prove that F^* is stream-equivalent to F_s'. These two FFPs have the same structure, i.e., there is a one-to-one mapping between the processes and the FIFO queues in the two FFPs. Consider a pair of corresponding processes in the two FFPs, P^* in F^* and P in F_s'. Because the structure of the two FFPs is the same, P^* has a trigger if and only if P has a trigger. Also, trigger messages are transmitted at every step and never omitted. Based on this, we use induction to show that the trigger signals of P^* and P have the same value at every step, and that the input data that are read by P^* and P are the same at every step when the trigger signals are true. The following facts are used in this proof: (1) state variables `ins'` and `outs'` of P^* memorize the latest inputs and outputs; (2) for every process $W^* \in W(P^*)$, W^* sends a message to P^* at a given step if and only if P^* reads the message from W^* at the same step. From (1) and (2) we can derive that P^* always gets the up-to-date inputs, either from a message from W^*, or from its state variable `ins'` when W^* sends a message with fresh-bit being false.

Communication Savings Analysis

Compared to the naive method, the communication savings achieved by the optimized method are, on the average (in bits per step), at least as follows.

$$\sum_{l:(W,R)\in L} P_W L_D + B^{RT}_{W,R} P^*_{W,R} L_D - \left(B^{RT}_{W,R}\left(1 - P_R\right) + \left(1 - B^{RT}_{W,R}\right) + B^{RT}_{W,R}\right) L_T \tag{4.7}$$

where L is the set of links in the Triggered SBD; W and R are the writer and reader blocks of a link l; P_W and P_R are the probabilities of W and R *not* being triggered at any given step; $B^{RT}_{W,R}$ is a Boolean variable indicating whether R is in $RT(W)$ or not; and L_T and L_D

are the lengths of trigger and data messages, respectively. The size of a control message is approximately the same a trigger message.

The first term of savings, $P_W L_D$, comes from the fact that, in the FFP, W only sends to R the new data which is produced when W is triggered. The second term is due to the fact that if $R \in RT(W)$, W only sends a message to R when R is triggered. Specifically, let $P^*_{W,R}(k)$ be the probability of savings due to the non-triggering of the reader R at step k. The savings are realized when the following two conditions are met: (1) R is not triggered while W is triggered at step k (the case where W is not triggered is already included in the first term of savings); (2) W is triggered at least once no later than the next time R is triggered. In this case, the output of W produced at step k need not be sent to R. $P^*_{W,R}(k)$ can be calculated as

$$P^*_{W,R}(k) = P_R\left(1 - P_W\right) \cdot \left(P_R^{N-k} + \sum_{i=0}^{N-k-1} \left(1 - P_R\right) P_R^i \left(1 - P_W^{i+1}\right)\right) \tag{4.8}$$

where N is the number of total steps a system runs. As N goes to infinity, $P^*_{W,R}$ becomes independent of k and is equal to

$$P^*_{W,R} = \frac{P_R\left(1 - P_W\right)^2}{1 - P_R P_W} \tag{4.9}$$

Returning to Equation (4.7), L_T bits must be deducted (in the worst case) from the savings with probability $(1 - P_R)$ if $R \in RT(W)$, due to the fact that an additional fresh-bit is sent from W to R at any step when R is triggered; and with probability 1 if $R \notin RT(W)$, since the fresh-bit needs to be sent at every step in this case. Finally, if $R \in RT(W)$, there is an additional backward signal sent from $T(R)$ to W at every step. Note that in most systems, L_D is much larger than L_T, therefore our savings could be significant. Moreover, some of the messages are often merged in the optimized method (e.g. a fresh-bit whose value is *true* is always merged with the data), which will produce even more savings than what was represented in Equation (4.7).

4.2.2.4 Distribution of Timed SBDs

Since Timed SBDs is a special case of Triggered SBDs, we could simply use the method described in Section 4.2.2.3. However, we can do better than that if we exploit the information about triggering patterns which is statically known in Timed SBDs. In particular, let P be the FFP process corresponding to a block M with (period,initial phase) pair (τ_M, θ_M). Let $\tau_P = \tau_M$ and $\theta_P = \theta_M$.

Let R be a process receiving data from P. To save communication load, P only need to send a message to R when P is triggered and the message will be read by R, i.e. R will be triggered at least once before the next time P is triggered. More precisely, at its k-th triggered instant, P needs to send a message to R if and only if R is triggered at least once within the interval between the k-th and $(k+1)$-th triggered instants of P. This is represented by the predicate $\mathsf{put?}(P, R, k)$, defined as follows:

$$\begin{aligned}&\mathsf{put?}(P, R, k)\\ &\mathrel{\widehat{=}} \exists j : k\tau_P + \theta_P \le j\tau_R + \theta_R < (k+1)\,\tau_P + \theta_P \\ &\equiv \left\lceil \frac{k\tau_P + \theta_P - \theta_R}{\tau_R} \right\rceil \tau_R + \theta_R < (k+1)\,\tau_P + \theta_P\end{aligned} \tag{4.10}$$

Similarly, let W be a process sending data to P. At its k-th triggered instant, P must expect a new message from W if and only if W has been triggered between the $(k-1)$-th and k-th triggered instants of P. This is represented by the predicate $\mathsf{get?}(W, P, k)$, defined as follows:

$$\begin{aligned}&\mathsf{get?}(W, P, k)\\ &\mathrel{\widehat{=}} \exists j : (k-1)\tau_P + \theta_P < j\tau_W + \theta_W \le k\tau_P + \theta_P \\ &\equiv \left\lfloor \frac{k\tau_P + \theta_P - \theta_W}{\tau_W} \right\rfloor \tau_W + \theta_W > (k-1)\,\tau_P + \theta_P\end{aligned} \tag{4.11}$$

The above predicates are used in the code of a process P generated from a Timed SBD, as shown below.

FFP process for a Timed SBD

```
P (inputs: ins; outputs: outs)
{
  initialize state, outs, and ins';
  k := 0;
  for (every R in R(P))
    if (theta_R < theta_P)
      put_outputs(outs[R]);
  while (true) {
    Wset, Rset := empty sets;
    for (every W in W(P))
      if (get?(W, P, k))
        add W to Wset;
    for (every R in R(P))
      if (put?(P, R, k))
        add R to Rset;
    wait until no queue from Wset is empty
      and no queue to Rset is full;
    get_inputs(ins[Wset]);
    for all i in Wset, ins'[i] := ins[i];
    (state, outs) := M.step(state, ins');
    put_outputs(outs[Rset]);
    k := k + 1;
  }
}
```

At every iteration, P computes the sets Wset (Rset) of processes that P needs to receive from (send to). Then P waits for messages (slots) to become available on the corresponding queues before it fires. To compute Wset and Rset, P maintains a local counter k: notice that k does not count synchronous steps, but rather the times that P has fired. P has period τ_P and therefore fires every τ_P steps. P also maintains a state variable ins' which, similar to the code of FFP process for general Triggered SBDs, memorizes the last messages received at the inputs.

The distribution method guarantees stream equivalence while mapping a Timed SBD to an FFP. The proof is shown in our technical report [73].

Communication Savings Analysis

In our method for Timed SBDs, the communication load for a link l is $max\{\tau_W, \tau_R\}^{-1} \cdot L_D$. Therefore, compared to the naive method, the savings achieved by our method are

$$\sum_{l:(W,R)\in L} \left(1 - \frac{1}{max\{\tau_W, \tau_R\}}\right) L_D \tag{4.12}$$

4.2.2.5 Case Studies

We demonstrate the communication savings achieved by our proposed method for Triggered SBDs in the following case studies. We first analyzed the savings for multi-mode systems, a special case of Triggered SBDs. Furthermore, we conducted experiments with randomly generated Triggered SBDs to show the effectiveness of our method in general case.

Multi-Mode Systems

Figure 4.7 shows a Triggered SBD. This diagram models a two-mode system, consisting of two separate sets of communicating blocks, plus a mode control block that triggers only one of the sets at any given time. The output of the control block is a Boolean signal: when it is *true*, the blocks of Mode 1 are triggered, and when it is *false*, the blocks of Mode 2 are triggered. Notation-wise, we use different types of arrow heads to distinguish trigger signals from standard inter-block communication signals, and we usually draw triggering signals as incoming to the top of a block.

First, we apply the "naive" method to distribute the model. After applying the trigger elimination method of [45], followed by the distribution method of [69], we get the FFP diagram shown in Figure 4.8. Each block M_0, M_{11}, M_{12}, etc. of the original diagram gives rise to a process P_0, P_{11}, P_{12}, etc. in the FFP. The triggers of the original diagram have now

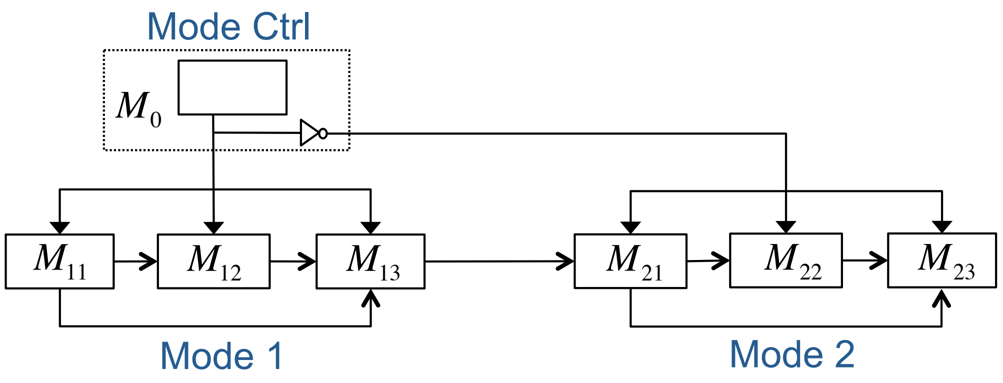


Figure 4.7: A Triggered SBD.

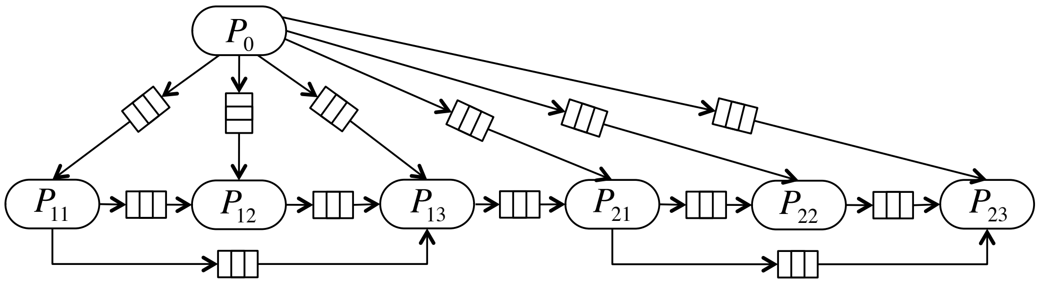


Figure 4.8: The FFP system resulting from the Triggered SBD of Figure 4.7 after trigger elimination [45] and distribution [69].

become standard inputs to the FFP processes. Each FFP process P executes the following pseudo-code:

```
P(inputs: ins, trigger; outputs: outs)
{
  initialize state and outs;
  while (true) {
    wait until all input/output queues
       are non-empty/non-full;
    get_inputs(ins, trigger);
    if (trigger) then
      (state, outs) := M.step(state, ins);
    put_outputs(outs);
  }
}
```

where `state` denotes the internal state of M that P inherits. In addition, output variables `outs` are also state variables in P.

Process P behaves as follows. It starts by initializing its state variables (including `outs` – the reason for this will become clear below). It then enters an infinite loop. At each iteration, P waits until all its input queues are non-empty (i.e., contain at least one message) and all its output queues are non-full (i.e., have room for at least one message). Then, P "fires", that is, it performs a synchronous step: one input message is read from each input queue (including the trigger) and one output message is written to each output queue, using the functions `get_inputs()` and `put_outputs()` (we assume that these functions are "smart enough" to know which variable corresponds to which queue). When the trigger is *true*, P uses the output function of M, `M.step()`, to update the outputs and the state. When the trigger is *false*, no updates are made and the values written at the outputs are the same as in the previous step (i.e., the process "stutters").

All processes in the FFP of Figure 4.8 execute concurrently, following the above pattern P. Although the processes are not synchronized, some loose form of synchronization is still imposed because of the queues: a process cannot fire when it is waiting for an input from another process, or for a downstream process to free up space in an output queue. This distributed concurrent system completes a *logical step* when all messages corresponding to the same synchronous step in the original SBD have been processed.

We use this notion to estimate the communication load in this FFP implementation. We can see that 6 trigger messages plus 7 data messages are transmitted at every logical step. The 6 trigger messages correspond to the messages sent from the control process P_0 to each of the other processes (the negation block is not implemented as a separate process, but as part of the control process). The data messages are sent by the processes among themselves: two messages from P_{11} to P_{12} and P_{13}, one from P_{12} to P_{13}, one from P_{13} to P_{21}, and so on. Let L_T and L_D denote the message lengths for trigger and data messages, respectively. Then, the communication load of the naive implementation is $6L_T + 7L_D$, measured in bits per logical step.

In the optimized implementation method that we present in Section 4.2.2.3, a producer process only sends a message to a consumer process when the consumer is triggered. In our running example, P_{11} only sends messages to P_{12} and P_{13} when the latter are triggered. In this example all processes in the set $\{P_{11}, P_{12}, P_{13}\}$ are triggered simultaneously, and similarly for $\{P_{21}, P_{22}, P_{23}\}$. Moreover, only one of the two sets is triggered at any given logical step. Therefore, in the optimized implementation, at most 4 data messages are transmitted in each logical step: 3 messages among processes of the same mode, plus 1 message from P_{13} to P_{21}. Moreover, the message from P_{13} to P_{21} is only transmitted at the beginning of a mode switch. After that, while the system remains in the same mode, only a control message is transmitted indicating that the data is the same as in the last step. The savings are significant and can be close to $4/7 \approx 57\%$, considering that the data messages are usually much longer than trigger/control messages (whose payload is only a few bits).

The two-mode model can be extended to a more general multi-mode model. Specifically, a k-mode model is a special type of Triggered SBDs. It consists of k sets of communicating blocks denoted as sets M_1 to M_k, and a set of blocks for mode control (denoted as M_0) that triggers only one of the k sets at any given time. Note that there might be communications between different sets of blocks.

Let L_T and L_D denote the message lengths for trigger/control messages and data messages, respectively. When distributing the model by the trigger elimination based distribution method, the average message load per synchronous step is as follows:

$$k \cdot L_T + \sum_{i=0}^{k} C_{M_i} L_D + \sum_{i=1}^{k} \sum_{j=1}^{k} C_{M_i M_j} L_D \tag{4.13}$$

where C_{M_i} denotes the number of messages between the blocks in the set M_i at a synchronous step, and $C_{M_i M_j}$ denotes the number of messages between the blocks in M_i and M_j.

On the other hand, when applying our distribution method, the average message load per synchronous step is as follows:

$$\begin{aligned} (k + k') \cdot L_T + C_{M_0} L_D + \sum_{i=1}^{k} P_{M_i} C_{M_i} L_D \\ + \sum_{i=1}^{k} \sum_{j=1}^{k} P_{M_i} C_{M_j M_i} L_T + \sum_{i=1}^{k} \sum_{j=1}^{k} P_{M_i M_j} C_{M_i M_j} L_D \end{aligned} \tag{4.14}$$

where P_{M_i} denotes the probability of the set M_i being triggered, and $P_{M_i M_j}$ denotes the probability that M_j is triggered at a synchronous step and M_i is triggered at the step before (i.e. mode switch from M_i to M_j). k' is the number of additional backward trigger messages.

In the case that the data messages are much longer than trigger/control messages, and that the mode switch happens sporadically, the communication saving ratio accomplished by our method can be approximated to

$$\begin{aligned} & \frac{\sum_{i=1}^{k} C_{M_i} - \sum_{i=1}^{k} P_{M_i} C_{M_i}}{\sum_{i=0}^{k} C_{M_i}} \\ \geq & \frac{\sum_{i=1}^{k} C_{M_i} - max_{i=1}^{k}\{C_{M_i}\}}{\sum_{i=0}^{k} C_{M_i}} \end{aligned} \tag{4.15}$$

In the case that the number of messages in each set is close to each other, the communication saving ratio approximates to

$$\frac{k-1}{k+1} \tag{4.16}$$

Intuitively, the savings are due to the fact that only the data messages in set M_0 and the set that is triggered by M_0 are transmitted at a synchronous step in our approach.

Randomly Generated Triggered SBDs

Furthermore, to show the effectiveness of our distribution method for Triggered SBDs in general case, we conducted experiments computing the communication savings on randomly generated Triggered SBDs. We use TGFF[3] to generate random directed acyclic graphs consisted of blocks and links, and randomly pick some of the links as trigger links. We then assign a probability of *not* being triggered to every block that has a trigger. Specifically, the probability is assigned as a uniformly distributed random number in the range $[0, 2x]$, where x is the expectation of the random number and should be no larger than 0.5. The communication savings of the randomly generated Triggered SBDs can be calculated using Equation (4.7).

We generated Triggered SBDs with the number of blocks ranging from 100 to 1000. The experiment result in Figure 4.9 shows the average communication savings for those Triggered SBDs when the average probability of blocks not being triggered goes from 0.1 to 0.5. We picked four communication protocols, CAN bus, ZigBee, Wi-Fi and TTP/C, which have different message overheads and maximum data payloads (and therefore result in different L_D and L_T). The communication saving ratios achieved by our distribution method increase approximately in a linear relation with the probability of blocks not being triggered, and the saving ratios for the protocols Wi-Fi and TTP/C are larger than CAN bus and ZigBee due to the fact that the former two groups can have bigger ratio of L_D to L_T than the latter two.

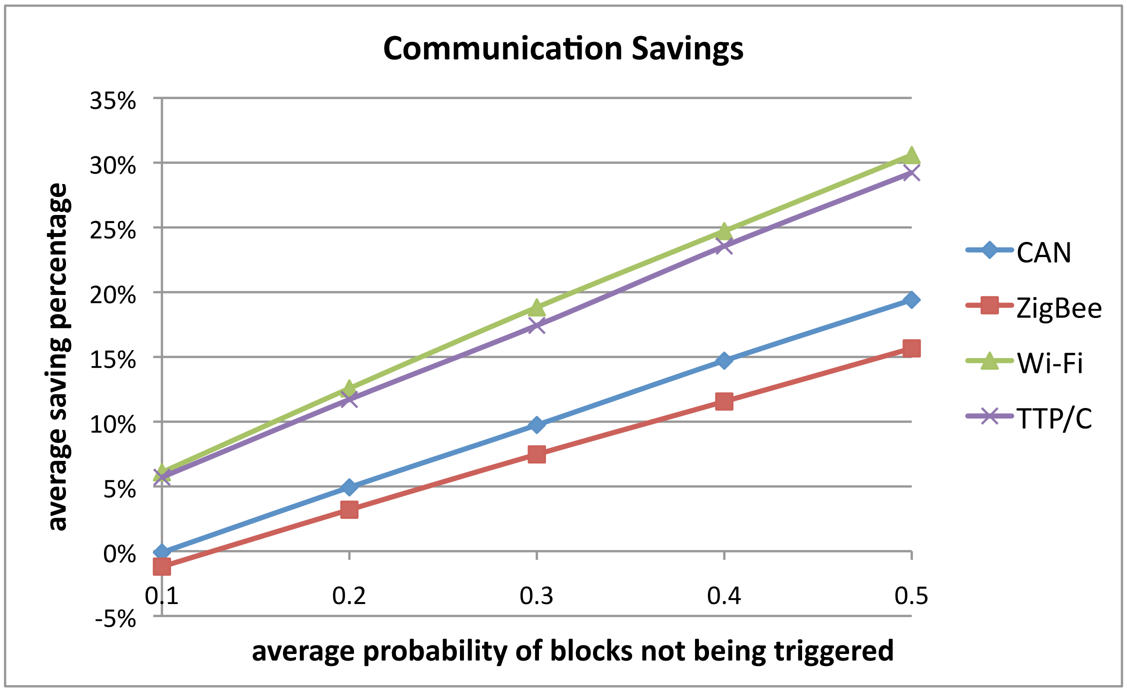


Figure 4.9: Average communication savings with different probability of blocks not being triggered.

We also conducted another experiment to show the impact on communication savings when the numbers of trigger links in Triggered SBDs are different. After a random graph is

[3] http://ziyang.eecs.umich.edu/ dickrp/tgff/

generated by TGFF, we randomly pick some of the non-source blocks to be ones with trigger inputs. Specifically, a non-source block is picked with a probability of p, and if a block is picked, one of its input links is randomly chosen as the trigger link. Therefore, the expected percentage of non-source blocks that have triggers is p. We generated such Triggered SBDs with the number of blocks ranging from 100 to 1000, and p ranging from 0.1 to 1. The probability of not being triggered is assigned to every block that has a trigger as a uniformly distributed random number in the range $[0, 1]$. The experiment result in Figure 4.10 shows the average communication saving ratios increase approximately linearly with the percentage of non-source blocks that have triggers.

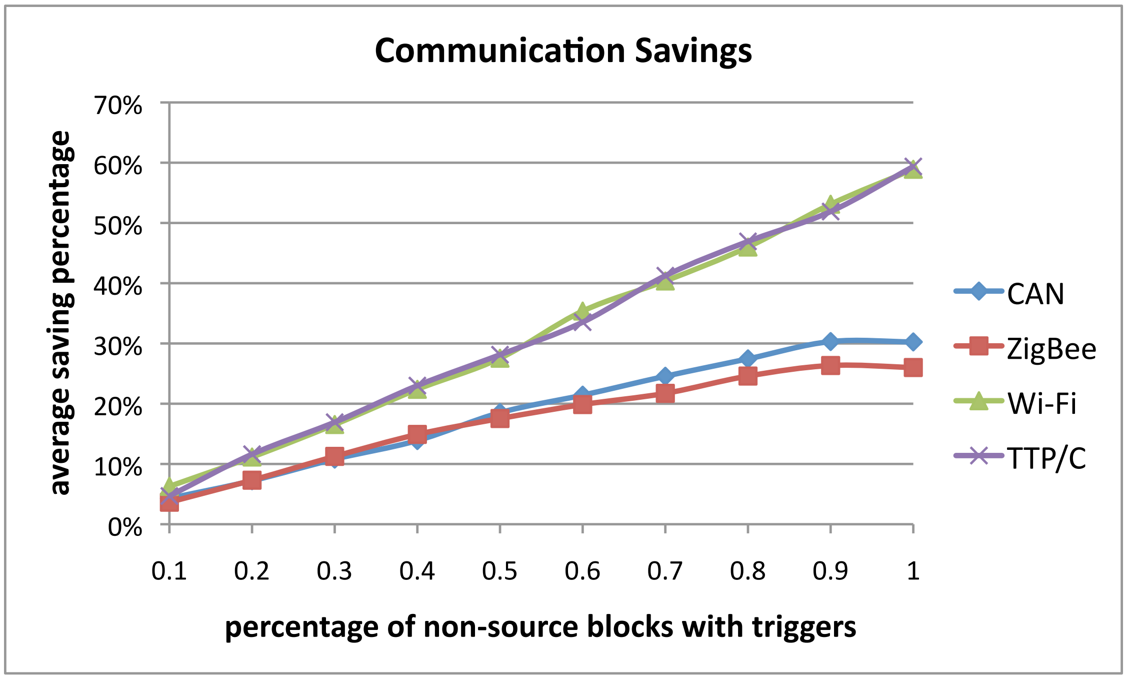


Figure 4.10: Average communication savings with different percentage of non-source blocks with triggers.

Chapter 5

Conclusion and Future Work

5.1 Closing Remarks

We propose a systematic software synthesis flow for distributed embedded systems. It enables integrating heterogeneous input models, conducts automatic design space exploration, and performs code generation while guaranteeing semantic correctness. In closing, we wish to outline the following characteristics of our approach:

Integration

The introduction of IF as a unified representation and an intermediate layer enables the integration of heterogeneous input models, as well as the integration of a modeling front-end and a synthesis back-end. The integration of heterogeneous inputs is particularly important, as with the increasing complexity of the distributed embedded systems, it is common to have different parts of the systems modeled in different languages or semantics, and by different teams. The usage of IF provides a unified environment to tackle the heterogeneity and apply optimization and synthesis tools. This feature is demonstrated in the room temperature control system case study presented in Section 2.2. The input model consists of a control algorithm modeled in Simulink and a plant modeled in Modelica. After translating both of them to a unified representation in IF, simulation can be applied to study the system behavior, as well as the following steps in the flow to optimize and synthesize the system, including mapping, code generation and communication interface synthesis.

Optimality

The mapping process in our synthesis flow focuses on *automatically* exploring the design space to optimize various design metrics such as performance, cost and extensibility. With the scale of the modern embedded systems (especially on the software side) rapidly increasing, such automation is essential to achieve optimal and reliable design. We propose a general mapping flow, as well as customized algorithms to optimally solve the mapping problems in two different application domains. For building automation and control systems, the focus is to minimize the cost of the control system by exploring mapping and architectural exploration together. One interesting aspect is the explicit consideration of the physical aspects such as building layout as well as sensor and actuator locations. For CAN-bus based systems, the focus is to maximize the extensibility of system, a metric that is very important for large-volume and long life-time products. We design an algorithm that combines mathematical programming with heuristics, which we believe offers the best trade-off among optimality, efficiency and flexibility.

Behavior Preservation

Another important aspect in our synthesis flow is to preserve the behavior of the input specification. This is extremely important for distributed embedded systems, many of which are safety critical. In many cases, this is also very challenging, as it is common for such systems to employ an asynchronous distributed architecture for cost concern, while the input specification is synchronous to facilitate verification. To bridge the gap between a synchronous specification and an asynchronous architectural platform, we propose a code generation back-end with communication interface synthesis to guarantee semantic equivalence. We extend the distribution method proposed in [69] to open systems that interact with the physical environment, and also propose a method for Triggered synchronous models, which guarantees behavior preservation while optimizing communication load.

5.2 Future Work

In the future, we expect the problem of software synthesis to have increasing importance in the design of distributed embedded systems. Some of the biggest challenges will include integrating software from multiple sources and verifying the system level properties, co-designing the software with the embedded hardware platform, analyzing the software behavior under the impact of uncertain physical environment, etc. Our proposed software synthesis flow may serve as an initial step for building a solid foundation of design methodologies and tools to address these challenges. Various synthesis, simulation, optimization and verification tools can be built around the IF and integrated to the flow.

One possible extension to the software synthesis flow is to co-design the control algorithms and the architectural platform. In our case study of mapping the room temperature control system, we tried exploring the architectural platform with a given control algorithm. We believe the other direction is also valuable – the control algorithm may be optimized by leveraging the characteristics of the potential architectural platform.

Other extensions may include considering emerging architectures in design space exploration such as multicore processors and wireless platforms, co-designing multiple subsystems such as the HVAC system and the lighting system in building automation and control, exploring other design metrics during the synthesis such as reliability and security.

Bibliography

[1] ANTLR. http://www.antlr.org/.

[2] ARCNET. http://www.arcnet.com.

[3] ASHRAE. http://www.ashrae.org.

[4] Dymola. http://www.3ds.com/products/catia/portfolio/dymola.

[5] EIKON-LogicBuilder for WebCTRL. http://www.automatedlogic.com.

[6] Modelica. http://www.modelica.org.

[7] Simulink. http://www.mathworks.com.

[8] The Hybrid System Interchange Format. http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp.

[9] Metasys GPL Programmer's Mannual, 2004.

[10] User's Guide EIKON for WebCTRL, 2005.

[11] Honeywell Spyder User's Guide, 2007.

[12] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[13] F. Balarin, M. Chiodo, P. Giusto, et al. Synthesis of software programs for embedded control applications. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, 1999.

[14] F. Balarin, M. D'Angelo, A. Davare, et al. Platform-Based Design and Frameworks: Metropolis and Metro II. In Gabriela Nicolescu and Pieter J. Mosterman, editors, *Model-Based Design for Embedded Systems*. CRC Press, 2009.

[15] I. Bate and P. Emberson. Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems. In *12th IEEE RTAS Conference*, pages 221–230, April 2006.

[16] D. Baudisch, J. Brandt, and K. Schneider. Dependency-driven distribution of synchronous programs. In *DIPES/BICC*, 2010.

[17] A. Benveniste, A. Bouillard, and P. Caspi. A unifying view of loosely time-triggered architectures. In L. P. Carloni and S. Tripakis, editors, *EMSOFT*, pages 189–198. ACM, 2010.

[18] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

[19] 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.

[20] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1):64–83, January 2003.

[21] A. Benveniste, P. Caspi, P. Guernic, et al. A Protocol for Loosely Time-Triggered Architectures. In *Proc. of the 2nd international conference on embedded software*, pages 252–265, 2002.

[22] E. Bini, M. Di Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Euromicro Conference on Real-Time Systems*, Dresden, Germany, June 2006.

[23] Bloomberg. Boeing Delays 787's First Flight to November-December, September 2007.

[24] Bloomberg. Recalls Triple as Electronics Run Cars, Swamp U.S. Regulators, March 2010.

[25] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, 1991.

[26] L. Carloni, F. De Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi. Platform-Based Design for Embedded Systems. In *The Embedded Systems Handbook*. CRC Press, 2005.

[27] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(9), 2001.

[28] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28:1437–1455, October 2009.

[29] P. Caspi and A. Benveniste. Time-robust discrete control over networked loosely time-triggered architectures. In *CDC*, pages 3595–3600. IEEE, 2008.

[30] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications. In *LCTES'03*, 2003.

[31] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*. ACM, 1987.

[32] International Data Corporation. Intelligent Systems Transforming the Embedded Industry, According to IDC, September 2011.

[33] 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.

[34] 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.

[35] J.J.G. Garcia and M. G. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *3rd Workshop on Parallel and Distributed Real-Time Systems*, 1995.

[36] T. Gautier and P. Le Guernic. Code generation in the SACRES project. In *Safety-critical Systems Symposium, SSS'99*, pages 127–149. Springer, 1999.

[37] A. Hamann, R. Racu, and R Ernst. A formal approach to robustness maximization of complex heterogeneous embedded systems. In *Proc. of the CODES/ISSS Conference*, October 2006.

[38] A. Hamann, R. Racu, and R Ernst. Methods for multi-dimensional robustness optimization in complex embedded systems. In *Proc. of the ACM EMSOFT Conference*, September 2007.

[39] A. Hamann, R. Racu, and R Ernst. Multi-dimensional robustness optimization in heterogeneous distributed embedded systems. In *Proc. of the 13th IEEE RTAS Conference*, April 2007.

[40] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.

[41] K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[42] Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.

[43] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[44] E. A. Lee and T. Parks. Dataflow process networks. In *Proc. of the IEEE*, pages 773–799, 1995.

[45] R. Lublinerman and S. Tripakis. Modular code generation from triggered and timed block diagrams. In *RTAS*, 2008.

[46] J. Lygeros, C. Tomlin, and S. Sastry. Controllers for reachability specifications for hybrid systems. *Automatica*, 35:349–370, 1999.

[47] M. Maasoumy, A. Pinto, and A. Sangiovanni-Vincenteli. Model-based hierarchical optimal control design for HVAC systems. In *Dynamic System Control Conference (DSCC), 2011*. ASME, 2011.

[48] A. Metzner and C. Herde. RTSAT– an optimal and efficient approach to the task allocation problem in distributed architectures. In *Proc. of the IEEE RTSS Conference*, 2006.

[49] A. K. Mok and W.-C. Poon. Non-preemptive robustness under reduced system load. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 200–209, Washington, DC, USA, 2005. IEEE Computer Society.

[50] M. Di 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.

[51] National Science and Technology Council, Committee on Technology. Federal research and development agenda for net-zero energy, high-performance green buildings. 2008.

[52] ILOG CPLEX Optimizer. http://www.ilog.com/products/cplex/.

[53] 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.

[54] 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.

[55] 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.

[56] A. Pinto, A. 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.

[57] Alessandro Pinto, Luca P. Carloni, and Alberto L. Sangiovanni Vincentelli. COSI: A Framework for the Design of Interconnection Networks. *IEEE Design and Test of Computers*, 25(5), 2008.

[58] P. Pop, P. Eles, and Z. Peng. Schedulability-driven frame packing for multicluster distributed embedded systems. *Trans. on Embedded Computing Sys.*, 4(1):112–140, 2005.

[59] P. Pop, P. Eles, Z. Peng, and T. Pop. Scheduling and mapping in an incremental design methodology for distributed real-time embedded systems. *IEEE Trans. VLSI Syst.*, 12(8):793–811, 2004.

[60] T. Pop, P. Eles, and Z. Peng. Design optimization of mixed time/event-triggered distributed embedded systems. In *Proc. of the CODES+ISSS Conference*, New York, NY, USA, 2003. ACM Press.

[61] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[62] R. Racu, M. Jersak, and R. Ernst. Applying sensitivity analysis in real-time distributed systems. In *Proc. of the RTAS Conference*, San Francisco (CA), U.S.A., March 2005.

[63] J. Romberg and A. Bauer. Loose synchronization of event-triggered networks for distribution of synchronous programs. In *EMSOFT'04*, pages 193–202. ACM, 2004.

[64] A. Sangiovanni-Vincentelli. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[65] C. Sofronis, S. Tripakis, and P. Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proc. of the 6th international conference on embedded software*, pages 21–33, 2006.

[66] IEEE Spectrum. This Car Runs on Code, February 2009.

[67] 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.

[68] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14(3):219–250, 1998.

[69] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. Comput.*, 57(10):1300–1314, 2008.

[70] M. Wetter and P. Haves. Modelica library for building hvac and control systems. https://gaia.lbl.gov/bir.

[71] F. Xia, F. Hao, I. Clark, et al. Buffered asynchronous communication mechanisms. *Fundam. Inf.*, 70(1):155–170, 2005.

[72] F. Xia, A. Yakovlev, I. Clark, et al. Data communication in systems with heterogeneous timing. *IEEE Micro*, 22(6):58–69, 2002.

[73] Y. Yang, S. Tripakis, and A. Sangiovanni-Vincentelli. Efficient distribution of triggered synchronous block diagrams. (UCB/EECS-2011-115), Oct 2011. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-115.html`.

[74] R. Yerraballi and R. Mukkamalla. Scalability in real-time systems with end-to-end requirements. In *Journal of Systems Architecture*, volume 42, pages 409–429, 1996.

[75] W. Zheng, Q. Zhu, M. Di Natale, and A. Sangiovanni-Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *RTSS*, pages 161 –170, 2007.

[76] W. Zheng, Q. Zhu, M. Di Natale, and A. Sangiovanni-Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *Proc. of the IEEE RTSS Conference*, 2007.

[77] Q. Zhu, Y. Yang, M. Di Natale, E. Scholte, and A. Sangiovanni-Vincentelli. Optimizing the Software Architecture for Extensibility in Hard Real-Time Distributed Systems. *the IEEE Transactions on Industrial Informatics*, 6(4):621–636, 2010.

[78] 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.

[79] Q. Zhu, H. Zeng, W. Zheng, M. Di Natale, and A. Sangiovanni-Vincentelli. Optimization of Task Allocation and Priority Assignment in Hard Real-Time Distributed Systems. *accepted by the ACM Transactions on Embedded Computing Systems, special issue on the synthesis of cyber-physical systems*, 2012.

[80] J. Zou, S. Matic, E. A. Lee, et al. Execution strategies for ptides, a programming model for distributed embedded systems. In *Proc. of the 15th Real-Time and Embedded Technology and Applications Symposium*, pages 77–86. IEEE Computer Society, 2009.