SOFL-based Dependency Graph Generation for Scheduling

Zhuo Cheng*, Haitao Zhang†, Yasuo Tan*, and Yuto Lim*
*School of Information Science, Japan Advanced Institute of Science and Technology, Japan
{chengzhuo, ytan, ylim}@jaist.ac.jp
†School of Information Science and Engineering, Lanzhou University, China
htzhang@lzu.edu.cn

***Abstract*—In multi-task systems, different tasks work together to achieve desired functions. To guarantee the correctness of the functions, the tasks are required to be completed in specific orders. Violation of such orders will lead to the systems in unpredictable states, which may cause disasters. However, with the continuously increasing complexity in the developments of systems, to correctly generate the task dependency relation is becoming a challenge. A primary problem is the requirement specification may not be accurately and easily understood by the developers carrying out different tasks. To solve this problem, formal specification provides a feasible solution. However, some difficulties (e.g., high requirement of significant abstraction and mathematical skills) has hindered the widely usage of formal methods. To address these difficulties, SOFL, a formal engineering methodology, has been proposed. In this paper, we propose a method for generating task dependency relation based on SOFL specification. This method is demonstrated through a detailed case study of cruise control system. Moreover, we also provide a checking algorithm to check if there exist mistakes in the SOFL specification based on the transitivity of task dependency relation. We believe that these works provide a firm basis for the design of scheduling.**

***Keywords*—scheduling, task dependency, SOFL, formal specification, cruise control system**

I. Introduction

Currently, almost all the practical systems are multi-task systems, such as chemical and nuclear plant control, space missions, vehicle control systems, telecommunications, and multimedia systems. In such systems, different tasks work together to achieve the desired functions. In order to guarantee the correctness of the functions, the tasks are required to be completed in specific orders. Any violation of such orders will lead to the systems in unpredictable states, which may cause disasters.

The execution order of tasks (i.e. schedule of tasks) is decided by the scheduler of the system. A scheduler decides the schedule of tasks based on its used scheduling algorithm. Research on scheduling algorithms has been for decades. Most of the research assumes the tasks are independent with each other (e.g., [1, 2, 3]), which are not quite practical for real systems. Some works (e.g., [4, 5, 6]) have considered the task dependency relationship. These works mainly focus on designing scheduling algorithms to schedule task set with such dependency relation. They assume the task dependency relation is known *a prior*. However, correctly generating such dependency relation is not easy, which requires carefully analyzing the requirement specification at the stage of designing the systems.

With the continuously increasing complexity in the developments of systems, to correctly generate the task dependency relation is becoming a challenge. A primary problem is that the requirement specification may not be accurately and easily understood by the developers carrying out different tasks. The major reason causing such problem is the notations and languages used in the specification lack of precise syntax and semantics. These notations and languages inevitably associate ambiguity and may lead to misunderstanding. To solve this problem, formal specification gives a feasible solution. With precise constrains of semantics and syntax, formal specification can precisely define behaviors of the system and provide a firm basis for the next developers to design the system.

Unfortunately, there exist some difficulties in using formal methods. For example, it requires significant abstraction and mathematical skills; it usually costs more in time and human effort for analysis and design [7]. These difficulties have hindered the widely usage of formal methods. To address these difficulties, SOFL, a formal engineering methodology, has been proposed in [7, 8], where SOFL refers to *structured-object-based-formal language*. It proposes changes to software process, notation, methodology, and support environments for constructing systems, which makes formal methods more practical and acceptable. To study how SOFL can be used to help developers generate the task dependency relation, in this paper,

- *we propose a method for generating task dependency relation based on SOFL specification.* The task dependency relation is expressed by a directed graph called *task dependency graph*. This method is demonstrated through a detailed case study of cruise control system. Through the case study, we can see that SOFL specification can effectively help developers to correctly generate the dependency relation of tasks, which provide a firm basis for the design of scheduling.
- *we provide a checking algorithm rERA to check if there exist mistakes in the task dependency graph based on the transitivity of task dependency relation.* If such mistakes exist, rERA can detect and fix such mistakes. As the task dependency graph is generated form the SOFL specification, which means the mistakes in the SOFL specification can also be detected and fixed.

Fig. 1. Function buttons on the control lever of a cruise control system. (The figure is from the home page of Audi.)

The remainder of this paper is organized as follows. We first describe the cruise control system and gives its requirement specification in section II. The SOFL specification of the cruise control system is explained in section III. In section IV, we describe the method of generating task dependency graph from SOFL specification. A checking algorithm for task dependency graph based on the transitivity of task dependency relation is explained in section V. Concluding remarks are given in section VI.

II. Cruise Control System

Cruise control system is a servomechanism that can maintain a constant vehicle speed as set by the driver. It accomplishes this function by measuring the vehicle speed, comparing it to the set speed, and automatically adjusting the throttle according to a control algorithm. It is usually used for long drives across highways. By using the cruise control system, drivers do not need to control the throttle pedal to maintain the speed of vehicles, which can alleviate the fatigue of drivers. Meanwhile, it can reduce the unnecessary change of speed, which usually results in better fuel efficiency. With these advantages, cruise control system has now been widely equipped in various brands of automobiles, such as BMW, Audi, and Volkswagen.

Cruise control systems developed by different automobile manufacturers usually have different auxiliary functions. Fig. 1 shows the control lever of a cruise control system equipped in an Audi automobile. A driver can activate different functions by pressing the function buttons on the control lever. Button ON and OFF are to turn on and turn off the system, respectively. After system turns on, when the button SET is pressed, if the speed of the vehicle is within a specific speed interval which is supported by the cruise control system, the system will start to maintain current vehicle speed until the driver presses button OFF, or CANCEL, or steps on brake. SPEED+ and SPEED- is used to adjust the set speed when system keeps on maintain current vehicle speed. When SPEED+ is pressed, the set speed will be increased, and the system will increase current speed to the set speed and maintain the speed of vehicle at that level. The button CANCEL can temporarily turn off the system, meanwhile, the button RESUME can resume the system to the moment at which the system is temporarily turned off.

A. Requirement Specification

As our objective is to investigate how to generate task dependency relation from SOFL specification, rather than develop a fully functional system, for simplicity, we only consider parts of the functions. Moreover, to the consideration of safety, a new CONFIRM button is provided.

After system turns on, the primarily functions required by a cruise control system are as follows (as function button SET in Fig. 1 is not considered in our design, in the following parts of the paper, system turns on means the system stars to maintain current vehicle speed).

1) Let the driver increase and decrease the value of the set speed. The set speed is required within a speed interval supported by the cruise control system. To the consideration of safety, a confirm operation is needed to confirm the setting.
2) Keep on maintaining the vehicle speed at the set value.

Although above specification is very simple, it still may cause misunderstanding. For example, the sentence "a confirm operation is needed to confirm the setting" does not clearly describe what will happen if the confirm operation is not performed. A designer may think that if an operation of increasing or decreasing is not followed by a confirm operation, the operation will be ignored. While another designer may think that the confirm operation is needed only when the driver has finished the setting (maybe after pressing button SPEED+ and SPEED- many times). In order to avoid any potential misunderstanding, as described above, formal notation can help greatly.

III. SOFL Specification

A SOFL specification is a hierarchical condition data flow diagram (CDFD) that is linked with a hierarchy of specification modules (**s-modules**) [8]. The CDFD comprises a set of condition processes and describes data flows between them, while the linked s-modules precisely defines the functionality of the components (condition process, data flow, data store) in the CDFD. Each condition process in the CDFD is linked with a **c-process** which is defined in the s-modules and describes functions in terms of **pre** and **post** conditions, within the specific specification context of the module [9]. More details about SOFL specification can refer [7] [8].

A. CDFD

The CDFD of the cruise control system is shown in Fig. 2, and the linked s-model is shown in Fig. 3. In Fig. 2, each box surrounded by narrow borders denotes a process, such as `SET_adjust()` and `CRU_control()`, which describes an operation. It tasks inputs and produces outputs. Each directed line with a labeled variable name denotes a data flow. A solid line denotes an active data flow, while a dotted line denotes a control data flow. The box with a number and an identifier (e.g., `temp_speed`) is a data store which can be accessed by processes. A directed line from a data store to a process represents the process can read the data from the store, while a directed line from a process to a data store means the process can read, write, and update the data in the store. More details about the components used in the CDFD can refer [8] [10].

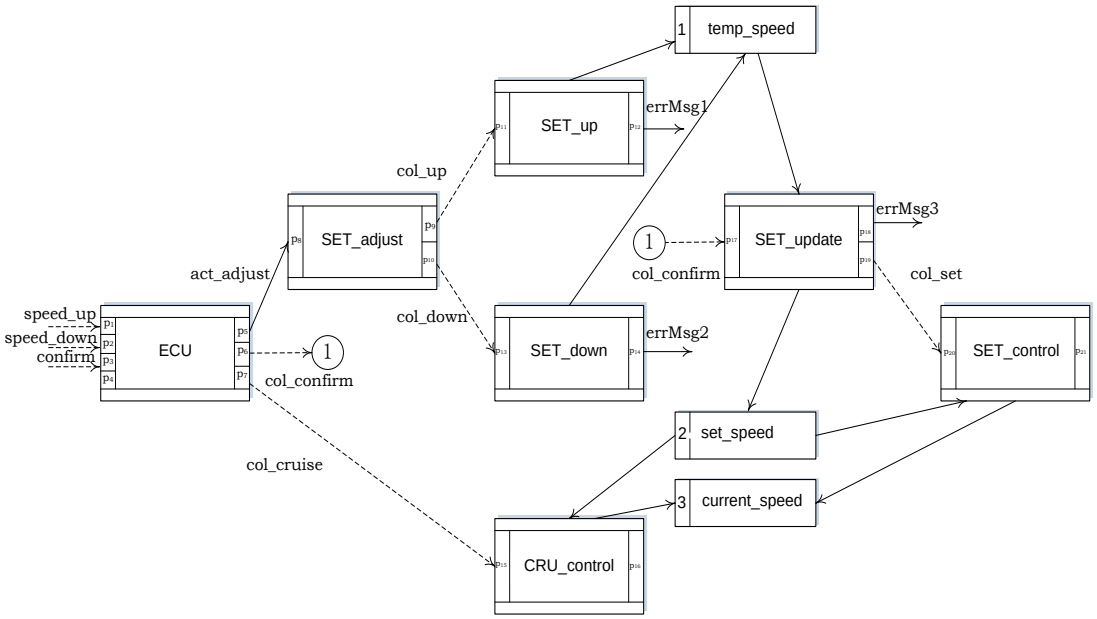

Fig. 2. Condition data flow diagram (CDFD) for the cruise control system.

The cruise control system comprises two primarily functions: set the desired vehicle speed (through increasing or decreasing the set speed) and maintain the vehicle speed at the set value. These two functions are triggered by `ECU()` (electronic control unit) process. Processes with names star with `SET` are for the first functions, and processes `CRU_control()` is for the second function. When the system is running, process `ECU()` keeps on monitoring the inputs of drivers. Different inputs will trigger different processes to achieve different functions. The selection of `speed_up`, `speed_down`, or `confirm` denotes the corresponding function button on the system control lever shown in Fig. 1 is pressed by the driver.

When button `speed_up` or `speed_down` is pressed, process `ECU()` will generate a data flow `act_adjust` to indicate which command has actually been selected, and passes this information to process `SET_adjust()`. Based on the value of `act_adjust`, process `SET_adjust()` will trigger either processes `SET_up()` (`speed_up` is selected) or `SET_down()` (`speed_down` is selected). Process `SET_up()` or `SET_down()` first reads the value of `temp_speed` from the data store, and try to update `temp_speed` by increasing or decreasing it with a constant value, respectively. As the cruise control system can only run within a designed speed interval, before updating the data, process `SET_up()` and `SET_down()` will first check if the updated value of `temp_speed` is within the interval. If not, an error message will be issued. Data `temp_speed` is a temporary data can be manipulated by process `SET_up()` and `SET_down()` and only after process `SET_update()` performs, the value of `temp_speed` can be assigned to `set_speed`. After process `SET_up()` or `SET_down()` completes the updates of `temp_speed`, it will sends the completion information to process `SET_update()`. Process `SET_update()` will assign data `temp_speed` to `set_speed` only after the `confirm` button is pressed. After process `SET_update()` assigns the value of `temp_speed` to `set_speed`, it will trigger process `SET_control()` to control current vehicle speed `current_speed` to the new set speed `set_speed`.

When no function button is pressed by the driver, it means the driver does not want to adjust the set speed and wants to maintain current vehicle speed, process `CRU_control()` will be triggered by process `ECU()`. Process `CRU_control()` maintains current vehicle speed `current_speed` to the value of `set_speed` based on a control algorithm.

B. s-module

Compared with the specification written in natural language given in section II-A, the functional abstraction expressed by the CDFD is obviously more comprehensible, especially, the dependency relations among processes can be clearly expressed. However, in order to completely define the CDFD, all the components (conditional process, data flows, data stores) in the CDFD must be precisely defined. To achieve this, the CDFD is linked with a s-model shown as in Fig. 3.

In the s-module, part **const** shows the constant variables used in the module. All the data flow variables, and data stores in the CDFD are defined in the **var** part. Each of them is defined in a specific data type. Keyword **inv** stands for invariant and indicates the properties that must be sustained throughout the entire specification. For example, `min_sp <= set_speed <= max_sp` in part **inv** means the setting value of the cruise control system must be larger than the maximum value that supported by the system and less than the minimum value. Function `Controller()` achieves the function of speed control based on a control algorithm. At this level of specification, the control algorithm has not been designed.

Process `Init()` is the initial process which performs only one time when the system stars up. We can see that, **pre** condition defines in the process `Init()` requires that `current_speed` should be less than `max_sp` and larger

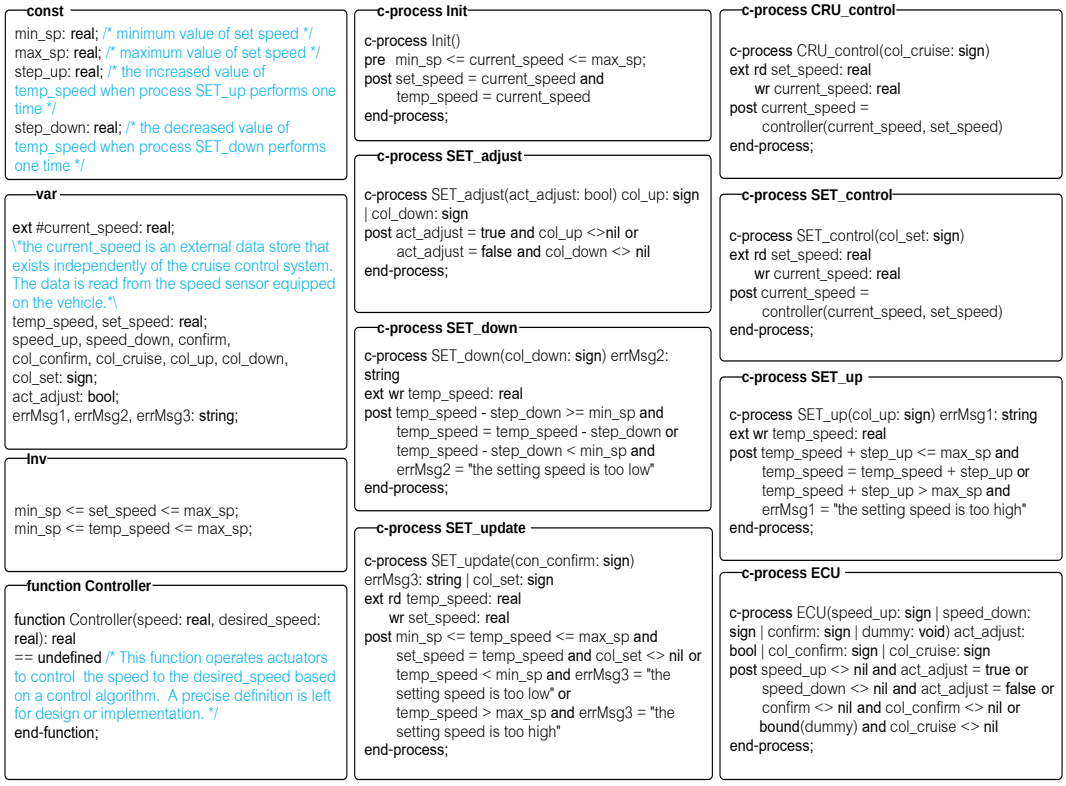

Fig. 3. s-module for the cruise control system.

than `min_sp`. This ensures that the system can star up only when vehicle is running within the speed interval that supported by the cruise control system.

Each processes in the CDFD is linked with a **c-process**. It describes functions of the processes in terms of pre and post conditions in which predicate logic is adopted. For example, the post condition in c-process `SET_adjust()` means: if the value of data flow variable `act_adjust` is true, process `SET_adjust()` will trigger `SET_up()` by generating control signal `col_up`, otherwise `act_adjust` with a false value will make process `SET_down()` be triggered.

IV. Task Dependency Relation Generation

After the SOFL specification is given, we can generate the task (i.e., process in SOFL specification) dependency relation directly from the specification. The task dependency relation can be expressed in a directed graph which is called *task dependency graph*. Its formal definition is as follow.

Definition (task dependency graph) *A task dependency graph is a directed acyclic graph. $G = (P, E, P_0, P_e)$, where P is port set of tasks, $E \subseteq P \times P$ is dependency relation (edge) set, with $(p_i, p_j) \in E$, $p_i \neq p_j$, where $p_i, p_j \in P$. $P_0 \subset P$ is the start port set, and $P_e \subset P$ is the end port set.*

An edge (p_i, p_j) in the task dependency graph means signal generated by port p_j can be issued only after port p_i generates its signal. We use $p_i \prec p_j$ to illustrate this dependency relation. The dependency relation is transitive. That is, $p_i \prec p_j, p_j \prec p_k \implies p_i \prec p_k$. A start port $p_i \in P_0$ features that $\forall p_j \in P$, $\nexists p_j \prec p_i$, while an end port $p_k \in P_e$ features that $\forall p_j \in P, \nexists p_k \prec p_j$.

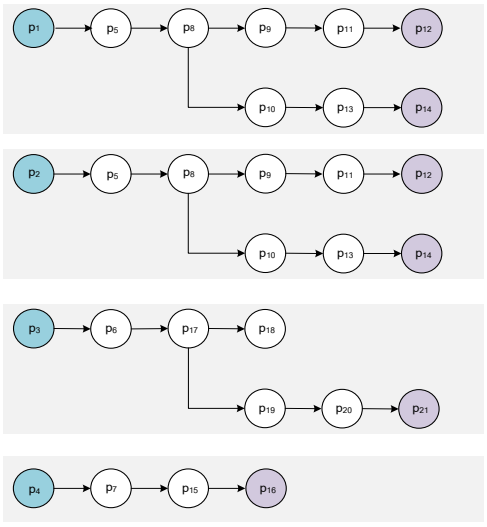

Fig. 4. Task dependency graph for cruise control system generated from the SOFL specification.

Based on this definition, from the SOFL specification, we can easily generate the task dependency graph. From the CDFD shown in Fig. 2, we can see that the input ports of process `ECU()` are the start port set. By checking the corresponding c-process `ECU()` in s-model shown in Fig. 3, we can know that port p_5 (the port index is denoted in each process of Fig. 2) can generate signal `act_adjust`

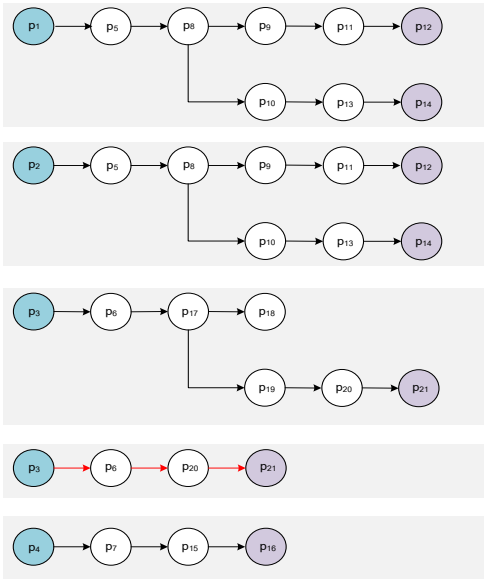

Fig. 5. Task dependency graph generated from the SOFL specification with a mistake.

only when port p_1 or p_2 accepts its signal `speed_up` or `speed_down`. This means there exist dependency relations between port (p_1, p_5) and (p_2, p_5), that is, there exist edges from p_1 to p_5 and from p_2 to p_5. By this method, we can get the task dependency graph for the cruise control system. The result is shown in Fig. 4. Note that, as either signals from port p_1 or port p_2 can activate port p_5 to generate its signal, we need two disjoint tree structures to represent the dependency relations.

In Fig. 4, there are four disjoint tree structures in the figures (denoted with shades). Each tree structure denotes a response of cruise control system to a specified operation of the driver. Ports p_1, p_2, p_3, p_4 are the start ports. The signal accepted by these ports are generated based on operations of the driver. Ports $p_{12}, p_{14}, p_{21}, p_{16}$ are the end ports. After their corresponding tasks finish running, a response of the cruise control system to the corresponding operation of the driver is completed. For example, if no function button is pressed by the driver, from the bottom tree structure, we can know that the running of task `CRU_control` (i.e., process `CRU_control()` in Fig. 2) will start to run, which is the corresponding response of the cruise control system.

V. Checking Algorithm

Task dependency graph is generated from SOFL specification to characterize task execution orders. If there are mistakes in the SOFL specification, by checking the dependency graph, some mistakes can be found. For example, if we accidentally define process `SET_control()` can be directly triggered by the signal `col_confirm` generated from port p_6 of process `ECU()`, with this mistake, by using the method described in section IV, we can get the task dependency graph shown in Fig. 5. The forth tree structure containing ports p_3, p_6, p_{20}, p_{21} with red lines are generated from the mistake. To check and correct such mistake, we propose an algorithm called *redundant Edge Removal Algorithm (rERA)*. The details are described in Alg. 1.

Algorithm 1 redundant Edge Removal Algorithm (rERA)

Input: task dependency graph $G = (P, E, P_0, P_e)$
Output: task dependency graph $G' = (P, E', P_0, P_e)$ without redundant edges
1: $E' := E$
2: **for all** node $p_i \in P$ **do**
3: compute $D(p_i)$, the set of descendants nodes of task p_i
4: compute $C(p_i)$, the set of child nodes of task p_i
5: **end for**
6: **for all** node $p_i \in P$ **do**
7: **for all** $p_j \in C(p_i), p_k \in D(p_i)$, and $p_j \in D(p_k)$ **do**
8: $E' := E' \backslash (p_i, p_j)$
9: $C(p_i) := C(p_i) \backslash p_j$
10: **end for**
11: **end for**
12: **return** the optimized graph $G' = (P, E', P_0, P_e)$

rERA maintains two sets for each node p_i: set $D(p_i)$, the set of descendants nodes of node p_i, and set $C(p_i)$, the set of child nodes of node p_i (line 2-5). In line 7, $p_k \in D(p_i)$ and $p_j \in D(p_k)$ indicate the dependency relation $p_i \prec p_k$ and $p_k \prec p_j$, respectively. Based on the transitivity of dependency relation, we can get $p_i \prec p_j$. If there exists $p_j \in C(p_i)$, which indicates the dependency relation $p_i \prec p_j$, as this dependency relation has already been indicated by other edges, the edge (p_i, p_j) is a redundant edge that should be removed from edge set E', and the node p_j should be removed from child node set of node p_i (line 8-9). Through using rERA, we can find the mistake and get the correct task dependency graph as shown in Fig. 4.

VI. Concluding Remarks

In multi-task systems, different tasks work together to achieve desired functions. To guarantee the correctness of the functions, the tasks are required to be completed in specific orders. To correctly generate the task dependency relation, in this paper, we propose a method based on SOFL formal specification. Through a detailed case study of cruise control system, we can see that the proposed method can effectively help developers to correctly generate task dependency graph. Moreover, we also provide a checking algorithm rERA to check if there exist mistakes in the task dependency graph based on the transitivity of task dependency relation. If such mistakes exist, rERA can detect and fix such mistakes. As the task dependency graph is generated form the SOFL specification, which means the mistakes in the SOFL specification can also be detected and fixed. We believe these works provide a firm basis for the design of scheduling.

References

[1] P. Mejia-Alvarez, R. Melhem, D. Mosse, and H. Aydin "An Incremental Server for Scheduling Overloaded Real-Time Systems," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1347–1361, 2003.

[2] A. Marchand, M. Chetto, "Dynamic Scheduling of Periodic Skippable Tasks in an Overloaded Real-Time System," In *Proc. IEEE/ACS International Conference on Computer Systems and Applications*, pp. 456–464, 2008.

[3] M. Xiong, S. Han, K.-Y. Lam, and D. Chen, "Deferrable Scheduling for Maintaining Real-Time Data Freshness: Algorithms, Analysis, and Results," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 952–964, 2008.

[4] S. Darbha and D.P. Agrawal, "SDBS: a task duplication based optimal scheduling algorithm," In *Proc. Scalable High-Performance Computing Conference*, pp. 756–763, 1994.

[5] P. Gai, M. Natale, and et al, "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform," In *Proc. Real-Time and Embedded Technology and Applications Symposium*, pp. 189–198, 2003.

[6] S.S. Craciunas and R.S. Oliver, "SMT-based Task- and Network-level Static Schedule Generation for Time-Triggered Networked Systems," In *Proc. International Conference on Real-Time Networks and Systems*, pp. 45–55, 2014.

[7] S. Liu, *Formal Engineering for Industrial Software Development using the SOFL method*. Springer-Verlag, 2004.

[8] S. Liu, A.J. Offutt, C. H.-Stuart, Y. Sun, and M. Ohba, "SOFL: A Formal Engineering Methodology for Industrial Applications," *IEEE Trans. Softw. Eng.*, vol. 24, no. 1, pp. 24–45, Jan. 1998.

[9] X. Wang and S. Liu, " An Approach to Declaring Data Types for Formal Specifications," *Proc. International Workshop on SOFL+MSVL*, LNCS, vol. 8332, pp. 135-153, Springer-Verlag, 2014,

[10] C. H.-Stuart and S. Liu, "An Operational Semantics for SOFL," *Proc. Asia-Pacific Software Engineering Conference*, pp. 52–61, Dec. 1997.