A Software Architecture Supporting Self-Adaptation of Wireless Control Networks

Yanzhe Cui[1], Richard M. Voyles[1], Xuexuan Zhao[2], Jiali Bao[2], Eric S. Bond[1]

***Abstract*— This paper focuses on building an architecture-based self-adaptation infrastructure for resource constrained robotic systems. The objective of this research is to provide an architecture to guide the engineers to conveniently build self-adaptive systems, alleviate the workload to program a task. The primary contributions of this paper are an architecture support of self-adaptive systems which enables a system to perform analyses on the system itself for such quality attributes as performance and reliability; a real-time functional components design framework for building self-adaptive embedded systems; and a mechanism to achieve fault tolerance in the presence of faults and uncertainties. We implement a simple, yet comprehensible case study, line-up-blocks, to demonstrate the effectiveness and simplicity of programming a self-adaptive task in the ReFrESH.**

I. Introduction

Robotic systems are playing increasingly important roles in the hostile and unpredictable environments encountered in search and rescue, nuclear waste cleanup and planetary exploration [1], [2]. However, studies on robotic systems used in the field have shown a noticeable lack of reliability in real world conditions [3], [4], [5]. As stated in [6], even the most carefully designed and tested robotic systems may behave abnormally under faults and uncertainty. Therefore, robotic systems should have the capability to cope with changing environments, variable resources, as well as the system errors while maintaining the goals and properties.

Engineers have responded to aforementioned self-adaptation need in somewhat limited forms through programming language features (such as exceptions and control flows) and in algorithms (such as fault-tolerant protocols). However, these mechanisms are often highly specific to the application and tightly bound to the code. As a result, systems with self-adaptation capability are costly to build.

To this end, as shown in Fig. 1, a self-adaptation infrastructure, called ReFrESH (*Re*configuration *Fr*amework for distributed *E*mbedded systems for *S*oftware and *H*ardware), was proposed in our last CASE paper [7]. While ReFrESH appears to be a conventional layered architecture, it differs in that it provides self-diagnosing and self-maintaining infrastructure support, built into a real-time operating system (RTOS), to manage functional requirements and non functional requirements across each robot boundary in the presence of uncertainties in the physical environment. In other words, ReFrESH explores a reflective view of self-adaptive systems. ReFrESH makes adaptation possible for engineers to easily specify reconfiguration strategies that are more global in nature which takes into consideration task goals and quality attributes.

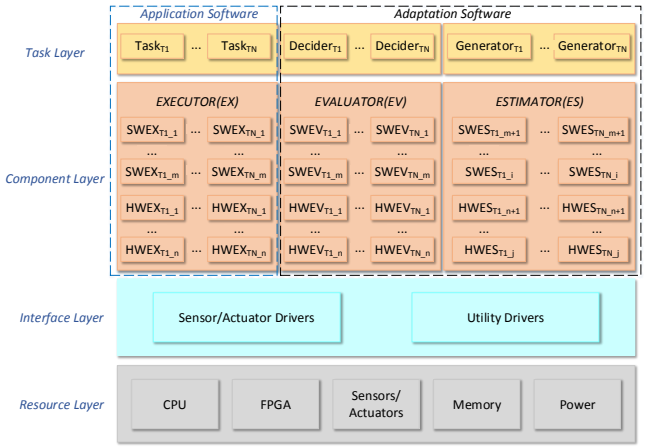


Fig. 1: 4-layer self-adaptation architecture to support fault tolerance and self-awareness. Evaluation and Estimation components support self-awareness in regard to both running and dormant functionality.

The basic ReFrESH architecture was introduced in [7] and this paper expands it. To guide engineers to flexibly build self-adaptive robotic systems, the expansion consists of: the *refined design philosophy* of each layer in ReFrESH, the *standard module design framework* that guides engineers to conveniently integrate ReFrESH and functionalities, as well as the *unified fault tolerance mechanism* within ReFrESH. To sum up, the contributions in this paper are:

- Elaborating the design philosophy of each layer in ReFrESH from responsibility, rationale, as well as behavior and capability perspectives;
- Proposing Extended Port-Based Object (E-PBO) to build the basis of a programming model to provide specific, yet flexible, guidelines for creating, integrating, monitoring and evaluating components;
- Demonstrating the fault tolerance mechanism for the known faults through the prescription known solutions and on-line synthesis of unknown solutions.

II. Related Work

Building self-adaptive systems is a major engineering challenge in terms of cost-effectively and predictable. New theories are needed to accommodate, in a systematic engineering

[1]Y. Cui, R.M. Voyles and E.S. Bond are with Purdue University, West Lafayette, IN, USA cui56, rvoyles, bond15@purdue.edu
[2]X. Zhao and J. Bao are with Tsinghua University, Beijing, China zhaoxx13, bjl13@mails.tsinghua.edu.cn

manner, traditional top-down approaches and bottom-up approaches [8]. Therefore, in this section, we start with the contributing disciplines that can be adopted in the design of self-adaptation architecture. Then, related self-adaptation architecture implementations are reviewed.

A. Functional Component Design

In robotics research, component-based software design decreases development time and is helpful to share the components among community. We investigate the state of art of module design frameworks in this part.

Generator of Modules (GenoM) [9] is a tool to design real-time software architectures which is more dedicated to complex autonomous mobile robots. GenoM provides an abstraction to create functional modules and it provides a tool to generate non-functional properties that are binded to modules. Open Robot Control Software (Orocos) [10] provides a set of libraries, it defines a functional module primitive and provides the non-functional resource information of a platform as well. OpenRTM [11] defines a module as separate core logic and a wrapper, which provide functional and non-functional characteristics of a module, respectively. However, these three module design methods cannot support self-adaptation, it means that there is no mechanism to change the structure of the configuration. The Chimera [12] Port-Based Object (PBO) module design framework defines the functionality, the input and output variable ports, module constants ports and service ports. It supports the static reconfiguration.

All aforementioned component design models specify the explicit connection mechanisms, it is capable of components integration. However, none of them is designed to support dynamic self-adaptation. In other words, they are not self-adaptive component design model. Therefore, in order to embedded component design model into a self-adaptive architecture, the model should have mechanisms of providing information for adaptation plans to analyze performance of system and synthesize configurations. The self-adaptive component model has three major advantages to support self-adaptive architecture:

- Model includes specific mechanisms for self-evaluation of the performance of execution units “in-vivo” at the component layer.
- Model includes specific mechanisms for estimation of the expected performance of execution units prior to execution “in- vitro” at the component layer.
- Model facilitates the design of component to include flexible mechanisms for deciding when performance has degraded and which hypothesized configurations are likely to exhibit improved performance at the adaptation plans.

B. Self-Adaptive Infrastructure Design

Self-adaptive infrastructure provides a tool to the engineers as a guidance to design a self-adaptation system. Using infrastructure, the workload can be decreased and the robustness of the systems would be increased. There is significant research in the area of self-adaptation architecture design. Multiple outstanding architectures inspired our research.

Rainbow [13] provides an extensible framework for sensors and actuators at the interface between the control infrastructure and the target systems. It focuses on achieving self-adaptation by changing component instances and bindings and also effecting behavior by changing operational parameters. Rainbow includes a set of pre-defined strategies to execute adaptation, so it does not account explicitly for automated construction of strategies that control the components.

Architecture in [5] was built on three-layered architecture. It applies three distinct, specialized meta-level components for the three fundamental activities of a robotic system: sensing, computation, and control. Meanwhile, it allows meta-level components to themselves be monitored, managed and adapted by other (higher layer) meta-level components. In this way, it provides intelligent facilities for constructing adaptation plans on-the-fly. PLAZMA [14] introduces an approach to software adaptation that utilizes modeling and planning techniques in a meta-layered architecture for self-adaptation. It simplifies the specification and use of adaptation mechanisms for system architects by freeing them from having to design the application architecture topology and plan for specific adaptations. As a result, the architecture avoids the difficulty of designing plans for unforeseeable conditions such as changing requirements and runtime failures.

Though aforementioned self-adaptive architectures solve the issues of runtime reconfiguration strategies generation, they (besides Rainbow) were not originally designed for distributed systems, so none of them specify an explicit mechanism for constructing configuration strategies with consideration of multi-agent systems.

IQ-ASyMTRe [15] aims to address both coalition formation and execution for tightly-coupled multi-robot tasks in a single framework. It enables the sharing of sensory and computational capabilities by allowing information to flow among different robots via communication. However, within this architecture, it neglects the fact that even though a coalition of components is generated, if the designated robot cannot host some capability, this configuration is indeed invalid.

The self-adaptive architecture proposed in this research provides a tool to decrease the workload of engineers by offering a design that supports self-contained/self-adaptive modules, separation of task and task repair, and dynamic synthesis of strategies.

III. Case Study Scenario

To illustrate the utility of the ReFrESH architecture, we pick a simplified task that is comprehensible, yet has sufficient complexity to present programming challenges.

We will use a 4 degree-of-freedom (DOF) arm and three 20 millimeter width blue blocks, as shown in Fig. 2 to execute a line-up-blocks task. The task is to search for the locations of three blocks, then pick them one by one and

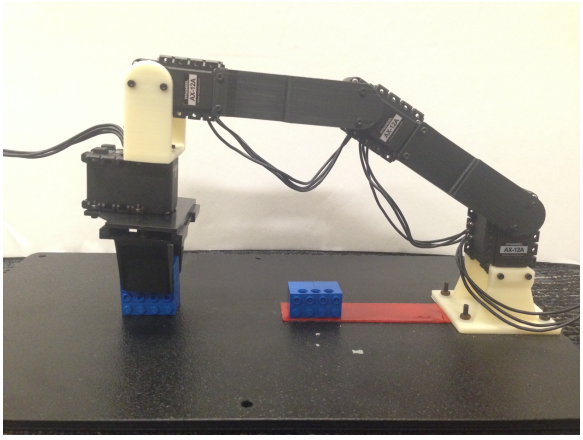


Fig. 2: 4 DOF arm used for line up blocks application.

place them in their corresponding locations. In order to show the simplicity of ReFrESH, we set up two error conditions in the system, one is that 'block fell out from the gripper' and another is that 'block collided with other blocks'. Therefore, when the task performance violates the above two rules, there is fault and uncertainty in the system.

IV. ReFrESH Implementation

In this section, ReFrESH implementation is introduced, which consists of: ReFrESH infrastructure design philosophy of four layers; E-PBO framework design to support self-adaptation; and fault tolerance for known faults through both prescription and on-line synthesized configuration.

A. ReFrESH Infrastructure Design Philosophy

ReFrESH, as shown in Fig. 1, is structurally divided into four layers: *Resource Layer*, *Interface Layer*, *Component Layer* and *Task Layer*.

1) Resource Layer: The responsibility of Resource Layer is to abstract the capability of a robot and provide the physical hardware support of building a self-adaptive system. The rationale for this layer is to describe the physical hardware and hosts the sensors/actuators interconnection, provides the resource knowledge to decision making unit. The Resource Layer is inspired by the fact that the performance of a system is affected by both uncertainties from its operating environment and from system itself. Changing environment may conflict the "functional requirement" of a task, such as in the line-up-blocks task, a vision sensor is blocked by dust which consequently degrades the target identification performance and then stops the whole system from successfully completing the task; degradation of the system itself may conflict the "non-functional requirement" of a task. Therefore, in order to specify self-adaptive strategies comprehensively, self-adaptive architecture should have capability to consider both "functional requirement" and "non-functional requirement". In ReFrESH, "functional requirement" is related to functionalities; while "non-functional requirement" is related to the physical equipments, such as CPU, FPGA AREA, MEMORY, POWER, SENSORS/ACTUATORS.

2) Interface Layer: The responsibility of the Interface Layer is to provide an abstract interface to access physical hardware and query system capability from the Resource Layer. The rationale for this layer is to provide reliable data(packet) transfer services to the layer above it. Specifically, through Interface Layer, point-to-point link is built between physical resource and their corresponding software or hardware functionalities. The Interface Layer is inspired from our previous work on Morphing Bus [16]. Morphing Bus eliminates the notion of an intermediate data format and thereby obviates the need for bus interface circuitry. Instead of a common data format to which all sensors and actuators are translated, the Morphing Bus transforms - or "morphs" - its signal lines to meet the needs of the connected sensors or actuators. For digital sensors and actuators, this morphing is achieved through a Field-Programmable Gate Array on the processor side of the bus.

3) Component Layer: The responsibility of Component Layer is to achieve system and task goals, encapsulate implementation details and provide abstract monitoring and control mechanisms over which the behavior and structure of the system can be adapted. The rationale for this layer is to encapsulate the instrumentation of the self-adaptive system to support a flexible and reusable framework for performance monitoring, evaluating, estimating and executing adaptation strategies provided in the layer above. The Component Layer contains the component framework that provides the managed system's functionality (*Executor (EX)*). It also contains instrumentation for monitoring (*Evaluator (EV)*) and estimating (*Estimator (ES)*) of components. ES provides mechanism for estimating the performance of a component, which is different from EV: EV only monitors performance of executing components in the current configuration; whereas ES "monitors" performance of components in the novel configuration strategy where some of the components may exist in the original configuration but others may not. This is the validating process before actually instantiating a novel configuration and is the reason we call this kind of monitoring as "estimating". The details of this component design framework will be presented in Section IV-B.

4) Task Layer: The responsibility of Task Layer is to define a task specification and goals, detect the degradation in the system, and cope with changes by synthesis self-adaptation strategies. The rationale for this layer is to be capable of performing strategic, computationally expensive, planning independently and concurrently with the execution of current configuration. The Task Layer takes charge of determining the necessity for fault alleviation, re-assembling components based on the requirements of a task, and generating an optimal configuration policy at runtime. Unlike a conventional Task Layer which only includes a state machine to control each component to satisfy the requirement of a task, such as by turning a component on/off; ReFrESH also extends the scope of the *Task* from the "execution" perspective to the "fault toleration" perspective by adding the "Decider" and "Generator" in order to support the capability of runtime reconfiguration. To accomplish this, the "Decider"

first requests all component related performance information from the *EV* to judge whether there is any reconfiguration requirement. If there is a need for reconfiguration, which signifies that one of the components or the configuration as a whole are not suitable for executing the current task, the "Generator" will be instantiated. The "Generator" executes in two phases: (1) dependency analysis, which shows the construction of a configuration and decides if a component could be replaced or amended by another homogeneous functional component; and (2) functional assembly, which shows the process of combining required components to generate all potential configurations. Since ReFrESH is based on the PBO kernel and PBO/RT provides flexible interfaces to control each component, it is convenient to analyze the dependency and link the components in the configurations together. For these generated configuration candidates, the corresponding *ES* will be instantiated to estimate the performance of each component in each candidate for the current task which helps the "Decider" to produce an optimal configuration. The implementation details of each of the aforementioned basic services in this level were illustrated in [17].

B. Self-Adaptive Module Design Framework

The Extended Port-Based Objects (E-PBOs) is modeled after the PBO [12] and adapts it to the self-adaptive robotic system design. Besides of the advantage of automatically assembling components of PBO, E-PBO framework also has two main advantages from self-adaptation support perspective: (1)it builds the basis of a programming model that provides specific, yet flexible, guidelines for robotics application engineers to create and integrate software components; (2) it forms the basis of a self-adaption model that provides specific methods for evaluating the running task configuration and estimating the new but non-running task configuration (if required) without interfering with the running configuration.

As shown in Fig. 3, to create the E-PBO the algebraic model of the port automation, which is similar to PBO is applied. Generally speaking, E-PBO is composed of two parts. One part is conventional **PBO Executor (EX)**, which defines the functionality and provides ports to communicate with other executing E-PBOs, reconfigure the module constants, or connect to sensors/actuators. The other part is the extended part, which consists of the **E-PBO Evaluator (EV)** and the **E-PBO Estimator (ES)**.

1) Evaluator - EV: The Evaluator (EV) of an E-PBO provides an online evaluation of the current level of performance of the E-PBO. EV runs concurrently with EX; it has full access or visibility to the internal states of the E-PBO. Engineers could specify the evaluation criteria, which provides a flexible way to modify the functionality. Unlike EX, EV only includes *evaluator performance output ports*, which are needed to supply the functional performance (such as target detection accumulated error) and non-functional performance (required power usage versus system power capability). Also, EV does not communicate with other E-PBOs, but instead connects to a system management unit.

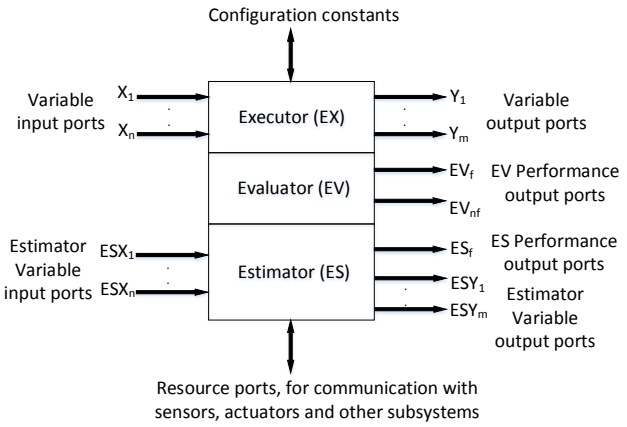


Fig. 3: Abstract view of an E-PBO. Subscript f denotes functional performance and nf denotes non-functional performance.

The outputs of EV provide the evidence for self-adaptation to the system to determine whether the performance of a task configuration is satisfactory or not.

2) Estimator - ES: The Estimator (ES) of an E-PBO provides an online estimation of the dormant E-PBO. ES runs separately with EX while the system has a novel configuration to cope with faults. ES has no access to the internal states of the E-PBO, thus it includes the same functionality of its corresponding EX. E-PBOs can be connected and communicate with other ES's through its standalone sets of ports: *estimator variable input ports* and *estimator variable output ports*. Moreover, *estimator performance output ports* provide the functional performance estimation to the *Decider*. After connecting all of the ES in a novel configuration, the Decider determines if the new task configuration is suitable or not. The ES of each E-PBO provides the recommendation for self-adaptation so that Decider can determine which new configuration should be used in the presence of faults in place of the current configuration.

Fig. 4 shows the configuration of line-up-blocks task that uses E-PBOs within ReFrESH. We use the same color code as Fig. 1. ReFrESH contains the task finite state machine (FSM) and E-PBOs in the layered architecture. The EV provides two indicators to the "Decider" to evaluate the task performance dynamically: functional performance is connected to *Func Performance Buffer* (red line in Fig. 4). For example, if the block falls out from gripper, the force on gripper, $Ev\_Fgripper$, would change to 0, thus the Decider would notice this fault and would call Generator to synthesis the novel solutions to repair this faulty situation.

C. Fault Tolerance in ReFrESH

ReFrESH provides service to evaluate task performance and to detect the fault. In this paper, we focus on tolerant known faults. In line-up-blocks task, we define two types of faults in Section III: (1) blocks fall out from gripper; and (2) blocks collided with other blocks.

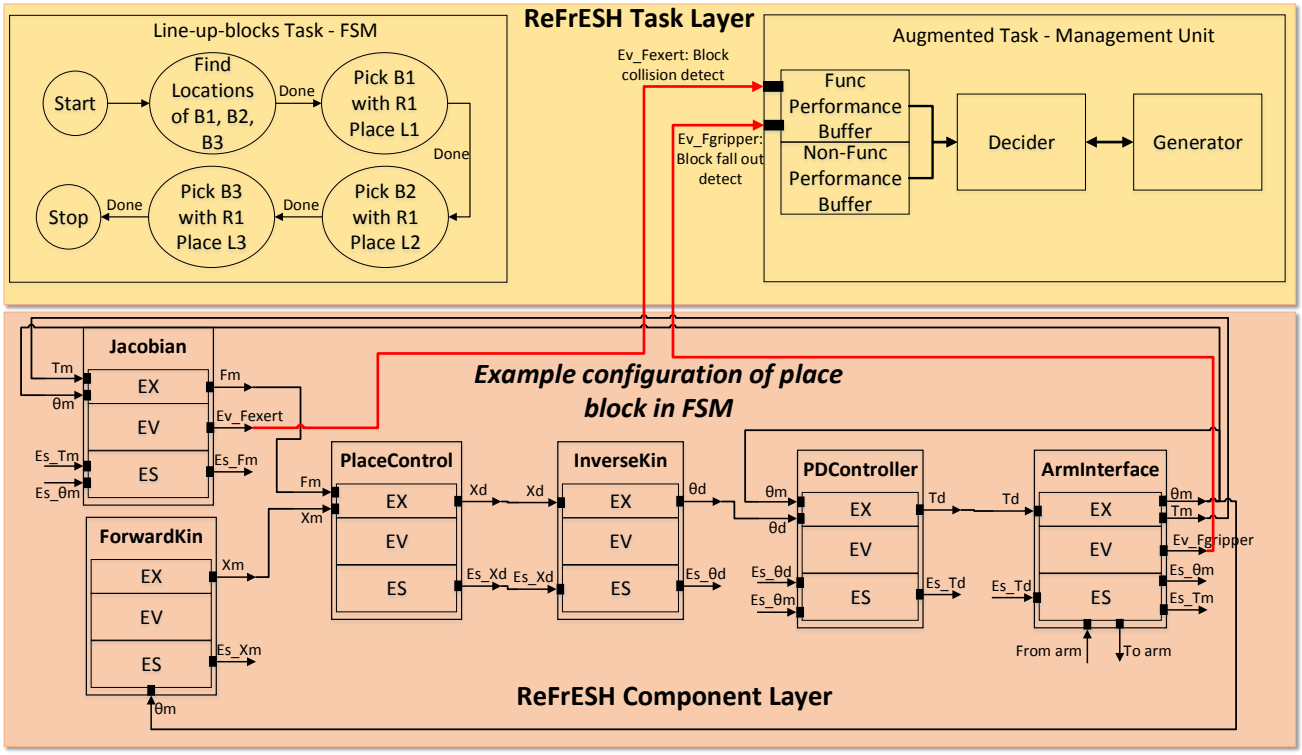


Fig. 4: An example of line-up-blocks configuration built by E-PBOs.

Even if the faults are predefined, the solution to repair the faults can be divided two categories that the ReFrESH can handles: one is prescription solution, which are pre-compiled by engineers, the ReFrESH uses these solutions directly; another is on-line synthesized solution, which is synthesized by the Generator in ReFrESH dynamically.

Prescription solution is trivial, since in the Validation and Verification phase before deploying system, engineers can assume and test the solutions based on the faults. The hard part for a self-adaptive systems is to 'think by itself', means how the robotic systems create a feasible novel solution based on its faulty situation. In this section, the online-synthesis of solution method is explored.

1) Brute Force Method: Brute-force search is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. In ReFrESH, a configuration is composed by instantiating a series of components and connecting their appropriate ports together. Component dependency defines the relationship between these components to show how their ports may be connected. The E-PBO concept provides a flexible component infrastructure based on port automata that makes component dependency analysis easy to perform. Each E-PBO has input, output, and resource ports and processes data from input to output as a result of a specific event. Communication between components is restricted to the input and output ports since the port names can be used to perform bindings. Therefore, with component dependency analysis we know which components in the system can be connected and in what manner. With this information it is straightforward to execute functional assembly.

As shown in Fig. 5, the line-up-blocks task can be decomposed into a series of actions that are performed by different E-PBOs. To point out, the 'E' in figure stands for E-PBO, 'SW' stands for software design and 'HW' stands for FPGA hardware core design. For a given task, a configuration candidate can be constructed by linking compatible E-PBOs that perform the appropriate actions. In other words, relevant *Components* are mapped to a configuration through a predefined correspondence between *Actions* and *Implements*. In general, there are multiple interchangeable components that can all implement the same action, such as vision action, multiple E-PBOs which corresponding to different CCD camera can provide image. In effect, there is more than one configuration that could successfully accomplish a task. We have demonstrated the details for comprehensive component analysis and component runtime construction in our previous CASE paper [7].

The brute-force-based configuration synthesis process in ReFrESH, however, can be classified to the well-know NP-complete problem PARTITION, thus this method is NP-hard problem. The main disadvantage of this method is that, for many real-world problems, the number of natural candidates is prohibitively large. For the small number of modules, the combinatorial problem will not explode, such as the case

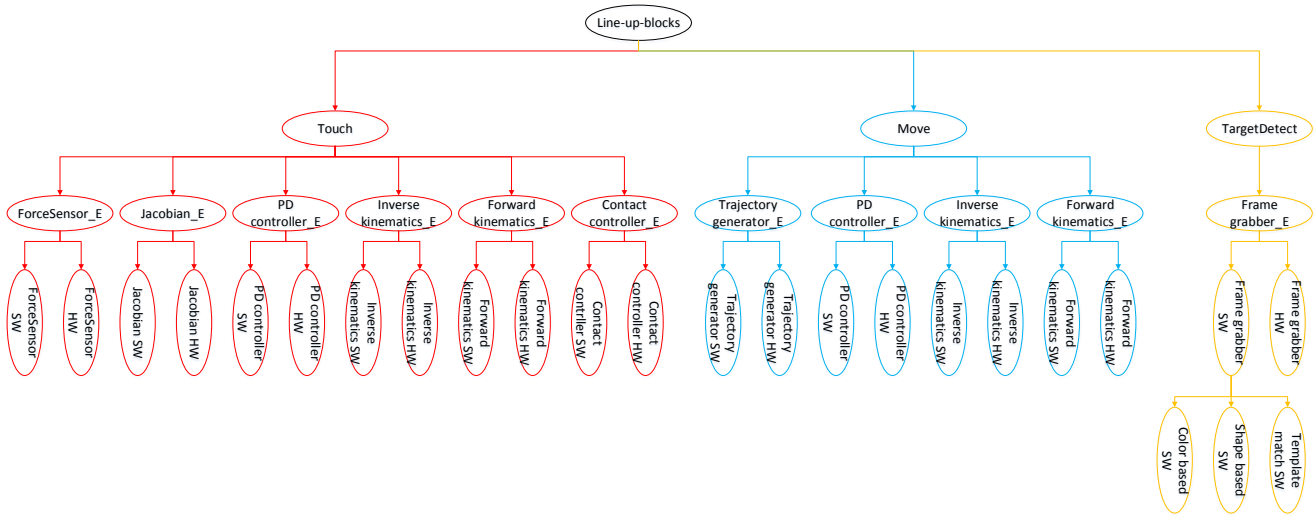


Fig. 5: Modules library to build line-up-blocks task in the format of tree structure.

in [7]; however, when the robot should execute a relatively complex task, the number of modules are large, so the search space will explode. Thus, brute force method is inapplicable in many cases.

2) Ontology Based Method: One way to speed up a brute force method is to reduce the set of candidate solutions, by building a structure for actions and using heuristics specific to the problem class. For example, in the configuration of line-up-blocks task, since all cameras related E-PBOs can be the alternatives of others in 'TargetDetect', in principle there are n cameras to be considered. However, because the physical settings are not alike, thus some cameras cannot be replaced by others, the candidates are all possible ways of choosing of a set of cameras from the set all n to j, where $j \subset n$. For other 'Touch' and 'Move' branches, the search space can be deduced based on the similar rules as the camera case. Therefore, we can further restrict the set of candidates to the task. Also, based on the structure, the Generator can use information from the Decider in ReFrESH to largely decrease the search space. For example, if the camera is broken, and the Decider decides to use 'teleoperation' to finish the left task. In this case, the Generator does not need to search from the 'TargetDetect' (in Fig. 5), the whole branch of it can be trimmed.

To better solve this NP-hard problem, we are urged to propose the semantics support to reduce the search space of candidate solutions. For example, if we know the fault of line-up-blocks task is that 'block falls out' and Decider in ReFrESH reports this fault, the system knows the fault is not from 'TargetDetect' and 'Touch', so it only needs to search a repair solution in action 'Move', then the branches of 'Touch' and 'TargetDetect' can be trimmed. We are going to adopt ontology techniques to define the domain knowledge, which consists of objectives (the functionality of a module), properties (a set of attributes and constraints), as well as the relationship with other modules (modules required inputs and provided outputs). To point out, this paper is an initial exploration about semantics and ontology, so we use this knowledge to build semantics. In the future work, based on experiment results and potential demand, we will adjust chosen features or add more features in semantics.

V. Case Study

The target of designing the ReFrESH infrastructure is to provide a tool to help engineers conveniently programming a self-adaptive system by consuming the least workload. Thus, in this section, through the illustration of line-up-blocks task, we show the various efficiency of programming a task between using conventional method (without ReFrESH) and using ReFrESH. To point out, in order to show the efficiency of using ReFrESH to program a task with self-adaptation capability, we only consider two faults: 'the blocks fall out from the gripper' and 'the blocks collide with other blocks'. The reason we only consider two fault types is that even if the fault number is rather small, programming a task to tolerant faults still adds a lot of design burden to the engineers without using ReFrESH. Using ReFrESH can alleviate the design burden greatly.

A. Programming a Task without ReFrESH

The conventional method of programming a robust task is that the engineers design a FSM. It includes the states of executing the task itself and the states that handling faults. Fig. 6 is the FSM of programming the line-up-blocks task. The black color ovals are the main flow of states of achieving the task, while the red color ovals are the faults handlers. For example, faults may happen in the execution of 'Pick B1 with R1 Place L1' state, to fix this fault, designer should add two more states to handle two possible consequences: (1) fell out from the gripper: adding finding B1's new location state, then execute picking B1 and place it in L1; (2) collided B2 and B3 while picking B1: adding the detection of new locations

of B2 and B3 state, then picking B2 from the new location and placing it in L2.

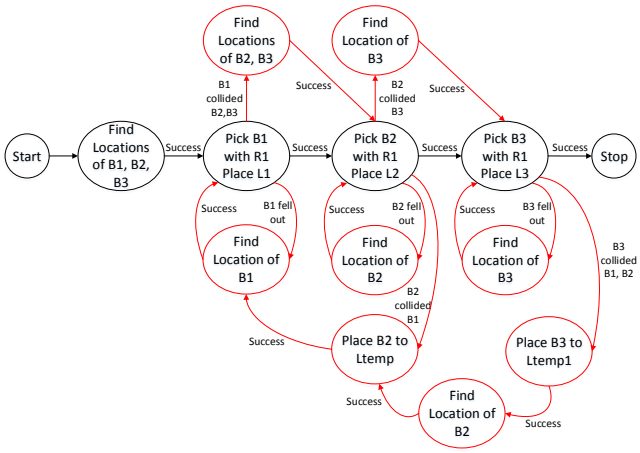


Fig. 6: Programming the line-up-blocks task w/o ReFrESH.

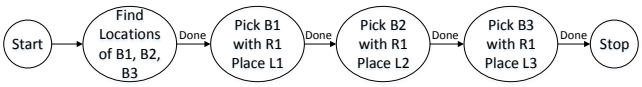
The programming is getting more complex for B3. For example, if B3 fell out, a new state of finding the location of B3 should be added and then re-do the pick and place; but if B3 collided to B1 and B2, since B1 and B2 were lined up already, it may change to location of B1 and B2, thus destroy the line of them. Thus, the system needs more states to handle this fault: placing B3 in its temporary location (Ltemp1); then finding the location of B2, picking B2 and placing B2 in its temporary location (Ltemp); then starting from handle the faults of B1 and so on.

We can see that, in this case study, the engineers should program 3 states to execute the task of line-up 3 blocks, as well as program 8 states to do fault tolerant. We can generalize the number of states: the number of states depends on the number of blocks (n) the system works on, thus n states; the number of states to tolerant faults depends on both the number of blocks and the number of faults the system tolerates (m), thus m^n states. Therefore, the totally number of states is $n + m^n$. The exponential increasing rate adds the huge programming burden to the engineers and makes building a task cumbersome.

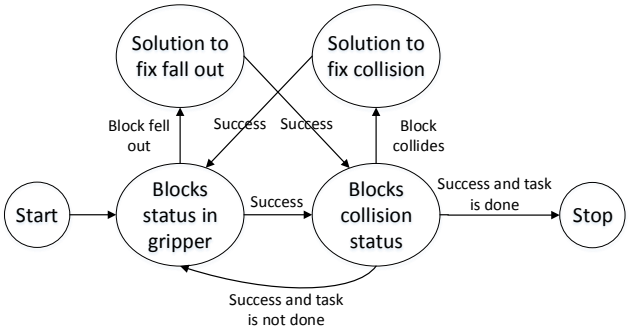
Furthermore, in order to detect the faults of falling out and collision, the engineers should also design the corresponding software modules to detect faults in each state and design the decision making unit to decide what actions should be taken. Without having infrastructure as the guidance, it is another obstacle and difficult part to program a task.

B. Programming a Task with ReFrESH

Programming a task using the ReFrESH is divided into two parts: main flow of a task, as shown in Fig. 7a; and errors handling of a task, as shown in Fig. 7b. For main flow of a task, the engineers are able to focus only on the execution of the task. The number of states in main flow depends on the number of blocks, thus n. Error handling is outside of the main flow of a task and it is preemptive to a task. The engineers should only abstract the faults of a system based



(a) Main programming flow to achieve a task.



(b) Preemptive errors handling flow of a task.

Fig. 7: Programming the line-up-blocks task with ReFrESH

on the performance criteria, and design a task repair FSM to cope with abstracted faults; then, based on each type of fault, generating corresponding solutions. For example, case study has two abstracted faults, so the engineers needs to add two more states to detect the faults and add another two more states to generate the solutions for each fault. From Fig. 7b, we can see that 4 states are shown in error handling flow.

We can also generalize the number of states of a task creating in ReFrESH: the number of states of the main flow depends on the number of blocks (n) the systems works on, thus n states; the number of states in error handling depends on the number of faults the system tolerates (m), thus $2m$ states. Therefore, the totally number of states is $n+2m$. The linear increasing rate alleviates the programming burden to the engineers and makes building a task easier.

1) Fault Detection through E-PBO: In order to detect faults in the task repair FSM, Decider needs information from corresponding E-PBOs. E-PBO provides the convenient self-adaptive module design framework for engineers. Two examples of how E-PBO support self-adaptation are demonstrated. E-PBO 'Jacobian' (as shown in Fig. 4) is used to detect block collision in this case study. The output of EX of 'Jacobian' is the force that exerts on the gripper, while the output of EV is the evaluation of force on the gripper. The EV output feeds into decider to detect collision. Similarly, 'GripperControl' E-PBO is to detect block fall out. The output of EV is the force indicated from the force sensor on gripper. Based on the output of EV, Decider can detect the fall out fault.

2) Solution Synthesis through Generator: In this case study, we have some prescription solutions pre-complied in the Generator for each fault. Thus, based on the various fault types, pick the different solution to repair the fault. To point out, we have tested the brute force method (NP-hard problem) in generating solutions in [7] in Generator, and we

will keep working on the semantics-based solution synthesis method to tackle this NP-hard problem.

TABLE I: Workload comparison (ReFrESH vs. w/o ReFrESH)

Method	Number of States	Extra Module Detect Fault
ReFrESH	$n + 2m$	No
None ReFrESH	$n + m^n$	Yes

3) Summary of Designing Fault Tolerant Systems: We summarize the efficiency of using ReFrESH to program a task analytically. As shown in Table I, the number of states that the engineers need to program is decreased from exponential $n + m^n$ to linear $n + 2m$ by using ReFrESH. Thus, even if building a complex system, the workload brought to the engineers just increases linearly; however, without using ReFrESH, it is tough to design a complex system. Moreover, the ReFrESH provides the E-PBO design framework to support self-adaptation, thus, the engineers do not need to design extra modules, which further alleviate design workload.

VI. Conclusions

We demonstrated the design philosophy of the ReFrESH infrastructure, E-PBO component design framework within ReFrESH to support self-adaptation, as well as the mechanism to do fault tolerance. Compared to the programming a task using conventional method, the ReFrESH provides the diagnosable and maintainable infrastructure support to help the engineers programming a self-adaptive task conveniently. Through the case study of line-up-blocks, we analytically illustrated that using the ReFrESH program a task decreases the workload from exponential complexity to linear complexity. In the future work, we will work on the exploration in ontology-based configuration synthesis methods to cope with the NP-hard problem in the ReFrESH.

ACKNOWLEDGMENT

This work was supported by National ScienceFoundation grants CNS-0923518 and CNS-1439717 with additional support from the NSF Safety, Security and Rescue Research Center. Xilinx, a member of the RoSe-HuB, provided support for hardware and software.

References

[1] N. Michael, S. Shen, K. Mohta, Y. Mulgaonkar, V. Kumar, K. Nagatani, Y. Okada, S. Kiribayashi, K. Otake, K. Yoshida, K. Ohno, E. Takeuchi, and S. Tadokoro, "Collaborative mapping of an earthquake-damaged building via ground and aerial robots," *Journal of Field Robotics*, vol. 29, no. 5, pp. 832–841, 2012. [Online]. Available: http://dx.doi.org/10.1002/rob.21436

[2] K. Nagatani, S. Kiribayashi, Y. Okada, K. Otake, K. Yoshida, S. Tadokoro, T. Nishimura, T. Yoshida, E. Koyanagi, M. Fukushima, and S. Kawatsuma, "Emergency response to the nuclear accident at the fukushima daiichi nuclear power plants using mobile rescue robots," *J. Field Robot.*, vol. 30, no. 1, pp. 44–63, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1002/rob.21439

[3] J. Carlson, R. Murphy, and A. Nelson, "Follow-up analysis of mobile robot failures," in *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, vol. 5, April 2004, pp. 4987–4994 Vol.5.

[4] L. Parker, "Alliance: an architecture for fault tolerant multirobot cooperation," *Robotics and Automation, IEEE Transactions on*, vol. 14, no. 2, pp. 220–240, Apr 1998.

[5] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, and B. Petrus, "Architecture-driven self-adaptation and self-management in robotics systems," in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, May 2009, pp. 142–151.

[6] L. Parker, *Reliability and Fault Tolerance in Collective Robot Systems*, May 2011.

[7] Y. Cui, R. M. Volyes, and M. H. Mahoor, "Refresh: A self-adaptive architecture for autonomous embedded systems," in *Conference on Automation Science and Engineering, 2013. CASE '13. Proceedings., 9th International Conference on*, 2013, pp. 1–6.

[8] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Software engineering for self-adaptive systems," B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Engineering Self-Adaptive Systems Through Feedback Loops, pp. 48–70.

[9] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "Genom3: Building middleware-independent robotic components," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, May 2010, pp. 4627–4632.

[10] H. Bruyninckx, "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3, 2001, pp. 2523–2528 vol.3.

[11] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. von Stryk, Eds. Springer Berlin Heidelberg, 2008, vol. 5325, pp. 87–98.

[12] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *Software Engineering, IEEE Transactions on*, vol. 23, no. 12, pp. 759–776, Dec 1997.

[13] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems," B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

[14] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic, "Plasma: A plan-based layered architecture for software model-driven adaptation," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 467–476. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859092

[15] Y. Zhang and L. Parker, "Iq-asymtre: Synthesizing coalition formation and execution for tightly-coupled multirobot tasks," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, Oct 2010, pp. 5595–5602.

[16] Y. Cui, R. Voyles, R. Nawrocki, and G. Jiang, "Morphing bus: A new paradigm in peripheral interconnect bus," *Components, Packaging and Manufacturing Technology, IEEE Transactions on*, vol. 4, no. 2, pp. 341–351, Feb 2014.

[17] Y. Cui, R. M. Voyles, J. T. Lane, A. Krishnamoorthy, and M. H. Mahoor, "A mechanism for real-time decision making and system maintenance for resource constrained robotic systems through refresh," *Autonomous Robots*, vol. 39, no. 4, pp. 487–502, 2015.