Fault Detection and Diagnosis for Component-based Robotic Systems

Min Yang Jung and Peter Kazanzides

Abstract— **In the software engineering domain, much work has been done for fault detection and diagnosis (FDD) and many methods and technologies have been developed, especially for safety-critical systems. In the meantime, component-based software engineering has emerged and been widely adopted as an effective way to deal with various issues of modern systems such as complexity, scalability, and reusability. Robotics is one of the representative domains where this trend appears. As technology advances, robots are beginning to inhabit the same physical space as humans and this makes the safety issue more important, even critical. However, the safety of recent component-based robotic systems has not been extensively investigated yet. One effective way to achieve system safety is fault tolerance based on FDD which recent robot systems can benefit from. For this purpose, we propose a FDD scheme for component-based software systems with the requirements of flexibility, extendability, and efficiency. The proposed FDD scheme consists of three main components: filter and history buffer, filtering pipeline, and FDD pipeline. We implemented this scheme using the *cisst* framework and show how it can be systematically deployed in an actual system. As an illustrative example, a FDD pipeline is set up to detect a thread scheduling fault on various operating systems (Linux, RTAI, and Xenomai) and experimental results are presented. Although the target of this example is only one type of fault, it demonstrates how the proposed FDD scheme can be introduced to component-based environments in flexible and systematic ways and how system designers can define a fault and FDD pipeline for it. It is obvious that the importance of dependability–especially safety–of robots will significantly increase as robots are deployed in our daily lives, directly operate on us, or interact closely with us. Thus, the FDD scheme proposed in this paper can be a useful basis for robot dependability research in the future.**

I. Introduction

As robot technologies advance, the range and types of tasks that robots can do has been continuously expanding and those tasks often involve humans. Compared to the early generation of robots where their workspace was physically isolated from humans, recent robots are getting closer to humans and their workspace sometimes has significant overlap with that of humans. In the case of medical and surgical robotics, robots even operate inside human bodies and this requires such robot systems to be safety-critical.

Safety is a system issue and thus both hardware and software components should be designed with the consideration of safety throughout all development phases [1]. However, building safe medical and surgical robot systems is not a trivial process because those systems typically require a variety of different devices such as input devices, tracking systems, imaging devices, and robot controllers, which increase the complexity of such systems, making integration and management of devices complicated. Furthermore, this becomes even more challenging if non-functional properties such as safety, reliability, interoperability, or performance are required to be considered together. The issue here is how to analyze, design, deploy, and manage these complex systems. Recently, component-based software engineering techniques have been adopted in the robotics domain to cope with these issues and thus safety needs to be considered within a component-based environment [2].

Fault tolerance is one effective way to achieve system safety. The starting point of fault tolerance is to be able to detect and identify faults in a system. In the software engineering domain, fault detection and diagnosis (FDD) has been extensively studied for traditional safety-critical systems such as nuclear power plant systems, chemical process systems, aircraft control systems, automotive control systems, train control systems, and so on. However, these FDD schemes do not consider the characteristics of component-based software systems which have a hierarchical and layered architecture, although the method and technique itself are applicable to the systems. Thus, we propose a FDD scheme for component-based software systems based on our previous work on systematic fault identification [3]. The proposed FDD scheme is designed such that various signal processing techniques and FDD methods can be implemented in a flexible and systematic way depending on the characteristics of faults of interest and it provides robot designers with run-time FDD capability. As a simple but illustrative example, we show how a fault is defined to detect the irregularity of periodic thread execution and how this fault can be detected and diagnosed on various operating systems (Linux, RTAI, Xenomai) using the proposed scheme. For experiments and implementation, we use the *cisst* libraries [4].

II. Background

A. Fault Detection and Diagnosis

Many FDD techniques and methods have been investigated in the software engineering domain. In the literature, a FDD method consists of three tasks: 1) *fault detection* to determine if something is wrong in the system and any fault occurred, 2) *fault isolation* to identify the location and type of the fault, and 3) *fault identification* to determine the severity of the fault that occurred [5]. Depending on the characteristics of the techniques used, FDD methods can be classified into two categories: model-based methods and data-based methods. Each scheme is further divided into quantitative

Authors are with the Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA. Peter Kazanzides can be contacted at pkaz@jhu.edu.

and qualitative methods [5], [6], [7], [8]. Fig. 1 shows a list of methods in each scheme.

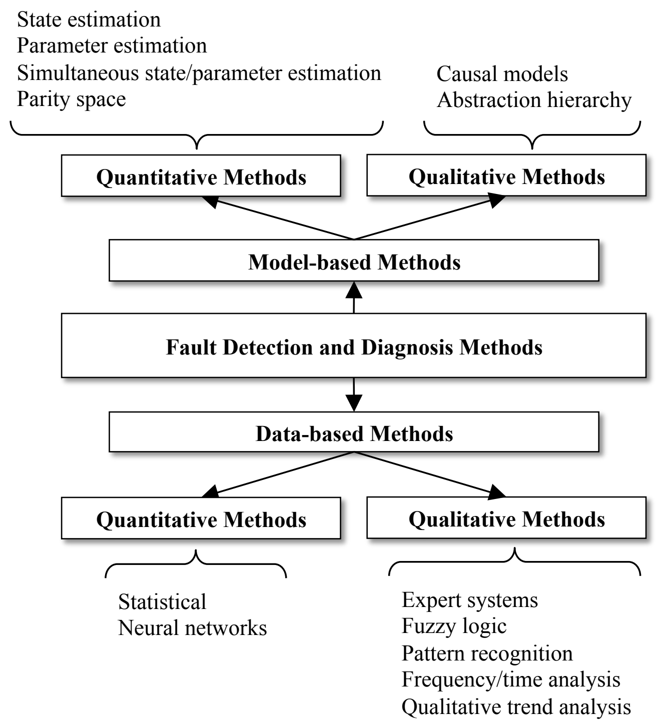


Fig. 1: Classification of FDD methods

B. The *cisst* Component-based Framework

The *cisst* package [4] is a collection of component-based open-source C++ software libraries originally developed for computer-assisted intervention systems [9]. It supports various operating systems such as Microsoft Windows, Linux, real-time Linux variants (RTAI and Xenomai), Mac OS X, and QNX. The *cisst* component model is shown in Fig. 2;

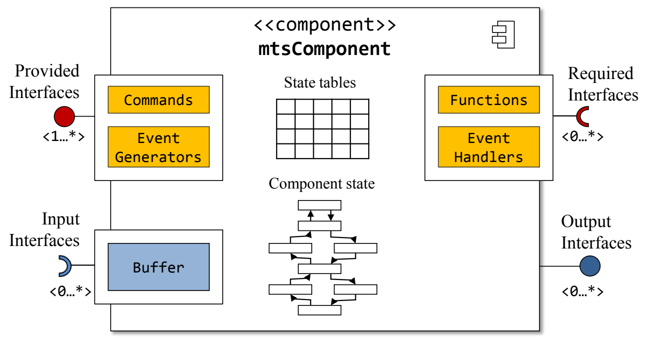


Fig. 2: Overview of *cisst* base component

it includes a list of *provided interfaces*, *required interfaces*, *output interfaces*, and *input interfaces*. Each provided interface can have multiple *command objects* which encapsulate the available *services*, and *event generators* that broadcast events. Each required interface has multiple *function objects* that are bound to command objects and it may also have *event handlers* to respond to events. A *connection* is defined as a pair of a required interface and a provided interface. A set of connections between interfaces determines the overall structure of and information flow within a system. Components can be active (i.e., with their own thread or threads) or passive. Active components can have different execution models, such as periodic execution and signal-based execution.

The *cisst* package supports two different but complementary configurations: (1) a standalone configuration that supports multi-threaded systems in a single process, and (2) a networked configuration that supports not only multi-threaded systems but also multi-process (and multi-host) systems [10]. Fig. 3 shows a distributed architecture of the networked configuration where components can be deployed in multiple processes across networks. Each process has a

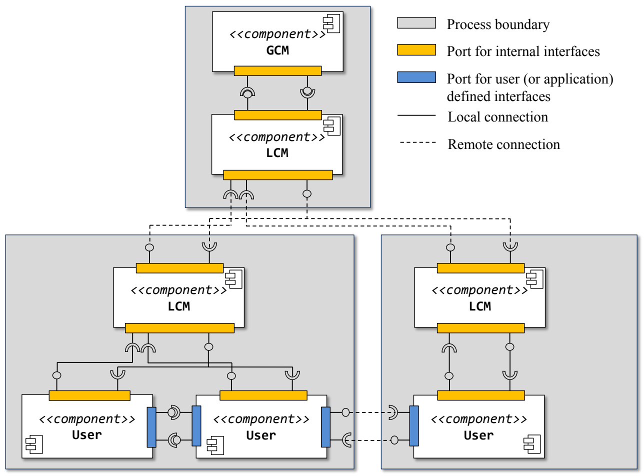


Fig. 3: Component-based architecture of *cisst*: distributed components over a network

local component manager (LCM) that is unique within the process and manages all components in that process. All LCMs connect to the global component manager (GCM). The GCM is a centralized component that maintains, manages, and provides system-wide information such as a list of processes, components, interfaces, and connections. This component-based architecture relies on the component-based data exchange mechanism (i.e., the encapsulation of services) between components. The architecture also allows both standalone and networked configurations to be deployed together to support the optimal performance of multi-threaded components in the same process as well as data exchange between different processes across a network.

III. Fault Detection and Diagnosis for Component-based Software Systems

The FDD methods for traditional safety-critical systems have been extensively studied in the last three decades and there are a variety of different FDD techniques and methodologies [5]. Our work aims to design a FDD scheme for component-based software systems (CBSS) so that existing FDD techniques can be systematically and flexibly introduced and deployed to CBSS, allowing recent component-based robotic systems to benefit from the previous FDD studies. The challenge here is how to support a wide range of FDD methods in systematic and flexible ways within component-based environments considering the inherent characteristics of CBSS. Furthermore, performance-related issues should also be considered together from a practical point of view because the complexity and scale of recent robot systems have significantly increased to perform a wide range and type of tasks. We summarize our design requirements first and then propose a FDD scheme for CBSS.

A. Design Requirements

- **Flexibility**: The FDD scheme must be able to adopt and support a variety of different FDD methods so that appropriate methods required by applications can be systematically deployed in the system. Also system designers should be able to deploy the FDD scheme at any layer of interest, with different granularities, depending on the characteristics of target objects or faults to monitor.
- **Extendability**: The FDD scheme should allow system designers to extend the scheme so that more than one FDD method can be combined or integrated. Support for various deployment options is also desirable.
- **Efficiency**: The FDD scheme should run efficiently such that it generates as minimal run-time overhead as possible on the target object being monitored.

B. FDD Scheme for CBSS

The basic unit of CBSS is a *component* which is defined by a *component model*. A component model defines *semantics* (what components are), *syntax* (how components are defined, constructed, and represented), and *composition* (how components are composed, assembled, or deployed) [11]. Although there is a wide variety of different component models, what is commonly found is a *layered architecture* because the composition and deployment of components within component-based environments is made in a hierarchical manner. Using the *cisst* component model (Fig. 4), our previous work identified the five layers of CBSS which can be applicable to other component models as well with slight structural modifications. Focusing on this hierarchical characteristic, we present a FDD scheme that consists of the filter and history buffer, the filtering pipeline, and the FDD pipeline.

1) Filter and History Buffer: We define two basic elements that are fundamental to our FDD scheme: *filter* and *history buffer*, as shown in Fig 5. A filter is comprised of input, filtering algorithm, and output. An input is read directly from physical sensors or from internal process measurements. Given an input, a filtering algorithm implements an actual filtering method over the input and generates an output. Depending on the filtering algorithm, a filter can have multiple inputs and/or outputs if it requires multiple measurements at the same time or generates multiple outputs. A *history buffer* is a time-indexed buffer that contains a history of data. Data should be accessible in a thread-safe

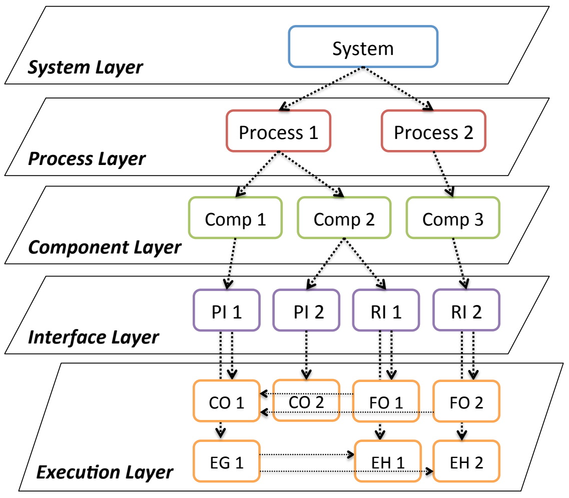
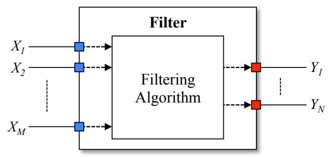
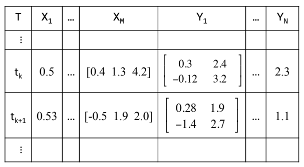


Fig. 4: Layered Architecture of Component-based Software Systems

and efficient way to allow simultaneous access by multiple components. A filter reads its inputs from this buffer and writes its outputs to the buffer.



(a) Filter



T	X_1	...	X_M	Y_1	...	Y_N
⋮						
t_k	0.5	...	[0.4 1.3 4.2]	$\begin{bmatrix} 0.3 & 2.4 \\ -0.12 & 3.2 \end{bmatrix}$	...	2.3
t_{k+1}	0.53	...	[-0.5 1.9 2.0]	$\begin{bmatrix} 0.28 & 1.9 \\ -1.4 & 2.7 \end{bmatrix}$	...	1.1
⋮						

(b) History buffer

Fig. 5: Basic elements for filtering

2) Filtering Pipeline: Filters can be cascaded in series such that outputs from a filter can be inputs to another filter as in Fig. 6. This allows filter desingers to design and implement complex filtering algorithms with a combination of simpler filters. One advantage of this "Lego block"-like approach is that filter designers have access to not only inputs and ouputs of the entire pipeline, but also intermediate inputs and ouputs. This brings more control over the pipeline and debugging capability to application developers. A single monolithic filter can, of course, be built for the entire filtering algorithm if it is desired. Another advantage is that filters can be attached to any layer of the system (vertical distribution) and a filtering pipeline can be split across the system (horizontal distribution).

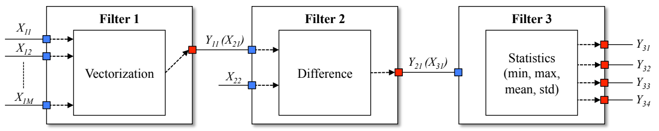


Fig. 6: Example of filter pipeline (cascaded filters, X_{ij}: j-th input to i-th filter and Y_{ij}: j-th output of i-th filter)

3) FDD Pipeline: Three tasks that a FDD scheme performs are fault detection, fault isolation, and fault identification (Sec. II-A). Our FDD scheme implements these tasks as a part of the *FDD pipeline* that has three steps: *monitoring and feature extraction*, *fault detection*, and *fault diagnosis* as in Fig. 7. In the monitoring and feature extraction step,

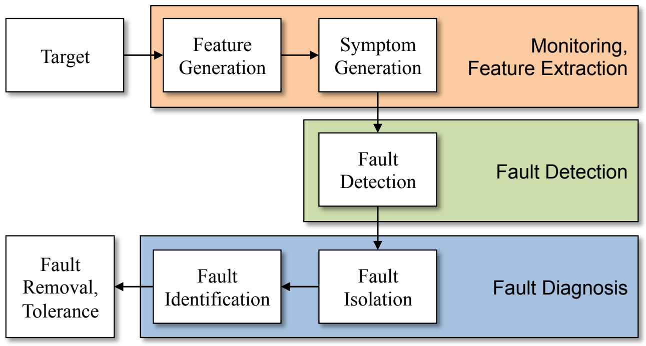


Fig. 7: FDD pipeline

quantities or measurements of interest are sampled from targets being monitored. Targets can be any objects or elements of any layer in the layered architecture of CBSS (Fig. 4). The sampled measurements are fed into filtering pipelines as inputs and the filtering pipelines generate *features* and *symptoms* as outputs. These outputs are then passed to the fault detection step which determines whether or not any fault of interest occurred based on the outputs, and initiates the fault diagnosis procedure, if any. The fault is then isolated and identified and the FDD pipeline ends with comprehensive information about the fault such as type, location, timestamp, and severity. This information provides a useful basis for next-stage schemes such as fault removal or fault tolerance, although they are beyond the scope of this paper.

This FDD pipeline can be defined at two different levels: *framework*-level and *application*-level. At the framework level, a FDD pipeline is automatically and internally set up by a framework to monitor application-independent elements such as thread periodicity, data communication bandwidth of a network channel, or internal buffer utilization. The application-level FDD pipeline is designed and specified by application system designers to monitor application-specific elements.

IV. FDD Framework in *cisst*

As a proof-of-concept, we implemented our FDD scheme for CBSS using *cisst*. In this section, we describe how the FDD scheme is implemented and deployed within the *cisst* component-based environment, while meeting the design requirements summarized in Sec. III-A.

First, we define a base class for filters. It does not have an actual filtering algorithm but has a placeholder for derived filter classes. It also provides APIs to easily define any number of filter inputs and outputs. An actual filter that is derived from this base class implements a filtering algorithm and only needs to specify inputs and ouputs. Together with the "Lego block"-like approach of the filtering pipeline (Sec. III-B.2), this plug-in feature makes it easy to update or replace filtering algorithms and enables the integration of 3rd-party math, statistics, or numerical packages, if necessary. A list of currently defined basic filters includes bypass filter, arithmetic filter, 1st-order differentiator filter, vectorization filter for the conversion of multiple scalar inputs into one vector output, norm filter (L_1, L_2, and L_∞ norm), and exponentially weighted moving average (EWMA) filter [12]. For the history buffer, we use the *state table* [13]. It is basically a time-indexed circular buffer and provides thread-safe, lock-free, and efficient data exchange mechanisms between threads (or components), enabling the filter's efficient data access to/from state tables. Each input and output is associated with a state table. Because the state table maintains the history of registered inputs and outputs of finite length, signal processing or statistical methods that require past data, such as Fast Fourier Transform, can be used as filters.

In each process, a separate monitoring component is defined to monitor all components in that process while minimizing run-time overhead of the FDD pipeline on the target objects. When a measurement is sampled from a target object, the measurement is updated to the state table that the target object is associated with and the target object continues to run. In the meantime, the monitoring component accesses the state table to fetch new measurements and executes the FDD pipeline to identify faults. This separation of data collection and pipeline execution enables lightweight run-time measurements and filtering.

If any fault is detected, an instance of a fault class is created. The fault class has placeholders for comprehensive information about the faults that can typically occur in CBSS. When the fault instance is created, it gets populated with detailed information about the fault such as fault type, location, timestamp, and severity. Then, it is propagated to the GCM using the *cisst* internal communication services, which provides the global knowledge of the system. In this way, the entire system becomes aware of the fault. If a filter requires measurements from other objects across a network or a monitoring component is located in a different process due to its heavy computation requirements, this FDD piepeline can be seamlessly split and distributed across networks using the *cisst* network layer [10].

V. Illustrative Example

When a robot control system runs at a high ($\geq$1 kHz) frequency, there are a number of fundamental and significant issues that must be addressed in order to achieve high-performance control. One of them is thread scheduling latency. How regularly a thread can be executed determines the stability of control algorithms and thus has direct impact on the overall system performance. When it comes to high-performance dynamic systems, this issue becomes significant [14]. Thus, among a set of systematic faults of CBSS [3], we choose the thread periodicity fault of the *Component* layer as an illustrative example and show how the proposed FDD scheme can be used to set up the FDD pipeline to detect the fault.

Fig. 8 shows an overview of the FDD pipeline installed. For a periodic task, an actual period at the beginning of each execution is obtained as the time difference between two consecutive timestamps, $x(t)$ (step A). This is compared to the nominal period value, $x_{nominal}$, and the difference is defined as a *feature* $y(t)$, s.t. $y(t)$=$x(t)$-$x_{nominal}$ (step B). This feature is passed to the thresholding filter, generating a *symptom* $z(t)$, s.t. $z(t)$=1 if $|y(t)| \geq 0.1 * x_{nominal}$ or $z(t)$=0 otherwise (step C). Depending on the frequency of nonzero $z(t)$, a fault is determined to occur or not, i.e., $w(t)$=0 or 1. In our experiments, we set $w(t)$=1 when nonzero $z(t)$ occurs more than once in a second (step D). If the fault occurs, an instance of the fault object is created in the fault isolation stage, which contains placeholders for detailed information about the fault such as the location, type, and timestamp. The instance is then passed to the fault identification stage (step E). The severity of the fault, defined as the number of nonzero $z(t)$ in a second, is added to the fault object instance and now it has the comprehensive information about the fault. This completes the FDD pipeline and the fault is propated to the entire system by broadcasting the fault instance to the GCM.

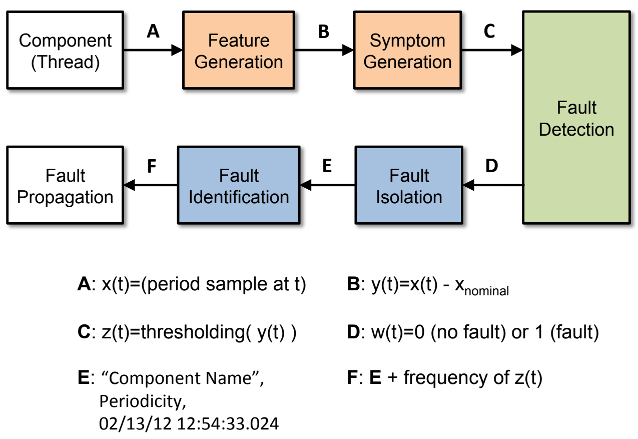


Fig. 8: FDD pipeline to detect thread periodicity fault

To get an idea of how frequently the thread periodicity fault occurs, we added this FDD pipeline to the *cisst* framework and experimentally measured thread scheduling latencies on several OSs. Target OSs include Linux and its real-time variants, RTAI and Xenomai (both 2.6 kernel). The machines used for these experiments are Intel Core Duo 2 3.1 GHz (for RTAI) and Intel Xeon E31275 3.4GHz (for Linux and Xenomai). Periodic tasks with five different frequencies (50, 100, 200, 500, 1k Hz) run for 20 minutes each and $x(t)$ is sampled at 10 Hz, generating a total of 12000 samples per run. This is repeated on Linux, RTAI, and Xenomai. As seen in the statistics and distribution of $y(t)$ (Fig. 9 and Fig. 10, respectively), average deviations were less than 10 μsec under Xenomai and RTAI and no fault was detected, whereas a total of 117 faults occurred under Linux. The "Lego block"-like characteristic of the filtering pipeline allowed us to easily probe and collect intermediate inputs and outputs of the pipeline. Fig. 11 shows these intermediate variables with the final FDD results.

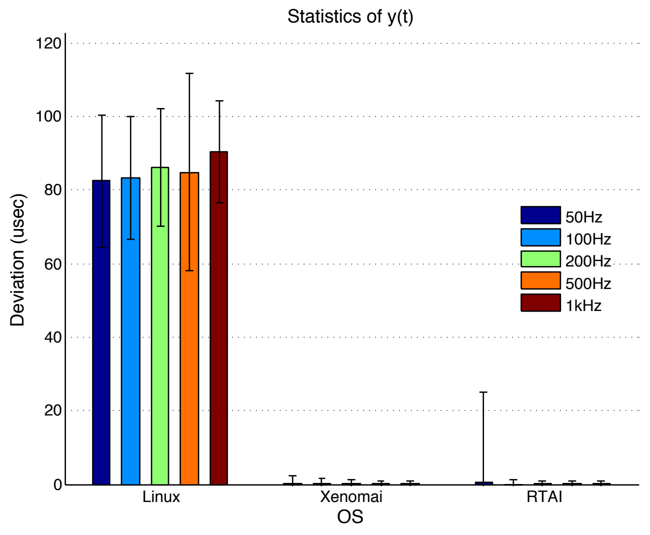


Fig. 9: Statistics of $y(t)$ on different operating systems

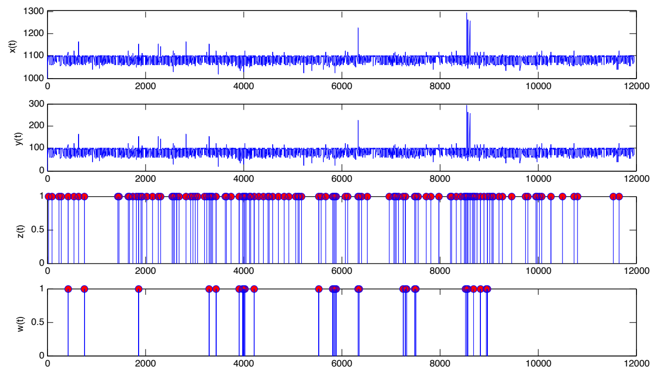
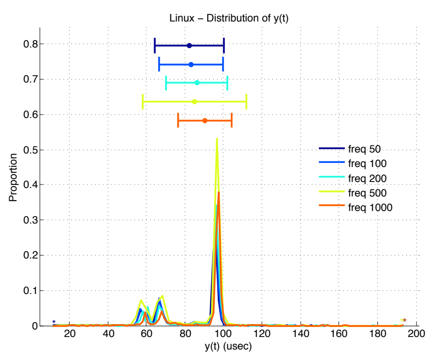


Fig. 11: Intermediate variables of FDD pipeline under Linux

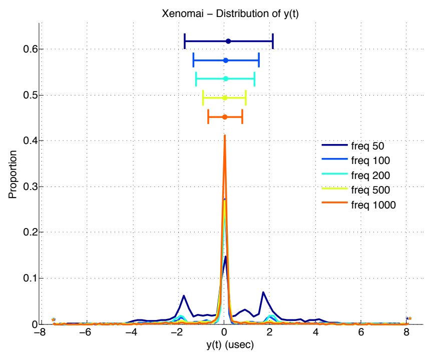
VI. Conclusions and Future Work

We proposed a fault detection and diagnosis scheme for component-based robotic systems. With its three main components (filter and history buffer, filter pipeline, and FDD pipeline), many FDD techniques and methods developed in the software engineering domain can be adopted in a systematic and flexible way. Using *cisst*, we implemented the proposed scheme and demonstrated that it can be deployed with flexibility, extendability, and efficiency. Experimental results with the thread periodicity fault were presented to show how system designers can define their own faults and the FDD pipeline.

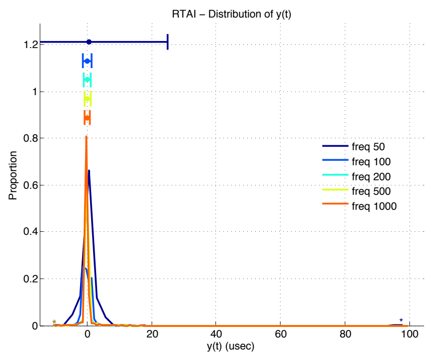
As the complexity and scale of robotic systems increase and robots are getting closer to our daily lives, the dependability issues of component-based robotic systems are becoming more important and getting more attention in the robotics domain. FDD is a fundamental element for fault tolerance, which is one of the effective ways to achieve system safety. The proposed FDD scheme provides tools for robot designers to systematically and flexibly deploy various FDD methods into component-based robotic systems and this will be a useful basis for dependability research in the



(a) Linux



(b) Xenomai



(c) RTAI

Fig. 10: Distribution of $y(t)$ on different operating systems

robotics domain.

The proposed FDD scheme has been applied to a framework-level FDD but can be easily and consistently extended to the application level as well. Our next goals include the extension of the FDD scheme so that it covers a wider range of faults and this will allow us to systematically deal with fault removal and fault tolerance. With these components available, the safety issues of robotic systems can be approached in more effective ways in the future.

References

[1] W. Dunn, "Designing Safety-critical Computer Systems," *IEEE Computer*, vol. 36, no. 11, pp. 40–46, Nov 2003.

[2] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering (Part I)," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, Dec 2009.

[3] M. Y. Jung, "A layered approach for identifying systematic faults of component-based software systems," in *Proceedings of the 16th International Workshop on Component-Oriented Programming*. New York, NY, USA: ACM, 2011, pp. 17–24.

[4] The *cisst* libraries. [Online]. Available: https://trac.lcsr.jhu.edu/cisst

[5] Y. Zhang and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," *Annual Reviews in Control*, vol. 32, no. 2, pp. 229–252, 2008.

[6] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. Kavuri, "A review of process fault detection and diagnosis, Part I: Quantitative model-based methods," *Computers & chemical engineering*, vol. 27, no. 3, pp. 293–311, 2003.

[7] V. Venkatasubramanian, R. Rengaswamy, and S. Kavuri, "A review of process fault detection and diagnosis, Part II: Qualitative models and search strategies," *Computers & Chemical Engineering*, vol. 27, no. 3, pp. 313–326, 2003.

[8] V. Venkatasubramanian, R. Rengaswamy, S. Kavuri, and K. Yin, "A review of process fault detection and diagnosis, Part III: Process history based methods," *Computers & Chemical Engineering*, vol. 27, no. 3, pp. 327–346, 2003.

[9] A. Kapoor, A. Deguet, and P. Kazanzides, "Software components and frameworks for medical robot control," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2006, pp. 3813–3818.

[10] M. Y. Jung, A. Deguet, and P. Kazanzides, "A component-based architecture for flexible integration of robotic systems," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010.

[11] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Software Eng.*, vol. 33, pp. 709–724, 2007.

[12] J. Lucas and M. Saccucci, "Exponentially Weighted Moving Average Control Schemes: Properties and Enhancements," *Technometrics*, vol. 32, no. 1, pp. 1–12, Feb. 1990.

[13] P. Kazanzides, A. Deguet, and A. Kapoor, "An architecture for safe and efficient multi-threaded robot software," in *IEEE Conf. on Technologies for Practical Robot Applications (TePRA)*, Nov 2008, pp. 89–93.

[14] P. Corke and M. Good, "Dynamic effects in visual closed-loop systems," *IEEE Trans. on Robotics and Automation*, vol. 12, no. 5, pp. 671–683, Oct. 1996.