Research on Runtime Environment of Spacecraft Testing System

Zhongwen Li
State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
Email: lizhongw@nlsde.buaa.edu.cn

Xianjun Li, Gang Ye, Wentao Yao
State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
Email: { lixianjun, yegang, yaowentao}@nlsde.buaa.edu.cn

***Abstract*—Spacecraft testing is an important phase of developing a spacecraft. In order to automate the testing process in parallel pattern, this paper presented the runtime environment architecture of testing system based on task-collaborating. In this architecture, there are three challenges: task-collaborating, task executing, resource management. Through analyzing the interference pattern between parallel test tasks, this paper proposed a collaborative strategy between parallel test tasks according to conflict detection. Then the structure of executing system and resource management platform were introduced. In prototype system, the experimental result proved effectiveness of the scheme presented in this paper.**

***Index Terms*—runtime environment, parallel testing, spacecraft, task collaboration**

I. INTRODUCTION

Manufacturers build satellites to comply with a set of requirements specified by the customer. Extensive tests are performed as part of the construction process of any satellite. These tests are carried out on both hardware and software at all stages of manufacture and delivery: equipment; subsystem; system; on-ground; and in-orbit [1]. To meet the testing requirements of spacecraft for different purposes, shorten the development cycle, improve reusability, reduce costs, ensure reliability, reduce or even eliminate the error rate. At present the space powers have been or are developing generic spacecraft testing systems, in order to improve spacecraft testing efficiency. Among them, the issue of parallel test in spacecraft testing system is a very important problem and needed to be resolved.

Given the current situation of spacecraft testing, this paper proposed architecture of task-collaborating parallel testing system, through analyzing interference pattern between parallel test tasks, proposed the collaborative strategy of parallel test tasks. After that, the executing system and resource management platform were introduced.

II. RELATED WORK

In traditional testing system, computers and test instruments are idle for more than 50% of the time, and the whole system's testing efficiency is relatively low [2]. Parallel Test Technology is a new technology developed from automatic test system (Automatic Test System, ATS) in the trend of further reducing testing time and cost. Parallel test technology improves system throughput by increasing the number of items being tested in unit of time and improves instrument utilization by reducing the idle time of instrument and the CPU [3].

To avoid interaction between parallel test tasks, the current research of parallel testing realization are more concentrated on dividing the test tasks into groups. Literature [3] proposed a scheduling algorithm theory of parallel test tasks based on graph coloring, using the "graph" to describe the instrument resources that are occupied by test tasks, and thereby achieved test tasks grouping. Literature [4], according to relationship between test tasks and instrument resource, realized the automatically generated sequence of parallel test tasks by state map.

During the procedure of spacecraft integrated testing, partially because of complexity of spacecraft, one test program is often required to operate many computers and components of spacecraft [5]. If we conduct multi-task parallel testing to the same spacecraft, may cause uncertainty state of the computers or components of the spacecraft. Therefore, when testing a spacecraft, it is impossible to directly group the test tasks and parallel test tasks within each task group [6]. Therefore, how to analyze the running status, so as to determine partial independence of test programs and finally realize parallel testing between the independent parts of test programs became a key point. While spacecraft test cases are multiple and complex, it is almost impossible for the testers to analyze test programs one by one and determine the internal mutual affiliations. Therefore, current spacecraft automated testing systems are more using traditional sequential testing method. Given the characteristics of spacecraft, how to conduct its parallel testing and how to build the runtime environment need to be further studied.

III. ARCHITECTURE OF SPACECRAFT TEST SYSTEM

doi:10.4304/jcp.6.4.657-663

As show in Fig. 1, to achieve parallel testing, architecturally, we divide the runtime environment of spacecraft automated test system into three levels, and main functions of each level are as follows.

The first layer, test resource layer, is used for deploying device server which is related to subsystem devices. It has connected to general or special test devices, managing and operating the underlying device according to commands and instructions sent over from the upper layer, and transmitting spacecraft test data information and command execution information collected by special test devices to the upper layer.

The second layer is test support layer, which is for deploying collaboration server, master control server and database server for parallel testing. Parallel test collaboration server is responsible for collaboration of the testing executing clients of upper layer by using collaborative strategies and conflict detection mechanism, which enables parallel testing of the spacecraft. The master control server is mainly used for providing services of forwarding test data and test commands provided by resource layer, while completing automatic storage of test data and monitoring the testing process. All the test application layer software only need to connect to the master control server to send remote control commands, receive telemetry data to control and manage test devices.

The third layer is application layer, in which the executing system clients for spacecraft automated testing lies in, and it automates test scripts executing for automate testing.

The follow-up chapters of this paper mainly discuss the collaboration strategy that running in the collaboration server, the executing system and the device management framework deployed in the device server.

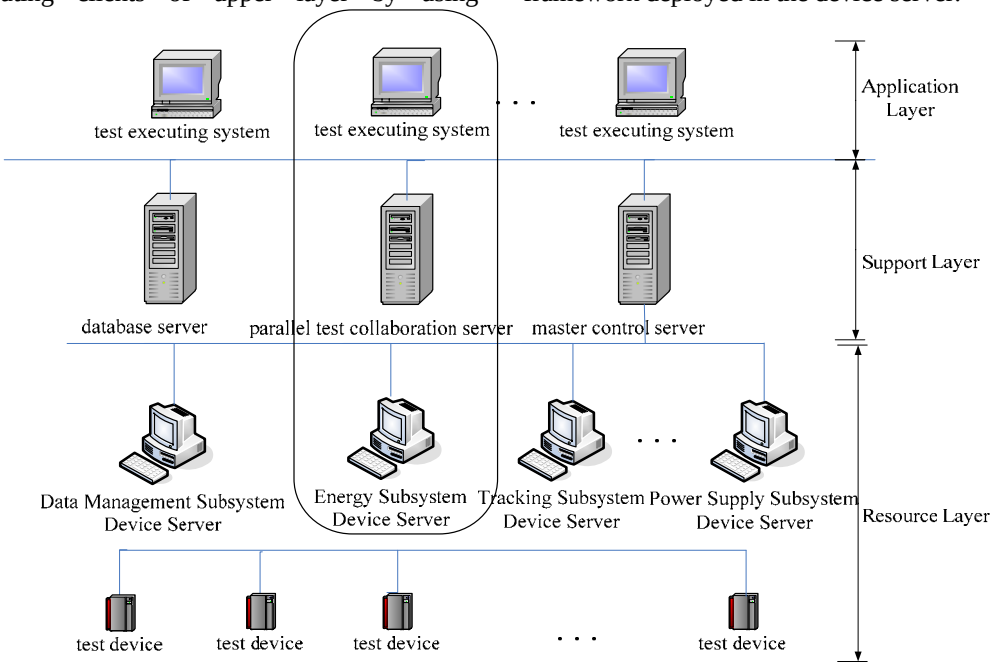

Figure 1. Architecture of spacecraft testing system

IV. COLLABORATIVE STRATEGY

In the process of spacecraft testing, the parallel test task will have a mutual interference, because different instructions may lead the same parameter to different changing, so when the test task judge the correctness of this parameter, it may get the wrong result. Fig. 2 gives a simple scenario of the mutual interference.

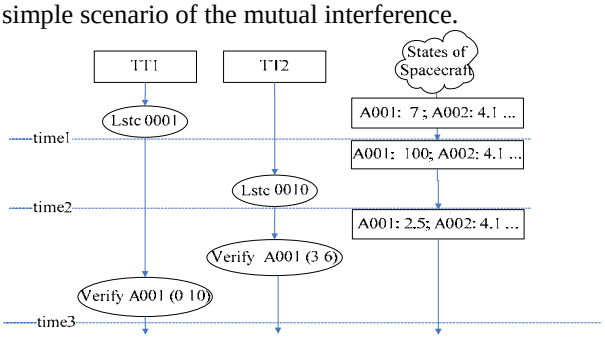

Figure 2. A simple scenario of the mutual interference

"TT1" and "TT2" represent two parallel test tasks. "Lstc" represents sending a scatter instruction to the spacecraft, and the number such as "0001" is the parameter of the "Lstc" instruction. The command "Verify" is to check the correctness of one state parameter of spacecraft, this command has two parameters, one is the code of parameter such as "A001", and the other is the upper and lower value this parameter should be. In Initial state, the "A001" state parameter of spacecraft with its value of 7, at the moment of "time1", test task of "TT1" sends a "Lstc" instruction to spacecraft, as result of this instruction, the value of "A001" should be between 0 and 10. But after that the value of "A001" changes to 100, in the expectation situation the "TT1" will found this error state of spacecraft with command of "Verify", however before "TT1" do that command, "TT2" send another "Lstc" instruction at the moment of "time2" to change the value of "A001" to 2.5. When "TT1" do the "Verify"

command, it lost the value of 100, it will get the wrong result obviously.

A. Conflict Detection

In order to conduct collaborative parallel testing of each test executing system, we need to define the conflict between test tasks first.

Definition 1 spacecraft entity

$SE := (ID_{SE}, ST, SV)$, in which is the unique identifier of spacecraft being tested, $ST = (st_1, st_2, ..., st_n)$ stands for status collection of tested spacecraft entity, and $SV = (sv_1, sv_2, ..., sv_n)$ is for the status value of tested spacecraft entity.

Definition 2 spacecraft test operation

$SM := (ID_{SM}, ID_{SE}, OPitem, OPtype, OPattr, OPts, OPte)$, in which is the unique identifier of operations that test program acting on spacecraft entity *SE*, ID_{SE} is the identifier of the *SE*, *OPitem* is name of the operation, *OPtype* is type of the operation, including read and modify, *OPattr* is list of all the spacecraft entity status related to operation *SM* on entities *SE*, *OPts* is the beginning time operation *SM* binds *OPattr*; *OPte* is the time of operation *SM* releasing the bound with *OPattr*.

Definition 3 spacecraft test operation conflict

1. $SM_i \bullet ID_{SE} = SM_j \bullet ID_{SE}$ (1)
2. $SM_i \bullet OPtype = SM_j \bullet OPtype$, (2)

In which $OPtype$ is modify operation

3. $SM_1 \bullet OPattr \cap SM_2 \bullet OPattr \neq \varnothing$ (3)
4. $SM_1 \bullet (OPts, OPte) \cap SM_2 \bullet (OPts, OPte) \neq \varnothing$ (4)

If meets the above four points, conflict exists between SM_i and SM_j, denoted by $SM_i \otimes SM_j$, on the contrary, if there is no conflict between SM_i and SM_j, denoted by $SM_i \oplus SM_j$.

Based on the above definition of spacecraft test operation, the detection operation between SM is given as follows for detecting conflicts:

$$\begin{aligned} &chk(SM_i, SM_j) := \\ &\{SM_i.Opattr \cap SM_j.Opattr \neq \varnothing \Rightarrow SM_i \otimes SM_j; \\ &SM_i.Opattr \cap SM_j.Opattr = \varnothing \Rightarrow SMi \oplus SM_j\} \end{aligned} \quad (5)$$

Make

$$set(SM) = \{SM_0..., SM_i \mid i = 0,1,...\} \quad (6)$$

collection of test operations *SM*.

Detection operation extended to two sets' detecting:

$$\begin{aligned} &chk(set(SM)_i, set(SM)_j) := \\ &\{\forall SM \in set(SM)_i, \exists SM_j \in set(SM)_j, SM \otimes SM_j \Rightarrow \\ &set(SM)_i \otimes set(SM)_j; \\ &\forall SM \in set(SM)_i, \forall SM_j \in set(SM)_j, SM \oplus SM_j \Rightarrow \\ &set(SM)_i \oplus set(SM)_j\} \end{aligned} \quad (7)$$

Detecting between the two sets above has given the method for detecting conflict between multiple test task programs in parallel test.

B Collaborative Strategy

Collaboration server collaborates testing between multiple executing systems based on above conflict detection, and the collaborative strategy is constituted by the following logic service components:

1) Test task (TT): Representing for all the test tasks of parallel testing. Test tasks itself are not part of collaborative strategies, but as service entities, collaborative process needs its participation.

2) Test task collaboration agent (TCA): Each test task will correspond to a collaborative agent instance of test task, and each test task communicates with the agent to achieve collaboration.

3) Time process service (TPS): Time process component maintains timing for each test task request, for time-out mechanism of request.

4) Command applying and waiting set service (CAWS): It maintains a list of command applications sent by each test task which is conflict through detecting.

5) Conflict arbitration service (CAS): The conflict arbitration agent will do conflict detecting of each command sending application.

6) Command binding collection service (CBCS): It maintains a list of all the commands which each test task has bound to, which did not conflict after detection.

Definition 4 message definition

Apply: Test task's application of instruction sending, it contains ID of test task, the code of instruction and the time of timeout, with form of $< Apply, TT_{ID}, InsID, Time >$.

Accept: Accept message represents test task can send its instruction: this message contains the code of instruction, with form of $< Accept, InsID >$.

Timeout: This message tells the related service component a timeout event happened, with form of $< Timeout, InsID >$.

Release: Test task release its occupation of an instruction, it contains the code of instruction, with form of $< \text{Re}lease, InsID >$.

InsInfo: Information of instruction, it contains the code of instruction and the ID of its test task, with form of $< Accept, InsID >$.

Refusa: Refusal message represents an instruction can not be sent. It contains the code of instruction, with form of $< \text{Re}\, fusal, InsID >$.

Time: The time of timeout, with form of $< TimeInfo, InsID, Time >$.

DelInsInfo: This message represents the instruction was sent, so this instruction should be deleted from the sending list of instruction. It contains the ID of test task and the code of instruction, with form of $< DelInsInfo, TT_{ID}, InsID >$.

DeleteIns: This message represent an instruction should be deleted from waiting list as a timeout event was happened. It contains the ID of test task and the code of instruction, with form of $< DeleteIns, TT_{ID}, InsID >$.

ReqWaiInsInfo: Request for all the information of instructions in the waiting list, with form of $<\text{ReqWaiInsInfo}>$.

WaiInsInfo: All the information of instructions in the waiting list, it contains a set of the instructions, with form of $<\text{WaiInsInfo}, \textit{InsIDSet}>$.

RequesInfo: Request for all the information of instructions in the occupation list, with form of $<\text{RequesInfo}>$.

RegInfo:All the information of instructions in the occupation list, it contains a set of the instructions, with form of $<\text{Re}\,\textit{gInfo}, \textit{InsIDSet}>$.

DeledRegInfo: This message will be sent to CAS when test task release the occupation of an instruction. Its form is $<\text{DeledRegInfo}, \textit{InsIDSet}>$.

In Fig. 3, collaborative process of parallel test tasks is as follows:

Step 1. TT sends message <Apply> to TCA.

Step 2. TCA constructs message <InsInfo> and sends it to CAS according to message <Apply>.

Step 3. CAS requests to CBCS for information about every instructions that are being executed, and conducts conflict detection according to the return <RegInfo> information.

Step 4. CAS test, if there is no conflict goto Step 5, else goto Step 6.

Step 5. CAS send registration information <InsInfo> to CBCS, at the same time sending message <Accept> to TCA, and TCA sending <Accept> message to TT, and command is send by TT.

Step 6. CAS send <Refusal> message to TCA, to refuse command sending. When the waiting time of TT reaches timeout, TCA inform TT timeout by <Timeout>, and then send message <DeleteIns> to CAWS, to remove timeout command from the waiting list.

Step 7. TT sends message <Release> to TCA to release acquisition of the command.

Step 8. TCA sends message <DelInsInfo> to CBCS to remove specified command information registered by TT from all the registered commands, and passes <DeledRegInfo> information to CAS after releasing.

Step 9. CAS receives <DeledRegInfo>, it sends a request <ReqWaitInsInfo> to CAWS to obtain waiting list, then CAWS will send message <WaitInsInfo> to CAS, then CAS picks out all the commands which are not conflict according to <WaitInsInfo> and <DeledRegInfo> and then sends <Accept> to the appropriate TCA.

Step 10. TCA sends message <Accept> to TT. Meanwhile send <InsInfo> to CBCS for registration and TPS sends the command.

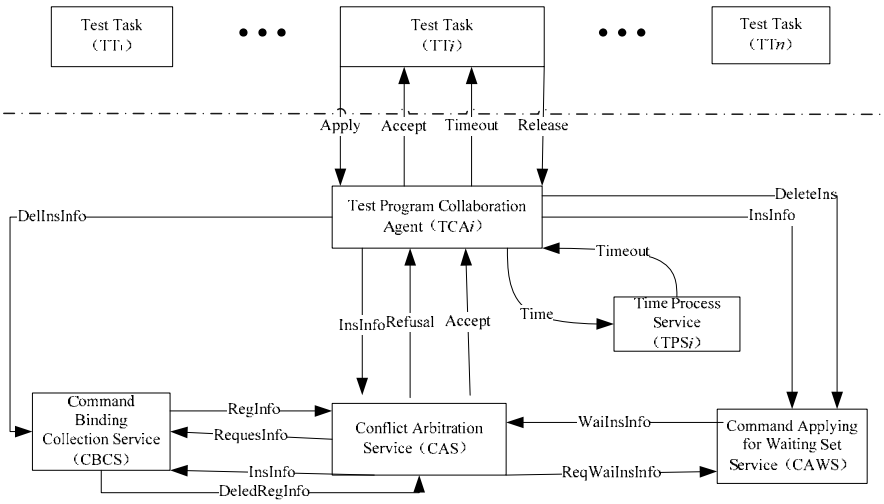

Figure 3. Logic diagram of parallel test collaboration

V. STRUCTURE OF TEST EXECUTING SYSTEM

During spacecraft testing, testers write test scripts under testing requirements, and run the test scripts to achieve automated testing on the spacecraft. Here is the composition of spacecraft executing system.

Fig. 4 is the structure diagram of test executing system .Parallel test execution system of spacecraft is divided into five parts: execution engine interface, execution engine, collision detection, execution monitoring and exception handling. The functions of each part are:

Execution Engine interface: execution engine interface is a definition of spacecraft parallel testing execution systems' interface with the outside world. This section is mainly responsible for receiving test program from the outside world, and transmitting it to the test task management module in execution engine; receiving the user control request from outside world and forwarding the request to the task control and management module of the execution engine.

Execution Engine: execution engine schedules a number of test programs of execution tasks received from the interface layer, and interpretively execute the various test programs for spacecraft testing. The execution engine instance is divided into three modules, namely the test task management module, the test program interpretive execution module and task control management module, and the modules' functions are:

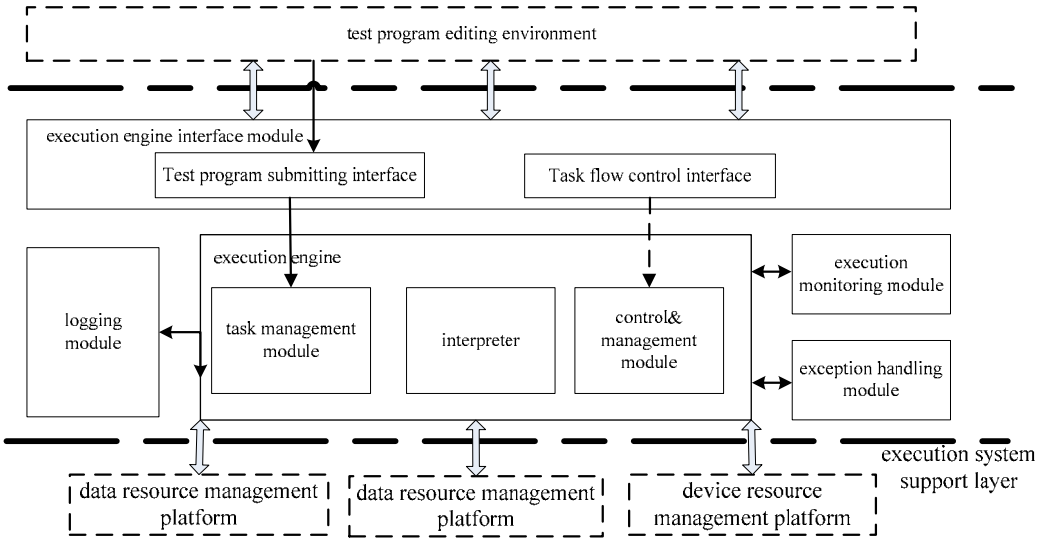

Figure 4. Structure diagram of test executing system

- *Test task management module*: Test task management module is responsible for adding test tasks into task queue, realizing task scheduling according to certain scheduling strategy, and transmitting test programs to interpretive actuator for interpretive execution.
- *Interpreter*: Test program interpretive actuator realizes test programs' interpretive execution by invoking interface provided by data resource management platform and device resource management platform, at the same time achieving collaborative working with the other testing executing clients by communicating with the collaboration server. When the test program is executed, the results of execution will be transmitted to test task management module.
- *Task control and management module*: Task control and management module is used for receiving task control commands (task control commands requested by user and task control commands in exception handling), real-time changing the status of executing task and saving information of task executing environment when the task status is changed.

Logging: Responsible for recording all the key information in test execution process, for analysis after completion.

Execution monitoring: execute monitoring is for monitoring the status of parallel task execution engine and task flow execution, as well as presenting monitor results to testers.

Exception handling: Exception handling module handles various run exceptions of test tasks.

VI. Structure of Testing Resource Platform

The structure of testing resource platform consists of three layers, as shown in Fig. 5, including resource connection layer, resource schedule layer and interface layer. Resource connection layer which locates at the bottom is the foundation of the platform. Resource schedule layer which is responsible for processing and scheduling resource request is the core of the platform. Interface layer provides resource access interface to applications.

Resource connection layer is the basis of testing resource platform. It is responsible for configuration parsing, registration and communication. This layer consists of three modules: configuration parse module, resource driver module and resource index module.

Configuration parse module is responsible for retrieval description information of resources from configuration file which uses XML encoding format. This module uses a popular XML parser-JDOM to parse the configuration file and then retrieve resource information which includes name, type, the name of driver and connection parameters of each resource. At last driver instances are initialized and registered into resource index module along with other resource information.

Resource driver module is a collection of driver instances running in testing resource platform. Because every resource has its own communication protocol and operations, so it's necessary to provide appropriate driver for each kind of testing resource. The driver encapsulates communication detail of the resource, implements data exchange with resource and provides a set of APIs to access resource.

Resource index module which is also called resource directory is the place where resource information and state are stored.

The task of resource schedule layer is to schedule requests for exclusive resources and allocating these resources to upper layer application in multi-thread environment. This layer which is the core of testing resource platform consists of two modules: request queue module and request schedule module.

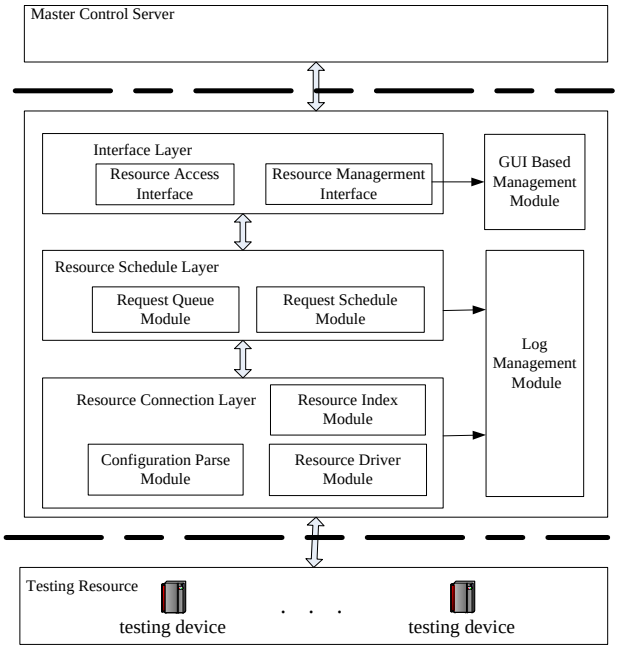

Figure 5. Structure diagram of resource platform

To each exclusive resource registered in the platform, a request queue is allocated for saving the requests for this resource. Request queue module maintains all of these request queues. If a resource is occupied by an application thread, requests from other threads for this resource will be added to its queue. When the resource is released, one of the requests in corresponding queue will be selected. The resource will be allocated to the request's owner thread which will continue to run.

Request schedule module schedules resource requests from upper layer application. When an application thread requests for a resource access, request schedule module will look up the resource information from resource connection layer firstly. If the resource is a shared one, its driver instance will be allocated to the thread. If not, request schedule module will check resource's state. If the state is free, request schedule module will allocate corresponding driver instance to the application thread and set resource's state to busy. And if current state of requested resource is busy, in other words it is occupied by another thread, request schedule module will add the request to corresponding queue and block the application thread. When application thread releases a resource, request schedule module will choose a request from the queue according to a schedule strategy, allocate the resource to request's owner thread and make this thread continue to run. Request schedule module adopts a FCFS (First Come First Service) strategy currently [7].

Interface Layer is at the top of spacecraft testing resource platform. The major effect of this layer is to provide a set of standard interface for upper applications to use this platform. According to its functions, this layer can be divided into resource access interface and resource management interface.

Resource Access Interface is to provide a set of interfaces for upper applications to request and operate resources. In this paper, we abstract each resource to be a service. Also, we represent each request of testing resource as request of corresponding service.

Resource Management Interface provides a set of functions to manage testing resources, including information querying, state querying, registration and logoff, resource request queue state monitor etc. Resource Management Interface mainly applies in the following Graphic Management Module.

VII. EXPERIMENTAL EVALUATION

We did experiment and verification in the prototype system which is implemented based on the above discussion. As follows:

Randomly selected 4 programs from 32 thermal control subsystem test programs, randomly selected 5 programs from 46 data management subsystem test programs, randomly selected 1 program from 4 power distribution subsystem test programs, and to the above-mentioned three spacecraft subsystem test programs, firstly perform sequence simulation running, then use parallel testing method to perform simulation running (using five executing clients for parallel testing), then the 10 test programs selected from three sub-systems are presented to executing system for simulation testing.

All devices include spacecraft were substituted by simulator. This experiment does not consider the time of waiting for test device that is required in the real experiment environment. The experiment repeats 5 times; and each time select 10 test programs from the three subsystems by the same proportion.

All the test results of parallel testing pattern are the same as the sequence pattern. And the time cost is show in Fig. 6.

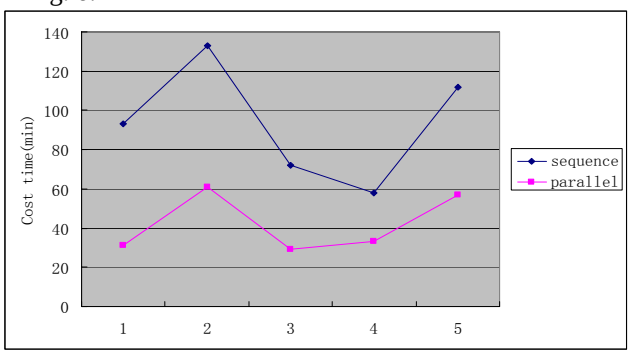

Figure 6. Time costs of two testing patterns

It can be seen from the above experiments that after conflict detection and task collaboration, parallel testing can get same conclusion with sequential testing, moreover, parallel testing can effectively shorten total test time, increase test efficiency. From the experiments, we also found that when the parallel executing test tasks belong to the same subsystem, the time spent is longer than tasks belong to different subsystems, it is because when testing the same subsystem, the probability that commands issued by test tasks are conflict is far greater than test tasks of different subsystems, so test tasks will spent more time on waiting. Therefore, to achieve better

parallelism, we should test as more as possible different spacecraft subsystems at the same time.

VIII. CONCLUSION

In order to automate the testing process in parallel pattern, this paper presented the runtime environment architecture of testing system based on task-collaborating. In this architecture, there are three challenges: task-collaborating, task executing, resource management. Through analyzing the interference pattern between parallel test tasks, this paper proposed a collaborative strategy between parallel test tasks according to conflict detection. Then the structure of executing system and resource management platform were introduced. In prototype system, some experiments were done, and result proved effectiveness of the scheme proposed in this paper. It preliminarily settled the problem of spacecraft parallel automatic test.

REFERENCES

[1] Cater S.J., Ahn D.B., Quigley D., "Automation of satellite requirement verification," *Aerospace Conference, 2006 IEEE*, "doi:10.1109/AERO.2006.1656150"

[2] Anderson J L Jr. High performance missile testing [A]. AutotestCon Proceedings [C]. IEEE, 2003 , pp: 19 – 27, "doi: 10.1109/AUTEST.2003.1243549"

[3] Li Xin, Shen Shituan, Lu Hui. Algorithms of tasks scheduling in parallel test based on graph coloring theory Journal of Beijing University of Aeronautics and Astronautics , 2007, pp:1068-1071.

[4] Ji ajing Zhuo; Chen Meng; Minghu Zou; A Task Scheduling Algorithm of Single Processor Parallel Test System. Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on Vol 1, pp: 627 – 632, "doi: 10.1109/SNPD.2007.383"

[5] T.Yamada, Generic Software for Spacecraft Testing and Operations Based on Spacecraft Model, American Institute of Aeronautics and Astronautics SpaceOps 2006 Conference 2006.

[6] Wang Qingcheng. Electrical Test Technology of Spacecraft [M]. Beijing: Press of Science and Technology of China, 2007.

[7] ZHANG Shu-dong, CAO Yuan-da, LIAO Le-jian. Job scheduling Algorithm Based on Credit Model in Cluster Environment[J]. Small scaled microcomputer system, 2005.12 Vol.26 No.12

Zhongwen Li (1981 -) is currently PhD candidate in State Key Laboratory of Software Development Environment, Beihang University, Beijing, China.

His research interests include workflow, spacecraft automatic testing, and real time system.

Xianjun Li (1977 -) is currently PhD candidate in State Key Laboratory of Software Development Environment, Beihang University, Beijing, China.

His research interests include spacecraft automatic testing, spacecraft integrate test information Processing, component technology.

Gang Ye (1982 -) is currently PhD candidate in State Key Laboratory of Software Development Environment, Beihang University, Beijing, China.

His research interests include spacecraft automatic testing, spacecraft integrate test, software testing，fault localization.

Wentao Yao (1984 -) is currently PhD candidate in State Key Laboratory of Software Development Environment, Beihang University, Beijing, China.

His research interests include software testing, fault localization.