AGENT-ORIENTED FAULT DETECTION, ISOLATION AND RECOVERY

AND

ASPECT-ORIENTED PLUG-AND-PLAY TRACKING MECHANISM

A Thesis

by

FEILONG CHEN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2003

Major Subject: Computer Engineering

AGENT-ORIENTED FAULT DETECTION, ISOLATION AND RECOVERY AND ASPECT-ORIENTED PLUG-AND-PLAY TRACKING MECHANISM

A Thesis

by

FEILONG CHEN

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

August 2003

Major Subject: Computer Engineering

ABSTRACT

Agent-Oriented Fault Detection, Isolation and Recovery and Aspect-Oriented Plug-and-Play Tracking Mechanism. (August 2003)

Feilong Chen, B.S., China Agricultural University

Chair of Advisory Committee: Prof. Richard Volz

Fault detection, isolation, and recovery are some of the most critical activities in which astronauts and flight controllers participate. Recent systems to perform the FDIR activity lack portability and extensibility, and do not provide any explanation of the system's activity. In this research, we explore the use of an agent-oriented paradigm and Java technology for better performance of FDIR activity. Also, we have explored the use of explanation in agent-oriented systems, and designed a system-activity tracking mechanism that helps the user to understand the agents' behavior. We have explored different ways to generalize this mechanism for arbitrary agent systems to use. Furthermore, we studied mechanisms to automatically add the tracking mechanism to an existing agent system. By using AspectJ, an aspect-oriented tool, a plug-and-playable tracking system has been built that can add the capability to track the activity of the system to any JACK agent system easily. Our experience can help further research on using aspect-oriented tools with agent-oriented paradigms together to obtain better performance.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

Fault detection, isolation, and recovery (FDIR) are some of the most critical activities in which astronauts and flight controllers participate. Researchers have been investigating automatic tools for training and assisting in these activities for some time. The Procedural Reasoning System (PRS) [1, 2, 3] and Distributed Multi-Agent Reasoning System(dMARS) have been used to developed FDIR applications. However, applications of both PRS and dMARS lack platform-independence and neither provides explanation of system behavior.

Developing an explanation component [4] to help a user understand what the agent system is doing is essential for two reasons. First, doing so makes the agent system usable as a training aid. Second, understanding what the agent system is doing is an important part of building human confidence in the system as an operational assistant and for recognizing when the agent system has been unable to take all factors into account. Thus, we explored different mechanisms to incorporate an explanation component into an agent system, and have designed a tracking mechanism that allows observation of what the agent system is doing and how it arrives at its decisions.

Our objectives in this research are to study the use of agent-oriented technique for building a highly portable, explanation-based system that is easily extensible and to explore approaches to provide effective explanation of the agent system's behavior. Agent-Oriented Programming techniques are utilized to get better extensibility, and an

This thesis follows the style of *IEEE Transactions on Systems, Man, and Cybernetics.*

effective explanation component, we call *system activity tracking*, has been designed to let the users observe the system activities. Furthermore, we will find a way to generalize the *system activity tracking* mechanism so that it can be applied to an arbitrary system and a way to automatically add the tracking mechanism to an arbitrary system. As a vehicle for developing the ideas and demonstrating the results, we have chosen the Space Shuttle Diagnostic domain.

As an initial approach, this thesis explores how to create a tracking system and its relation to general agent oriented systems. Subsequently, we examine different ways to separate the incorporation of the tracking mechanism from the development of the agent-oriented system being tracked. In particular, we create mechanisms that allow observation of system activity to be obtained automatically, with minimal developer input. We first approach this by developing a preprocessing tool that reads the agent, event, and plan code, and inserts the hooks to the tracking system automatifcally [5]. Doing so changes a program from a single tracking system to a tool for building a tracking system for arbitrary agent systems.

Then, in the next phase, we explore an even simpler way to implement the tracking system independently of the specific agent system, the use of Aspect-Oriented Programming. A open software tool, AspectJ is used to fulfill this mission. AspectJ enables clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, and multi-object protocols. AspectJ is a tool for Java. However, we use JACK, an Agent-Oriented Programming software to ease the development the FDIR system, and AspectJ cannot be applied directly to JACK programs. We describe the techniques we have developed to integrate AspectJ with JACK programs.

In section 2, we briefly describe the background for the work described here, including the **B**elief **D**esire **I**ntention (BDI) model, two predecessor systems, PRS and dMARS, JACK and AspectJ. In subsequent sections, we describe the design and implementation of our agent-based FDIR system, the tracking system, and how the two systems interact; two different approaches are described after that. Examples are shown.

2. BACKGROUND

In this section, we describe background information of space shuttle FDIR, explanation-based systems, agent-oriented and aspect-oriented programming paradigms.

2.1. Space Shuttle Fault Detection, Isolation and Recovery

Researchers have been investigating automatic tools for training and assisting in the FDIR activities for some time. As a test domain in which to conduct our studies, we have used the Reaction Control System (RCS) of the Space Shuttle. The RCS provides propulsive forces from a collection of jet thrusters to control the attitude of the spacecraft [1]. There are three RCS modules, two aft and one forward, each of which contains a collection of primary and vernier jets, a fuel tank, an oxidizer tank, two helium tanks, and manifolds(see Figure 1 for an overview of RCS). Each system provides both fuel and oxidizer propellant flows. These flows are maintained by pressurizing the propellant tanks with helium. The helium supply is fed to its associated propellant tank through two redundant lines, designated A and B. A number of pressure and temperature transducers are attached at various parts of the system for monitoring. Each RCS module receives manual and automatic commands via the shuttle's general-purpose computers. The problem is to automate the malfunction procedures that diagnose and reconfigure the RCS when leaks are detected (see Figure 1for an overview of RCS).

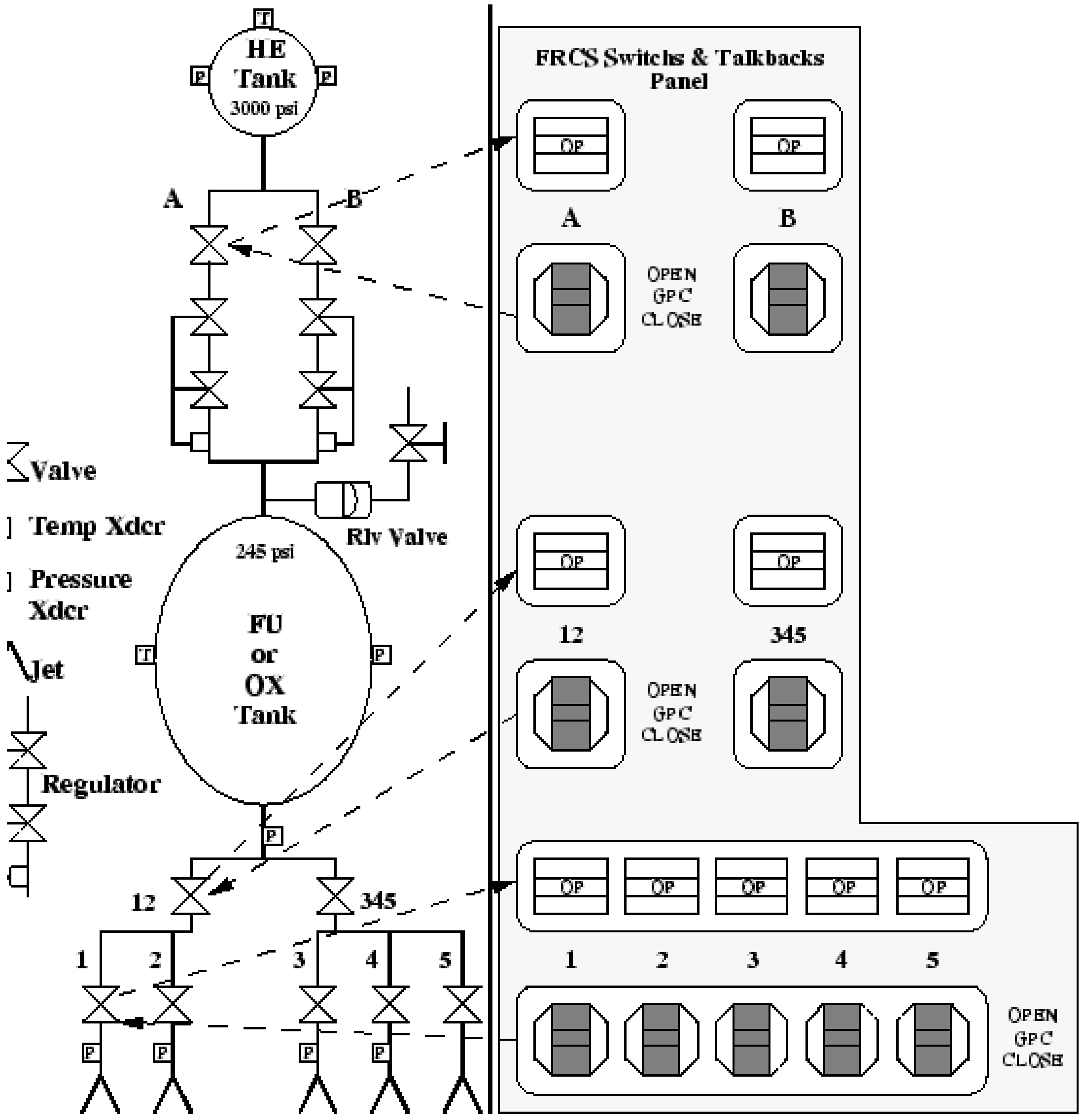


Figure 1. System Schematic for the RCS

Applications for space shuttle fault diagnosis have been developed using two architectures, the Procedural Reasoning System (PRS) of Ingrand and Georgeff [1, 2, 3] and the Distributed Multi-Agent Reasoning System (dMARS), a C++ implementation of the PRS architecture [6]. For fault handling purposes, they used fault analysis trees as a basis for developing agent plans. These applications utilize an expert's procedural knowledge [7, 8, 9] for accomplishing goals and tasks. Procedures for monitoring, diagnosing faults upon detection, and recovering from failures are automatically selected and executed to help keep the RCS working within required specifications. The applications demonstrate the real-time management of the Reaction Control System (RCS) on the NASA space shuttle. However, both PRS and dMARS, implemented in Lisp and C++ respectively, lack portability and extensibility, and neither provides efficient and sufficient explanation of the system activity to the user at runtime.

The PRS and dMARS architectures have grown out of using the BDI paradigm [1, 6]. The BDI architecture [10] is the basis for most agent-oriented systems. It typically contains four key components: beliefs, goals, intentions, and a library of plans. A BDI agent’s belief represents the agent’s knowledge about the world. An agent’s desires (or goals) are descriptions of desired tasks or behaviors. Intentions are an ordered set of chosen desires; an agent will try to achieve an intention until either it believes the intention is satisfied, or it believes that the intention is no longer achievable. The plan library contains a set of plans, which may be executed to achieve intentions.

A PRS module is made up of four components: a database, a set of current goals, a set of plans, and an intention structure, corresponding to the four keys of BDI architecture, respectively. The database stores the system’s current belief about the world. The intention structure consists of a (partially) ordered set of all plans chosen for execution at run-time. The set of plans are also called Knowledge Areas (KAs). Each KA describes a sequence of actions to perform in certain situations or to achieve some goal. Each KA consists of a body, which describes the steps of the procedure (usually in form of a tree, see section 3.1 for a similar tree), and an *invocation condition*, which specifies the situations for which the KA is useful. Together, the invocation condition and body of a KA express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions [1].

2.2. Explanation Component

A system becomes more complicated as it becomes more powerful. Sometimes it is difficult for a user to understand a system's behavior. Obviously, understanding the system makes a user more comfortable with and more confident in the system, and makes the coordination between the user and the system more effective. More, explanation is essential for training purposes.

An explanation component can provide users with a problem-solving context, discourse history, domain knowledge structure, etc. [4, 11]. All this information helps users understand the system's behavior. An effective explanation component therefore can be very useful and sometimes critical, especially when the system is highly sophisticated.

In this thesis we create an explanation mechanism that traces the sequence of activity involved the execution of an agent-oriented program, and displaysto users the activity in a tree-like structure The tracking mechanism is described in detail in Section 3.2.

2.3. Agent-Oriented Programming

Agent-Oriented Programming (AOP) [12] can be viewed as a specialization of the object-oriented programming (OOP) paradigm. OOP proposes viewing a computational system as collection of modules that are able to communicate with each other and that have individual ways of handling incoming messages. AOP specializes the framework by fixing the mental state of the agents (one kind of module) to consist of components such as beliefs, capabilities, and decisions. AOP makes development of agents easier.

JACK is an *Agent-Oriented* development environment built on top of and fully integrated with the Java programming language. It includes all components of the Java development environment as well as offering specific extensions to implement agent behavior. There are four basic constructs in JACK [13]: **agent**, **event**, **plan**, and **beliefSet**. Another construct, **capability**, is basically a collection of events and plans. As an object-oriented system is modeled in terms of objects, an agent-oriented system is modeled in terms of **agents**. In JACK, agents exhibit reasoning behavior following the BDI model of artificial intelligence [10]. In accordance with the BDI model, a JACK agent is an autonomous software component that has explicit goals to achieve or events to handle (**desires**). These agents are programmed with a set of **plans**. An agent pursues its given goals (**intentions**), adopting the appropriate plans according to its current set of data (**beliefs**) about the state of the world. This combination of desires and beliefs initiating context-sensitive, intentional behavior is part of what characterizes a BDI agent.

Events are the origin of all activities in a JACK system. Events can be posted within plans, agents and **beliefSet**s. Furthermore, there is a specific type of event, called an **automatic event**, which is posted automatically whenever some logical condition is satisfied. This latter is particularly useful for initiating a goal upon the occurrence of some external event, e.g., the detection of a leak. Within plans and agents, events can be posted explicitly through a variety of reasoning methods. This corresponds to establishing a goal in the BDI methodology. Some agent must be defined to handle each type of event (one agent can handle multiple event types, if desired). Agents can select among various plans to handle the event, depending upon the situation.

In a JACK agent system, each event type is handled by a specific agent. The agent may try one or more (in case of plan failure) plans for handling the event. These plans may post yet other events for the same or other agents. For example, in trying to isolate a leak, an agent may ask another agent to read the pressure on a specified manifold. The sequence and relationships of these events give the user a good picture of the process being used to resolve a situation. The sequence of events can be represented in a hierarchical structure that resembles the file structure on a computer. We will explore allowing a user to look selectively at portions of the hierarchical representation, much the same as one manipulates a file system in Windows Explorer. In addition, we want to show values returned from events, which further helps the user follow what the agent system is doing.

Each plan is capable of handling a single event, which is identified by a `#handles event` declaration in the plan. When an instance of a given event arises, an agent may execute one of the plans that declare they handle this event type. A JACK plan consists of a plan body, user-defined reasoning methods, and normal Java methods. Within the first two, additional events can be posted. A plan body is the sequence of actions to be executed when the plan is invoked. Other methods are called within the body.

2.4. Aspect-Oriented Programming

Aspect-Oriented Programming (ASOP) [14, 15, 16, 17, 18] is based on the idea that computer systems are better programmed by separately specifying the various concerns (properties or areas of interest) of a system, and composing them into a coherent program using the ASOP technique. In a typical application, there is one core concern and number of other concerns [15, 16]. For instance, the core concern of the FDIR system is fault detection, isolation, and recovery; another concern is the tracking of system activity. The behavior of other concerns, such as the tracking of system activity, would normally cross-cut behaviour of the core concern of the application. If developed as in the past, the code for such a concern would often scatter into several structural elements and become unnecessarily complicated. ASOP separates such crosscutting concerns from the core concerns and simplifies the realization of them. To do so, ASOP provides the concept of aspects: mechanisms for localizing the expression of crosscutting concerns. By collecting crosscutting concerns into aspects, ASOP congeals behavior that traditional programming languages would distribute throughout the system into a single textual structure, and makes the code cleaner and easier to understand.

Human beings think and speak by specifying various concerns. For example, "call a method track() before any event is posted. " Using traditional programming languages, we have to transform such a sentence into something like: "call track() before an event E1 is posted; and before E2 is posted; and so on. " Code has to be inserted wherever an event is posted. The power of ASOP is that it supports a richer set of structural expressions in which human beings think and speak. For instance, the sentence above can be realized as the pseudo-code below:

```
Before(): any event is posted{
          Call track();
}
```

with this code being state only once, rather that having to insert code at every posting of an event.

In section 5, the third phase of the research is described, in which we use ASOP techniques to develop a stand-alone tracking mechanism that can plug into the FDIR system as well as general JACK systems easily.

AspectJ[14] is an aspect-oriented extension of Java. It adds a new concept, a join point, and several new constructs: `pointcuts`, `advice`, `introduction` and `aspects` to Java. An `aspect` is AspectJ's unit of modularity for crosscutting concerns, and is defined in terms of the other three described above. `Pointcuts` and `advices` dynamically affect program flow, while an `introduction` affects a program's class hierarchy statically.

A `join point` is a well-defined point in the program flow. `Pointcuts` capture certain join points (and sometimes, values at those points), and invokes `Advice` code that will be executed.

An `introduction` changes existing classes by adding new members to existing classes and declaring a hierarchy between existing classes, e.g.: adding methods to an existing class, adding fields to an existing class, extending an existing class from another, implementing an interface in an existing class, and converting checked exceptions into unchecked exceptions. This provides a mechanism for `aspects` to set, modify and get values to help them achieve the cross-cutting conerns. While `advice` primarily operates dynamically at run time, an `introduction` works statically at compilation time.

3. PHASE 1: EXPLANATION-BASED, AGENT-ORIENTED FDIR SYSTEM

In this section, the first phase of this research is described. An agent structure to perform the FDIR tasks, and an explanation component, the system activity tracking, are created. Their integration as a comprehensive FDIR system is described.

3.1. The FDIR System

Fault detection, isolation, and recovery for the shuttle is typically based on malfunction procedures developed by NASA and its contractors. Once a failure is detected (based on sensor readings), these procedures describe a process for making additional tests and identifying the element that caused the failure. Malfunction procedures are essentially decision trees closely related to fault trees, which provide a logical analysis of conditions that could have caused the fault. The use of an agent architecture for implementing these provides great flexibility in adding or modifying procedures and in specifying the context in which each is appropriate to use. This was also the basis for the PRS FDIR implementation [1]; aside from being obsolete, however, the PRS implementation provided no explanation of agent activity. In this section, we describe the development of the isolation and recovery system, and in Section 4, we describe the addition of the activity tracking system.

We describe our approach to building agent-oriented FDIR systems by example in terms of the isolation of a leak within the Reaction Control System (RCS). There are five manifolds, multiple pressurized gas tanks, several feed tubes connecting the tanks to the manifolds and a number of valves in an RCS system, and a leak in any of them could cause an overall leak to be detected (see Figure 1).

To maintain knowledge of the state of the world a **beliefSet** is defined. Our agent system monitors the world (i.e., the RCS) and manipulates its **beliefSet** according to the sensed status. In order to initiate fault isolation activity, a set of automatic events are defined. For example, `Leak` (shown in part below) defines the automatic event that initiates activity for a fault in the forward RSC system. When a change of the world's state is detected, the system may take action based on the rules defined (via automatic events, for instance). For example, when a sensor detects a leak, the FDIR system modifies a **beliefSet** to record the leak. An automatic event `Leak` is then triggered, resulting in the addition of a goal to handle this leak. The following code illustrates the automatic posting of the event `Leak`:

```
public event Leak extends InferenceGoalEvent {


#uses data DisplayLeak dl;

logical String $sub-system;

…

#posted when (dl.leaking( $sub-system) );

…

}
```

The statement `#posted when (dl.leaking($sub-system) );` defines the condition under which this event will be triggered; it also serves as the constructor of the event and can be expanded to the following form:

```
#posted when (dl.leaking( $sub-system) ){

    //constructs the event

};
```

The variable `dl` is an instance of a **beliefSet** `DisplayLeak` (declared by the `#uses data` statement), `dl.leaking($sub-system)` is a **beliefSet** querying function, in this case, will return true if a leak is detected and the variable `$sub-system` will be bound to the sub-system where a leak is detected using unification (and the value of `$sub-system` cannot be changed after that). The querying function will be performed whenever `DisplayLeak` is modified and the event will be raised automatically when the querying function returns true.

This exposes a general mechanism for establishing isolation goals. The initial detection consists of two stages, the updating of the **beliefSet** based upon sensor inputs, and the definition of the conditions under which automatic events are to be raised (and, of course, the events themselves). Since plans and agents are defined independently to handle each event type, a flexible and easily extensible system results. One can simply add automatic event conditions, event types, agents and plans as needed. This notion is illustrated in Figure 2.

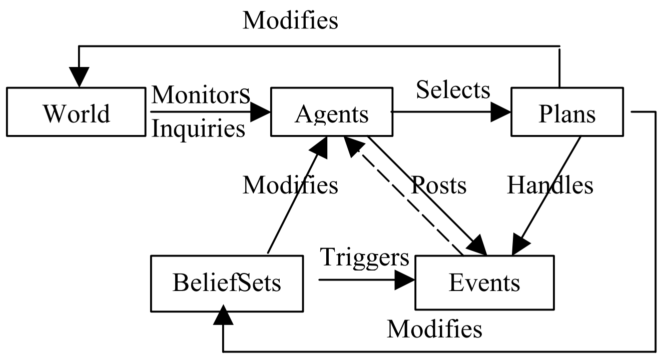


Figure 2. Relation of major component types in the FDIR system

As shown by Figure 2, postings of events (addition of goals) invoke agents to handle the events (to achieve the goals). Agents then select plans to handle the events. The plans may cause postings of other events directly or indirectly (via modifying the **be-liefSet**s, which then triggers automatic events). In this FDIR system, these correspond to raising and achieving a sequence of goals such as checking pressure of devices of the RCS system (inquiring the world), securing the RCS system (modifying the world), and finally isolating the leak (modifying **beliefSet**s) and recovering the RCS system.

Figure 3 shows a partial decision tree for isolating a leak. This is similar to the Knowledge Areas used in PRS [1].

To reason about which plan to use to try to achieve a goal, an agent may use the following three steps. First, only a plan defined to handle the event can be used. Second, each plan may have a `#context` clause containing the conditions that test one or more of the agent's **beliefset** relations. Only if the conditions are satisfied may such a plan be used. Third, a plan may have a `#relevant` clause, which takes an event instance as parameter and states an admission condition based upon parameters of the event. The first step is similar to PRS-KA's triggering conditions, and the other two steps are similar to PRS-KA's context part.

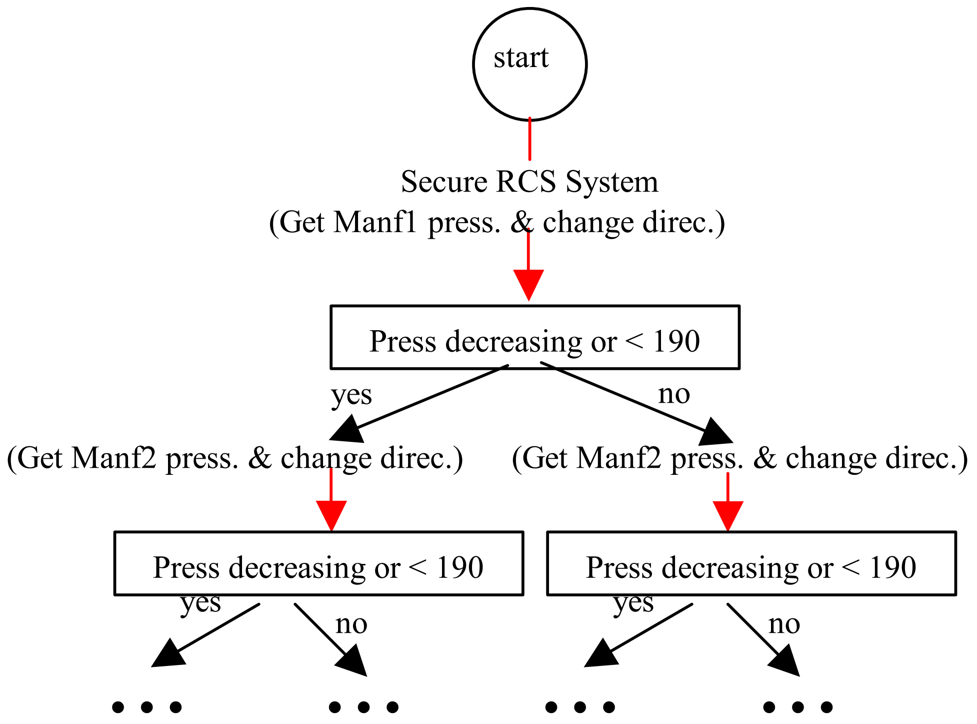

start
Secure RCS System
(Get Manf1 press. & change direc.)
Press decreasing or < 190
yes
no
(Get Manf2 press. & change direc.)
(Get Manf2 press. & change direc.)
Press decreasing or < 190
Press decreasing or < 190
yes
no
yes
no

Figure 3. Partial decision tree for leak isolation

When the conditions defined by `#context` clause and the `#relevant` clause (if available) is satisfied, the plan starts its execution, i.e., the execution enters the body of the plan. Within the body, a plan can call user-defined reasoning methods or pure Java methods, and can post events if necessary to add goals for agents (the one that "owns" the plan itself or other agents) to achieve. Below is part of the plan that performs the tasks based on the decision tree shown in Figure 3.

```
plan leakIsolPlan extends Plan {
   #reads data Who w;
   #uses data MessageDialog messageDialog;

  #handles event leakIsolEvent lie;
  #sends event Response re;
  #posts event getPressPresschgEvent pressPressChng;
  #posts event getPresschgEvent pressChng;
  #posts event secureEvent sec;

  logical String $agent;
  double Limit = 190;

  context(){
       …
  }

    #reasoning method
    descreasing(String what){
        …
    }
    #reasoning method
    increasing(String what){
        …
    }
    #reasoning method descreasingOrBelowLimit(
        logical String what,double  limit){
        …
    }
    #reasoning method increasingAndAboveLimit(
        logical String what, double limit){
        …
    }

body() {
    @subtask(sec.secure());
    if (increasingAndAboveLimit( $manf1, Limit) ) {
        if ( descreasingOrBelowLimit( $manf2, Limit) ) {
          lie.leaking_part = $manf2.toString() ;
        }
        else {
          if(increasingAndAboveLimit($manf3, Limit)){
            if(increasingAndAboveLimit($manf4,Limit)){
               if ( increasing( $hetk.toString()) ){
                 if(increasing($proptk.toString())){
                   lie.leaking_part="Helium leg of "+
                   lie.rcs+" "+$manf5.toString()+" Leak";
                 }
                 else {
                   lie.leaking_part=$proptk.toString();
                 }
               }
               else {
                  lie.leaking_part = $hetk.toString() ;
```

```
                    }
                  }
                  else {
                     lie.leaking_part = $manf4.toString() ;
                  }
              }
              else{
                 if(increasingAndAboveLimit($manf4, Limit)){
                       lie.leaking_part = $manf3.toString() ;
                  }
               else {
                   lie.leaking_part = "3-4-5-TANK-LEG Leak";
               }
             }
          }
        }
        else {
           if(descreasingOrBelowLimit( $manf2, Limit)){
               lie.leaking_part ="1-2-TANK-LEG";
           }
           else {
               lie.leaking_part =$manf1.toString() ;
           }
        }
    @wait_for(messageDialog.display(lie.leaking_part));
       @reply(lie, re.response());
        }

     }
```

While one could use a single agent to handle all event types in the system, it is an inelegant and difficult-to-maintain solution. First, as the system grows, the agent file becomes so huge with a whole lot of event and plan definitions in it that it is difficult to maintain. Second, such an agent is hardly reusable. In reality, a human performs only a certain type of jobs and incorporates with others when a task is beyond his capability. An agent system usually employs multiple agents (multi-agent system has become a popular research area). In our example, there are two agents that work together to manage a leak alarm and isolate the leak. An agent, called `LeakManager`, monitors the world. When it detects a possible leak, it will issue an alarm, check for regulator blockage or failure, and put the system into a safe mode if either of these has occurred. If these are not the failures, it determines that a leak has occurred and raises an event, `leakIsolEvent`, to set a goal for another agent to isolate the leak. It is worth noting that this also models a kind of teamwork.

In a JACK system, one agent contacts another by referencing its name. Thus, there has to be some means by which LeakManager can determine the isolator's name. We could simply make the former remember the name of the latter. However, a more scalable and maintainable approach is needed. For example, when the system grows and hundreds and thousands of agents have to be incorporated, it will be extremely inefficient to make the agents remember each other. **L**anguage for **A**dvertisement and **R**equest for **K**nowledge **S**haring (LARKS) [19, 20, 21] addresses the problem of agent interoperability. LARKS is an agent capability description language that lets an agent register a description of its capabilities with a middle agent; an agent can also request a service through the middle agent, and the middle agent will search its database to find a capability description that is similar enough with the service requested; if found, the service will be filled. The LARKS matchmaking process employs a matching engine that has five different customizable filters for context matching, word frequency profile comparison, similarity matching, signature matching, and constraint matching. In our FDIR system, a simpler but still effective way is used, which is to build a database to use as a look-up directory for the agents to find names corresponding to necessary functions. For example, the LeakManager queries the database for an agent that can do "leak isolation". The query returns the name of an agent that performs leak isolation.

To aid in the explanation of what an agent is doing, our system tracks the sequence of events, agents and plans involved in achieving a goal and displays them to the user. The explanation is in form of a hierarchical tree, with nodes representing events, agents, and plans. The relationship of the events, plans and agents is represented in the branching of the nodes, and part of the FDIR system's belief will be shown when a user puts the cursor upon a node. Details of the tracking mechanism are described below.

During this work, some general rules of constructing agent-oriented FDIR system based on the decision trees are summarized as following:

- For each decision tree, construct a plan.
- For each plan, construct a goal (an event) for invoking it.
- For invoking of another decision tree(a plan)
    - in the middle of execution of a decision tree (plan), raises a goal as a subtask (uses @subtask(..));
    - at the end of a decision tree(plan), raises a goal as a separate task
- System's belief stored in **beliefSet**s
- Use automatic events to construct rule-based actions

The advantage of our system over PRS and dMARS is that it is a platform-independent, inherently distributed, agent-oriented, explanation-based system.

3.2. Tracking System

In order to help a user understand how an agent oriented system operates, an agent activity tracking system has been created. All activity starts with the posting of an event. When an event is posted, an agent will handle it using a plan. The tracking system is based on displaying the nesting of event postings, agent handling, and plan usage in a

JTree. Different colors and icons are used to distinguish the three kinds of nodes: event, agent, and plan.

In order to keep track of the relationships among the events and plans in a nested posting hierarchy, a unique argument called `TrackingInfo` is associated with each event instance. The plan(s) that can be used by an agent to handle a specific event can be thought of as sharing the same `TrackingInfo` with the event. For example, the very first event posted, called E1, would be assigned "1" as its `TrackingInfo`. Suppose a plan P1 is used to handle E1; then if P1 posts events, E2, E3, E4 in order, the `Track-ingInfo` of E2, E3, E4 would be 1.1, 1.2, 1.3, respectively. The dot indicates the relationship "posted by"; and the increasing numeric values indicate the sequence order. Since the `TrackingInfo` is unique for each event (and the plan(s) handling them), the relationship among them is sufficiently well defined.[1] (See Figure 4)

As described above, the system starts with the posting of an event. The first event posting must occur outside of a plan, i.e., by an agent or an automatic posting. For the first event, `TrackingInfo` must be initialized to "1". If subsequent events are posted outside of a plan, the corresponding `TrackingInfo` must be initialized to the next increment of the number last used, i.e., the second one would be "2", etc.

Maintenance of `TrackingInfo` for events posted within plans involves two issues: appending a ".1" to `TrackingInfo` or incrementing `TrackingInfo`. The former occurs only when the first event within a plan is posted. The latter occurs when subsequent events are posted in a plan.

[1] The relationship is actually only partially defined. For E1 and E2 with TrackingInfo 1.3.4 and 2.2.5, we cannot determine their posting order. However, the order is not needed for representing the agents' activities using a tree-like structure.

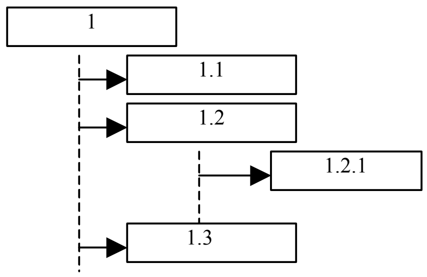


Figure 4. The sample tracking tree

Two extra fields are added to each plan in order to maintain the correct `Track-ingInfo`. The first is a `boolean firstEvent,` which is initialized to `true` when a plan is created and used to control the `TrackingInfo` passed to events the plan raises. The second is a `String, tracking,` which is used to assign `TrackingInfo` to every event posted within this plan. When a plan is created, `tracking` will be initial-ized to the value of `TrackingInfo` of the event handled by that plan. When an event is posted within a plan, the value of `firstEvent` will be checked; if is `true`, the value of `tracking` will be assigned to the `TrackingInfo` of the event being posted and the value of `firstEvent` will be changed to `false`; otherwise, the value of `tracking` will be incremented and then assigned to the event being posted. This strat-egy maintains the sequence and nesting-level information.

In order to provide more details of the operation of the FDIR system, a technique for displaying the current system belief has been developed in addition to the process tracking system. When the user moves the cursor over an event or a plan in the tracking tree, a portion of the system belief associated with the event/plan will be displayed, tell-ing what is going on. If the plan handling the event has not finished its execution, and the cursor is over an event, a text description will be displayed, telling what the event is for.

If the plan has finished, the result of the execution will be appended to the original text display of the event. For example, in the FDIR system, "*possible leak or regulator failed closed*" is displayed when the event `Leak` is first posted but "*possible leak or regulator failed closed-Leak alarm on*" is displayed after the event has been successfully handled. For plans, in the first case, the display will just say that the plan is still working; while in the second case, the result will be displayed.

A protocol has been developed for expressing the textual descriptions and data to be displayed. First, for each event in the system, an extra property called `goal` is added, which is a text description of the goal of the event. Another property, called `result` will be used to specify what result should be displayed. If the event has parameters whose values are to be decided by the plan handling the event, `result` specifies which parameters are to be displayed; otherwise, `result` can be a message that says the event has been handled successfully and the goal achieved. JACK program developers are required to provide meaningful, customized values for `goal` and `result.` Since the former is static, it can be directly defined in each event type. For example, the code below is included in the event type `leakIsolEvent`:

```
String goal = "Isolate the leak."
```

However, the latter can be dynamic and undetermined before runtime. For instance, when a programmer wants to include a variable in `result`, that variable usually is not initialized until the event is handled successfully. Our solution is to require the programmer to define a method, called `getResult()` within the event type, which returns the value of `result`. This method will be called only at the end of the execution of a plan, after all related variables have had their values defined. Below is a sample of the `getResult()` method for `leakIsolEvent`.

```
String leaking_part;

getResult(){

   return result =

"The leaking part is " + leaking_part;

}
```

The plan used to handle this event type will, if successful, place the correct value in the string `leaking_part`. The plan then passes the result to the tracking system to enable it to display the result for the user.

The key component of the tracking system is a tracking agent created to handle two event types, `TrackingEvent` and `returnValueDisplayEvent`; it uses two plans, `TrackingPlan` and `returnValueDisplayPlan`. As described above, extra fields are added to events and plans in FDIR system. Then, in each plan other than the two plans belonging to the tracking system, an instance of `TrackingEvent` that packs `TrackingInfo` and other information about the event/plan is created and sent to the tracking system. This is done at the beginning of the execution of each plan in the FDIR system. The `TrackingPlan` decodes this information and displays the activity of the FDIR system based on this information. When a plan succeeds, an instance of `return-ValueDisplayEvent` that packs `TrackingInfo` and the `result` of the event is created and sent to the tracking system immediately before the plan returns. The `re-turnValueDisplayPlan` decodes the `TrackingInfo` and associates the `re-`

`sult` with proper event and plan. Figure 5 Shows the interaction of the FDIR and tracking systems.

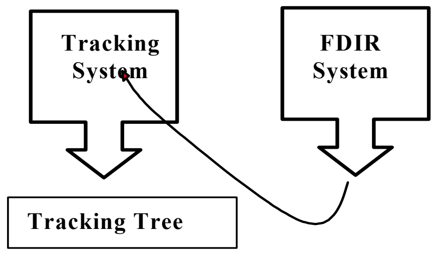


Figure 5. The interaction of diagnostic system and tracking system

The following code, which is part of a plan shows how this is done. Note that `trackingAgent` is a String representation of name of the agent handling tracking stuff, and `_tracking_Event` is an instance of TrackingEvent.

```
body() {
/********Here is the beginning of a plan's execu-
tion********/
@send(trackingAgent, _tracking_Event =
_tracking_Event.tracking(lie.TrackingInfo,"leakIsolEvent"
,"leakIsolPlan", agnt,lie.initialDisplay)) ;
…
}
```

4. THE SECOND PHASE: UTILIZING THE PREPROCESSOR

In order to avoid the onerous task of manually inserting all calls to the tracking system, a preprocessor has been built that automatically inserts all code necessary for use of the tracking system.

The preprocessor has four parsers for JACK files, one each for agents, events, plans, and capabilities. Given a folder containing a JACK agent system, the preprocessor searches for the JACK files and then uses the parsers to analyze the files and insert necessary code. There is also a parser for Java files. The preprocessor looks for the Java file which contains the `main()` method, and inserts the code that instantiates the tracking system.

The event parser will add an extra parameter called `TrackingInfo` to each event. Extra code is also added to initialize `TrackingInfo`.

The agent parser will insert the reference code to import the tracking agent and events, and a declaration that two events, `TrackingEvent` and `returnValueEvent`, be sent by the agent. If there is any event posted within the agent file[2], code will be inserted to generate the tracking information for the event the agent is posting. The capability parser is similar to the agent parser.

The plan parser does the toughest work. It inserts reference code and declaration code as described above; it inserts code that maintains local tracking information within itself; it assigns proper tracking information to any event it posts. It also inserts code that

[2] All events could be said to be posted by the agents, whether they are posted within a plan, a database, or elsewhere; but here we are talking about the case in which there is a method of an agent, which posts an event.

creates instances of `TrackingEvent` and `returnValueEvent` carrying proper information and sends them to the `TrackingAgent`. An instance of `TrackingEvent` will be sent each time an event (except the two above) is posted. `ReturnValueEvent`, however, is sent only at the successful completion of a plan's execution. In JACK, a plan has two member methods, called `pass()` and `fail()`. If the plan succeeds, the former will be executed immediately before the plan returns; the latter will be executed otherwise. Both can be overloaded by the user to include user code to be executed. The plan parser checks whether or not these two methods are overloaded. If they are, the parser adds code necessary to create an instance of `returnValueEvent` that carries proper information about the current plan and sends it to `TrackingAgent`. If the two methods are not overloaded, then the parser adds the overload that performs the same actions as described above. By doing so, `TrackingAgent` is informed of the success or failure of a plan's execution and the result of its execution.

In this section, we show the code of an event leakIsolEvent, both before and after preprocessor modification, and the plan that handles that event, leakIsolPlan. The extra code added by the preprocessor is in bold.

Figure 6 shows the code for leakIsoEvent before the preprocessor modifies it. This event is posted whenever a leak is detected. The field `goal` specifies the reason for posting this event, i.e., to "find out which part caused leak"; The method `getResult()`, which returns a String telling the result of the handling of the event, uses a variable `leaking_part` that is not defined until the successful execution of the plan that handles this event.

```
event leakIsolEvent extends BDIMessageEvent {
 String rcs;
 String goal;
 String result;
 String leaking_part;

 #posted as isolate(String rcs) {
   this.rcs = rcs;
   goal=rcs+" is leaking,
      finding out which part caused leak";
 }
 public String getResult(){
     return result ="Leak Isolated.The leaking
        part is " + leaking_part;
  }
}
```

Figure 6. Code for leakIsolEvent before modification

Figure 7 shows the modified `leakIsoEvent`. The preprocessor adds the code shown in bold. Note that an extra field, `TrackingInfo` is added to the event; and the constructor (or posting function, as JACK documentation refers to it) of the event is modified to include an extra parameter that will initialize `TrackingInfo`.

```
event leakIsolEvent extends BDIMessageEvent {
 String rcs;
 String goal;
 String result;
 String leaking_part;
 String TrackingInfo;
 #posted as isolate(String t, String rcs) {
   TrackingInfo = t;
   this.rcs = rcs;
   goal=rcs+" is leaking,
      finding out which part caused leak";
 }
  public String getResult(){
     return result ="Leak Isolated.
        The leaking part is " + leaking_part;
  }
}
```

Figure 7. Code for leakIsolEvent after modification

Figure 8 shows the plan, `leakIsoPlan`, which handles `leakIsoEvent`. The modified version is shown in **Figure 9**. Notice that the preprocessor adds extra code to the plan that imports the necessary package (containing the tracking system), that declares and maintains an extra field, `String_tracking_info`, and that interacts with the tracking system. At the end of the plan, two methods, `pass()` and `fail()` are added, which overload the two methods in the parent class plan. Upon the successful end of execution of the plan, `pass()` will be called, which sends necessary information to the tracking system so that the tracking system can display the result of the plan's execution to the user. If the plan fails, `fail()` will be called instead, which informs the tracking system the failure of the plan's execution.

```
package Isolator;
...
plan leakIsolPlan extends Plan {
  #handles event leakIsolEvent lie;
  #sends event Response re;
  #posts event getPressPresschgEvent pp;
  #posts event secureEvent sec;
  ...
  #reasoning method descreasingOrBelowLimit
        (logical String what,double limit) {
   pp.getValue(…);
          …
  }  //end of reasoning method
//other reasoning methods ...
 body() {
  @subtask(sec.secure());
  if(increasingAndAboveLimit($manf1,Limit)){
    if(descreasingOrBelowLimit($manf2,
        Limit)){
      lie.leaking_part = $manf2.toString();}
    else{
      if(increasingAndAboveLimit($manf3,
        Limit)){    ...  }  ...  }  ...
  }  ...
  @reply(lie, re.response());
 }
}
```

Figure 8. Code for leakIsolPlan before modification

A full set of rules for tracking code insertion is appended at the end of this thesis. Please see the appendix for more details.

```
package Isolator;
import TrackingAgent.*;
plan leakIsolPlan extends Plan {
 #sends event TrackingEvent _tracking_Event;
 #sends event returnValueEvent return_Event;
 String _tracking_info;
 String trackingAgent = "Tracker";
 #handles event leakIsolEvent lie;
 #sends event Response re;
 #posts event getPressPresschgEvent pp;
 #posts event secureEvent sec;
 #reasoning method descreasingOrBelowLimit
      (logical String what,double limit) {
   …
  pp.getValue(_tracking_info=_tracking_Event.
   increment(_tracking_info),…);
 …}   //of reasoning method
//other reasoning methods ...
 body() {
  String agnt = getAgent().name();
  @send(trackingAgent, _tracking_Event =
        _tracking_Event.tracking(
          lie.TrackingInfo,"leakIsolEvent",
          "leakIsolPlan",
          agnt,lie.initialDisplay));
  _tracking_info =
     _tracking_Event.append(lie.TrackingInfo);
  @subtask(sec.secure(_tracking_info =
                  _tracking_Event.
                  increment(_tracking_info)));
  if(increasingAndAboveLimit($manf1,Limit)){
    if(descreasingOrBelowLimit($manf2, Limit)){
      lie.leaking_part = $manf2.toString();}
    else{
      if(increasingAndAboveLimit($manf3,
        Limit)){    ...  }  ...  }  ...}  ...
  @reply(lie, re.response(_tracking_info =
    _tracking_Event.increment(_tracking_info)));
 }   //end of body
 #reasoning method pass(){
   @send(trackingAgent,
   return_Event.returned(..));}//end of method
 #reasoning method fail (){ ..}
 }  //end of reasoning method
}  //end of plaN
```

Figure 9. Code for leakIsolPlan after modification

5. THE THIRD PHASE: ASPECT-BASED TRACKING

In Section 4, a preprocessing tool is described that automatically adds the system-activity tracking capability to an arbitrary agent system. The preprocessing tools saves system developers' effort and time to build the tracking mechanism from scratch. However, it modifies the original code of the agent system and it is difficult to undo the insertion without backup of the original. In this section, a more flexible and powerful approach is described with use of Aspect-Oriented Programming tool. In particular, AspectJ is used.

5.1. Aspect Interaction with Tracking System

Both the static and dynamic constructs of AspectJ are important to this project. We employ `introduction` to add an extra parameter, TrackingInfo, to all events, as well as methods to set and get that parameter. We define `Pointcuts` to capture the points in program flows where an event is posted, and `advice` to set the extra field, TrackingInfo, for the event being posted, and pass the information on to the plan handling this event. Together with other AspectJ constructs, we can track and display the JACK system activity in a tracking tree, as we did previously, but without using code-parsers and tracking agent. The `aspects` will form the tracking system. Figure 10 is an overview of the proposed system.

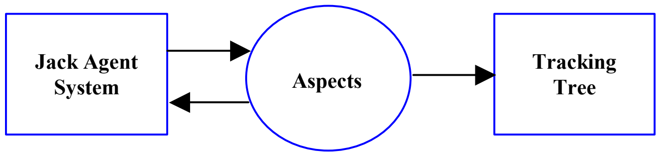


Figure 10. Overview of the system with AspectJ

The `pointcuts` were originally developed for specific events and plans. However, the straightforward approach would require that *`pointcuts`* be produced for each individual event and plan. With such an approach, if one wanted to use AspectJ for a different JACK system with different events and plans, the `pointcuts` would have to be re-written, making it difficult to reuse them. To generalize the `pointcuts` as well as the tracking mechanism, we developed two interfaces (In appendix, see 1A.2), `ajcBasePlanInt` and `ajcBaseEventInt`, for the plans and events, respectively. The `pointcuts` are defined to see only the activity of these two interfaces. We then require that the events and plans in a specific JACK system implement these interfaces; in this manner, the activity of the system is visible to the `pointcuts` and can be tracked and displayed to the user, regardless of what JACK-based system is used.

All events and plans in given JACK system will be declared to implement the corresponding interface using the AspectJ *`introduction`*. For instance, the following code makes plan `leakIsolPlan` implement `ajcBasePlanInt`.

```
declare parents: leakIsolPlan implements ajcBasePlanInt;
```

Also via the AspectJ *`introduction`*, an additional field, `ajcBaseEventInt handled_event`, is added to each plan; this field is used to refer to the event handled by the plan. The *`pointcuts`* then deal with the base classes rather than individual events and plans; therefore, a single copy of each *`pointcut`* is enough, no matter how many events and plans there are in a given JACK system. Moreover, the *`pointcuts`*, which are generalized, are separated from the AspectJ *`introduction`*, which is dedicated to a specific JACK system. Therefore, the `aspect` containing the `pointcuts` need not be modified when it is applied to different JACK systems.

However, as seen in the code sample above, each plan and event must explicitly appear in the `introduction`. Fortunately, it is easy to write a short preprocessor to examine the directory in which the JACK code is placed and to generate the needed `introduction` automatically. Details will be provided in 5.2.

The modified system is shown as in Figure 11.

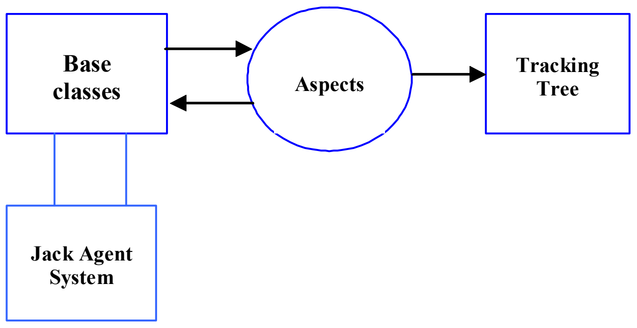
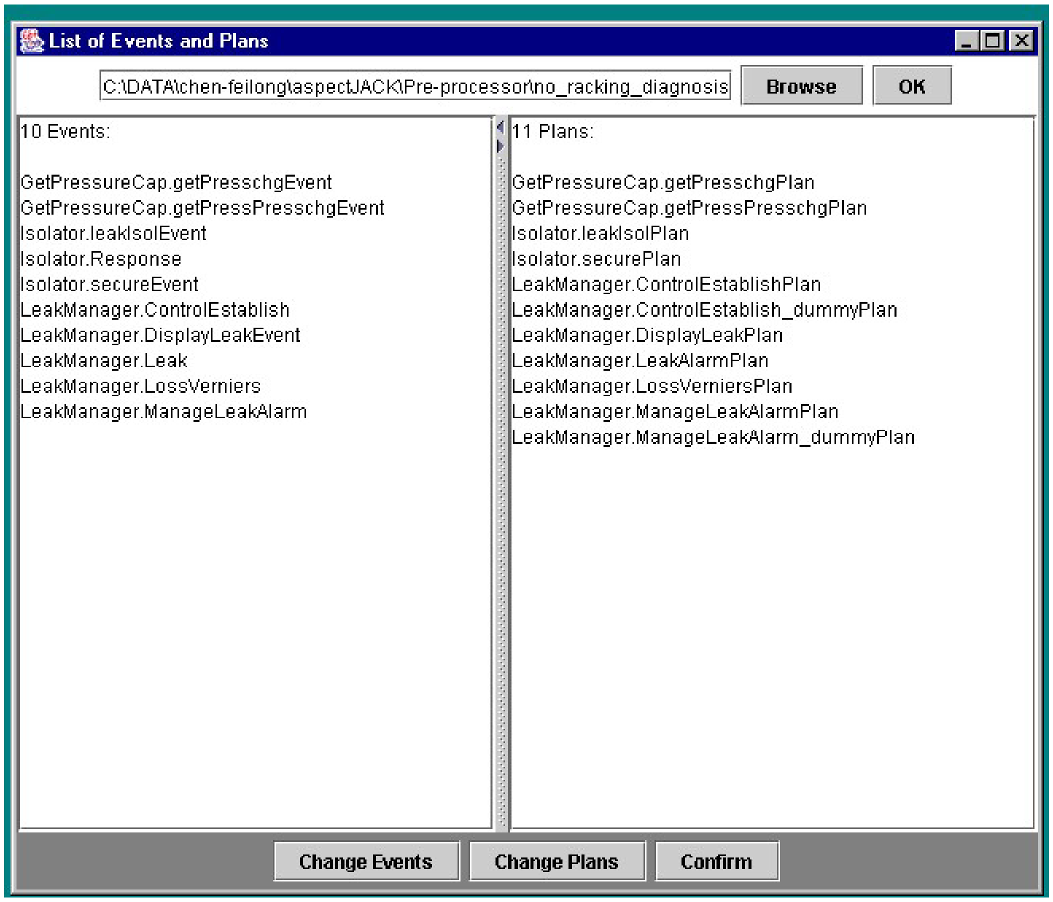


Figure 11. Overview of the modified system

5.2. A Simple Processing Tool

To make the events and plans in a specific JACK system implement the interfaces automatically, a preprocessing tool has been developed. When running the processing tool and specifying a folder that contains the JACK system, the preprocessing tool collects the necessary information, i.e., the event and plan types, and then generates an aspect that makes the event and plan types implement the interfaces. It then adds to them extra fields and methods as needed for tracking purposes. This is accomplished via an AspectJ introduction. Figure 12 is a snapshot of the preprocessing tool.



```
C:\DATA\chen-feilong\aspectJACK\Pre-processor\no_racking_diagnosis
```

```
10 Events:

GetPressureCap.getPresschgEvent
GetPressureCap.getPressPresschgEvent
Isolator.leakIsolEvent
Isolator.Response
Isolator.secureEvent
LeakManager.ControlEstablish
LeakManager.DisplayLeakEvent
LeakManager.Leak
LeakManager.LossVerniers
LeakManager.ManageLeakAlarm
```

```
11 Plans:

GetPressureCap.getPresschgPlan
GetPressureCap.getPressPresschgPlan
Isolator.leakIsolPlan
Isolator.securePlan
LeakManager.ControlEstablishPlan
LeakManager.ControlEstablish_dummyPlan
LeakManager.DisplayLeakPlan
LeakManager.LeakAlarmPlan
LeakManager.LossVerniersPlan
LeakManager.ManageLeakAlarmPlan
LeakManager.ManageLeakAlarm_dummyPlan
```

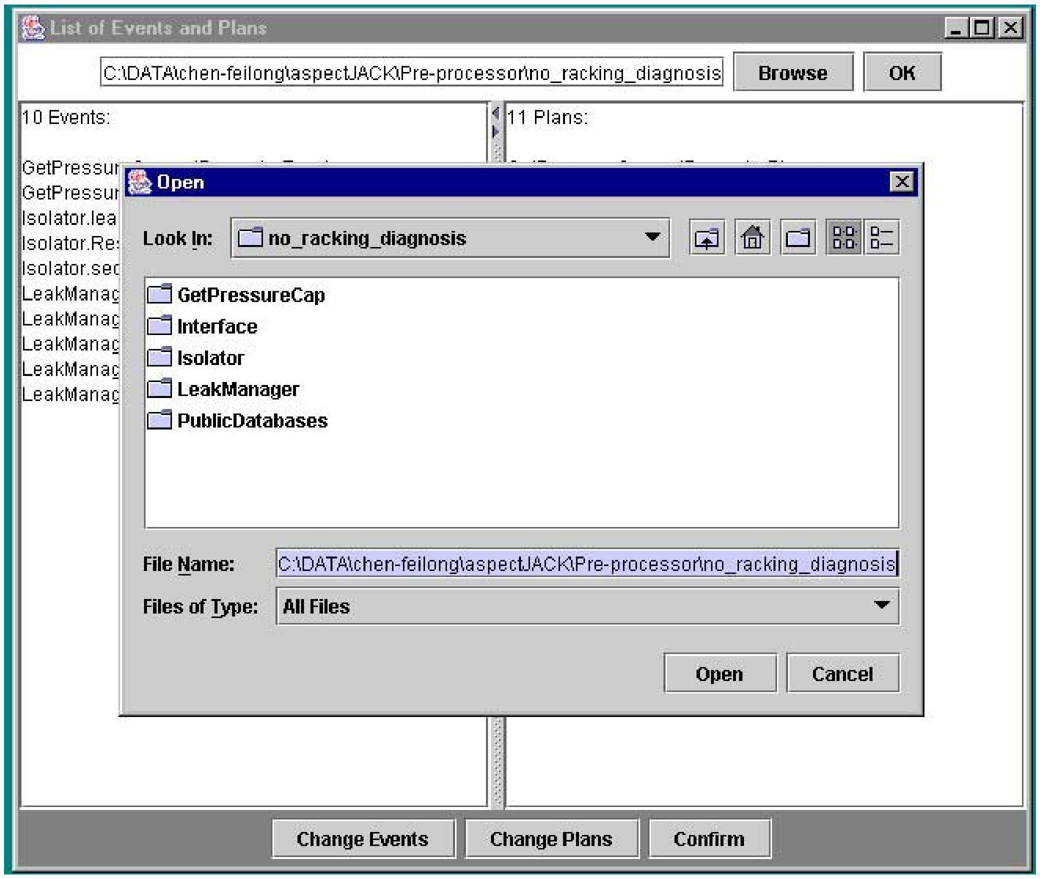
Figure 12. The preprocessing tool

As shown in the figure, the preprocessing tool allows the user to specify the location; events and plans found in that location are displayed to the user. If for some reason the userwants to remove an event from the list so that it won't be tracked, he or she can click **Change Events**. The list then becomes editable. The default state for the list is uneditable to prevent accidental editing of the list.

To specify the location of a JACK system, the user types the path directly, or clicks **Browse** to select a folder (see Figure 13). After selecting a folder, the user clicks **OK** and then the lists of events and plans are updated. After the lists are updated, the user clicks **Confirm**, and the preprocessor generates an aspect, based on the lists and then exits. The aspect is named following the convention `jack-`

`Folder_relation_modifier.`

`java`, where `jackFolder` is the name of the folder that contains the JACK system; for the example, in Figure 13, the aspect will be named `no_tracking_diagnosis_relation_modifier.java`. The aspect generated will be stored where the package of the preprocessing tool is located.



C:\DATA\chen-feilong\aspectJACK\Pre-processor\no_racking_diagnosis

Figure 13. Selecting a folder containing the JACK system

In the remaining of this section, several *`pointcuts`* are described in detail. The aspect that contains a full set of developed *`pointcuts`* can be found in the appendix (See 1A.2).

5.3. Initializing Tracking Fields in a Plan

First, we describe a *`pointcut`* (Figure 14) that detects the creation of every plan and initializes the two fields of that plan. The field `handled_event` is initialized to the event that the current plan will handle, and the string `tracking` is initialized to the string obtained taking the tracking information of the `handled_event`, i.e., `handled_event. TrackingInfo` and appending a ".1" to it.

A difficulty arises in finding an appropriate plan-creation call that AspectJ can trap. The creation code does not appear in the user-written plan code; rather, it is in the code generated by the JACK compiler. By studying the Java code generated, we found that a method with prototype

```
Plan createPlan(Event, Task)
```

is called by each plan to initiate itself, and this method can be used to define the *`pointcut`*, as shown in Figure 14.

```
pointcut handled_event(Event e): call(Plan
createPlan(Event, Task)) && args(e, ..);

after(Event e) returning(ajcBasePlanInt pa):
handled_event(e){
      pa.handled_event=(ajcBaseEventInt)e;
      pa.tracking = Tracker.append(pa.
        handled_event.getTrackingInfo());
        …
}
```

Figure 14. The pointcut initiating the fields for a plan

The code `args(e, …)` in the figure means that `e` is a reference to the first argument of `createPlan(Event, Task)` which gives the *`aspect`* access to that reference. The code within the braces is called *`advice`* in AspectJ; it is additional code that is run at the `joinpoints`. The `advice` in this `pointcut` initiates the two fields for the plan, using `e`.

5.4. Using Pointcuts to Capture Event Posting within Plans

A second `pointcut` captures the `joinpoints` where an event is posted within a plan via the posting function `@post(…).` Figure 15 shows the `pointcut`. Note that all event postings happen in either the plan body or a user-defined reasoning method. Again, the code that AspectJ needs to trap is not in the user-written code, but within the JACK-generated code. By studying the corresponding Java code, we found that both the plan body and the user-defined reasoning methods will be translated to a class that extends PlanFSM. Thus, (See Figure 15)

```
call(public * Agent.postEvent(..)) && args(ev,..) &&
this(fsm)
```

defines the joinpoints desired. Note a reference to the event being posted is obtained by

```
call(public * Agent.postEvent(..)) && args(ev,..)
```

and a reference to the currently executing object is obtained by

```
this(fsm)
```

Using the two references, the `aspect` has access to the reference of the current plan, together with its three fields and the event being posted. The *advice* then checks whether or not the event is the first event posted by this plan; if it is the first, the plan's `tracking` is assigned to the event's `TrackingInfo` and `firstEvent` is set to false;

otherwise the value in the rightmost position of the plan's `tracking` value is incremented and assigned to the event's `TrackingInfo`.

```
after(ajcBaseEventInt ev, PlanFSM fsm) returning():
  call(public * Agent.postEvent(..)) && args(ev,..)
    && this(fsm)  {
    ajcBasePlanInt pa = (ajcBasePlanInt)fsm.getPlan();
        if(pa.firstEvent){
           ev.setTrackingInfo(pa.tracking);
           pa.firstEvent = false;
        }
        else
           ev.setTrackingInfo(Tracker.increment(
                   pa.tracking));
   }
```

Figure 15. Pointcut for @post(...)

Another posting function `@subtask(…)` has also been implemented. This *`pointcut`* is defined by

```
call(public * *.push(..)) && args(ev) && this(fsm)
```

and the same `advice` is used. That is, the "call" part of Figure 15 is replaced by the one shown above.

More posting functions such as @send(..), @reply(..), @achieve(..), @insist(..) and @maintain(..),have been implemented, but are not described individually here.

5.5. Using Pointcuts to Capture Event Posting within Agents, beliefSets and Automatic Event Posting

Events can also be posted from within agents, through automatic events and from within **beliefSet**s. Different *`pointcuts`* are required for these. There are two posting functions `postEvent(…)` and `postEventAndWait(…)` that can be used by an agent to post events. A *`pointcut`* was created for each of the two posting functions. Figure 16 illustrates the *`pointcut`* corresponding to `postEvent(…)`.

```
before(ajcBaseEventInt ev) : call(public *
Agent.postEvent(..)) && args(ev,..)&& this(Agent){
       ev.setTrackingInfo(Tracker.NextNum()+"";
  }
```

Figure 16. Pointcut for event posting within agents

However, this *`pointcut`* can be generalized for automatic event postings and event postings within **beliefSet**, by changing `this(Agent)` to `!this(PlanFSM)`. The modified *`pointcut`* is shown in Figure 17.

```
before(ajcBaseEventInt ev):call(public Agent.postEvent(..))
     && args(ev,..)&& !this(PlanFSM){
       ev.setTrackingInfo(Tracker.NextNum()+"";

}
```

Figure 17. Pointcut for event postings outside of plans

Changing `postEvent(..)` to `postEventAndWait(..)` we get the *`pointcut`* for the latter posting function.

5.6. Using Pointcuts to Display System Activity

`pointcuts` can also be developed to display the system activity. A class TrackDisplayer has been built to create the tracking window and dynamically add the nodes (events, plans and agents) to the tracking tree. The code shown in Figure 18 initiates an instance of TrackDisplayer when the system starts.

```
TrackDisplayer track;
pointcut main():
      execution(public static void main(..));
before() : main(){
     track = new TrackDisplayer();
  }
```

Figure 18. The pointcut initiating TrackDisplayer

Then the *`pointcut`* shown in Figure 14 is modified such that a method in TrackDisplayer is called to add to the tracking-tree nodes corresponding to the event being posted, the agent handling the event, and the plan used. The modified *`pointcut`* is shown in

Figure 19. `handledEvent()` and `getAgent()` are JACK plan methods. They return the string representation of the event type that the plan handles and the reference to the agent the plan belongs to, respectively. The method `name()` is a JACK agent's method, which returns a String representation of the agent. The method `addNodes(…)` has the prototype:

```
addNodes (String trackingInfo, String event, String agent,
String plan, String eventLabel)
```

By decoding the first parameter, TrackDisplayer finds the correct location to which to add the nodes; then it creates a node for each of the next three parameters. The last parameter is the text that will be displayed when the user moves the cursor over an event. Currently "No comment" is used for all events. Later, this parameter will be used to display the goal of the event.

```
after(Event e) returning(ajcBasePlanInt pa): handled_event(e){
        pa.handled_event = (ajcBaseEventInt)e;
        pa.tracking =
           Tracker.append(pa.handled_event.getTrackingInfo());
        String planName = pa.toString();
        String agentName = pa.getAgent().name() ;
        StringTokenizer st = new StringTokenizer(planName);
        track.addNodes(pa.handled_event.getTrackingInfo(),
           "Event " + pa.handledEvent(),"Agent " +
            agentName.substring(0,agentName.indexOf("@")),
            "Plan " + st.nextToken(), "No comment");

  }
```

Figure 19. Modified pointcut for adding nodes to the tracking tree

Another two `pointcuts` are created to set the text that will be displayed when the user places the cursor on a plan. Different comments will be displayed, depending on whether the plan succeeds or not. Figure 20 shows the two `pointcuts`.

```
before(ajcBasePlanInt pa): call(* pass()) && target(pa){
   track.returnValueDisplay(pa.getEvent().getTrackingInfo(),
         pa.getEvent().getResult());

  }


before(ajcBasePlanInt pa): call(* fail()) && target(pa){
   track.returnValueDisplay(pa.handled_event.
     getTrackingInfo(), "Plan failed.
          Try another plan.");
}
```

Figure 20. Pointcuts setting text-display for plans

Note that a method, `pass()`, will be called when a plan successfully completes its execution, and `fail()` will be called if it does not. The *`advice`* calls a method `track.returnValueDisplay(…)` to set proper text for the plan. The system builder could define the text (by defining the `getResult()` method) to be displayed for each plan and the *`advice`* will pick it up.

5.7. Plugging in the Tracking Mechanism

After the execution of the preprocessing tool, the tracking mechanism is ready to be plugged in. Next we describe how the tracking mechanism can be added to a specific JACK system.

First, we compile the JACK system using the JACK pre-compiler to generate corresponding Java files. This could be done either before or after execution of the preprocessing tool. AspectJ can apply only to Java code and thus the aspect-based tracking mechanism is useless without the Java code.

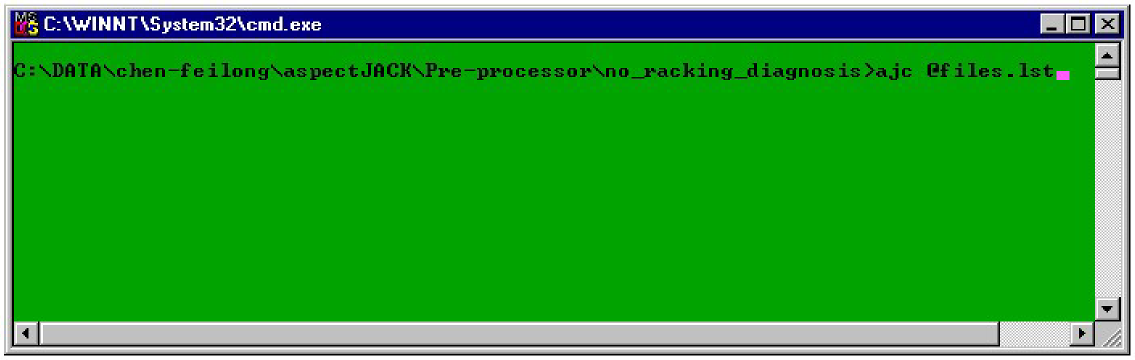
Second, we write an .lst file that includes all the Java files needed for compilation. The files include the two aspects and the two interfaces described above, files for tracking and displaying, files needed for the GUI, and the files generated from the specific JACK system. This file will be used as the "make file" for the AspectJ compiler. Refer to the AspectJ document for detailed format of the .lst file. Figure 21 shows a sample .lst file.

```
../EventTracking_display.java
../no_tracking_diagnosis_relation_modifier.java
../TrackDisplayer.java
../ajcBaseEventInt.java
../ajcBasePlanInt.java
../Tracker.java
*.java

GetPressureCap/*.java
Isolator/*.java
Leakmanager/*.java
PublicDatabases/*.java
```

Figure 21. A sample .lst file

Third, we execute ApectJ, giving the .lst file as an argument. One way for doing that is to use the windows command prompt (See Figure 22). Refer to AspectJ document for further instruction.



```
C:\DATA\chen-feilong\aspectJACK\Pre-processor\no_racking_diagnosis>ajc @files.lst
```

Figure 22. Using command prompt to execute AspectJ

After successful compilation, we can run the JACK system as usual. However, this time the activity of the JACK system will be tracked and displayed using a tree-like system. We can explore the tree and read about what the system is doing and how the event has been handled.

To unplug the tracking mechanism, we compile the system again using the JACK pre-compiler (Using the –clean option to remove JACK-generated files first). The aspect-based tracking mechanism is highly portable, easy to apply and easy to remove.

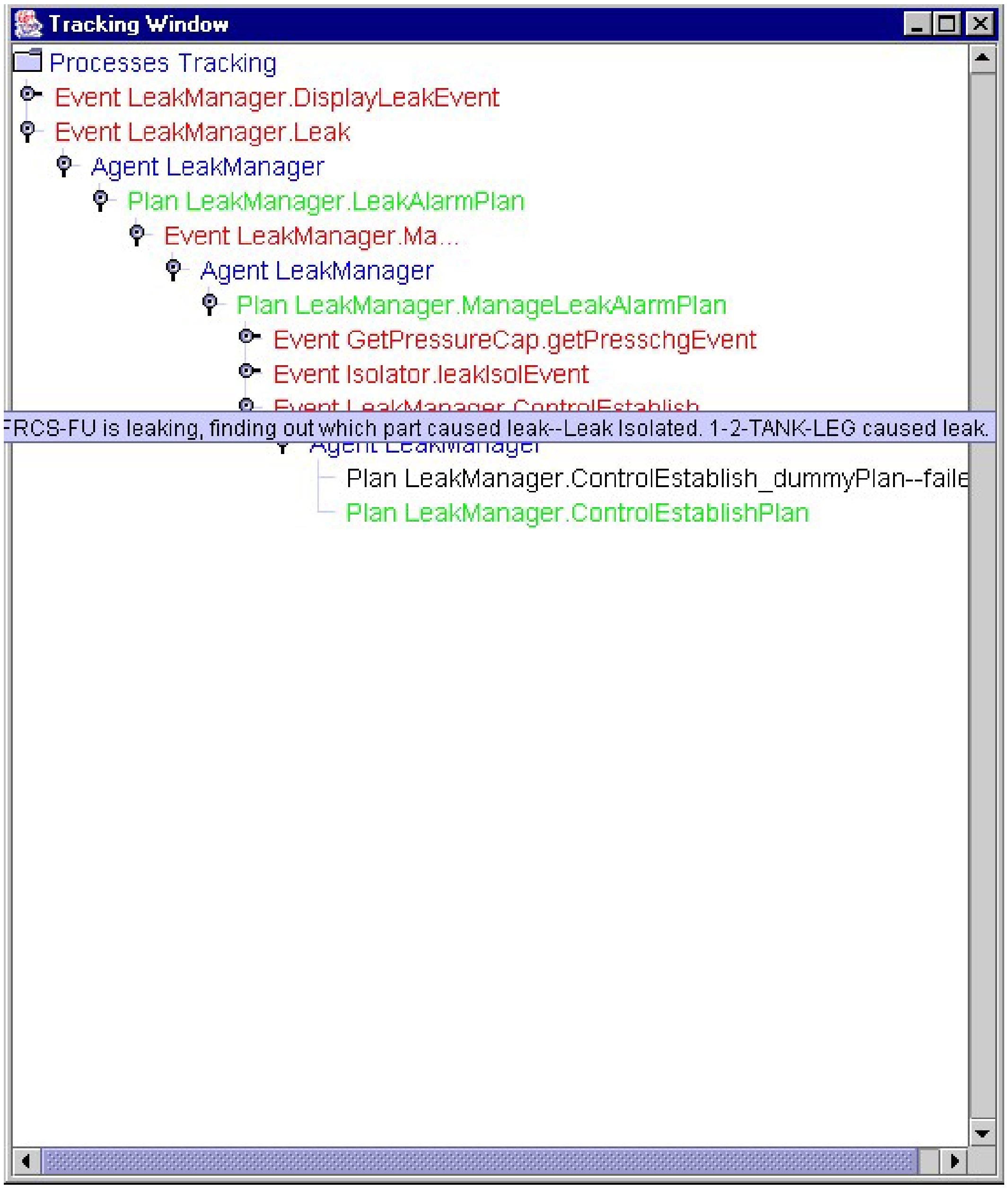
6. EXAMPLE AND RESULTS

We used the FDIR system to test the `aspects`. As described above, a preprocessor scanned the directory containing the sample system and produced an `aspect` containing `introductions` declaring events and plans implement the two base interfaces. Figure 23 shows part of the relation-modifying aspect generated for the FDIR system.

```
aspect no_tracking_diagnosis_relation_modifier {
/*********code weaving GetPressureCap.getPresschgEvent**********/
  declare parents: GetPressureCap.getPresschgEvent implements
ajcBaseEventInt;
  String GetPressureCap.getPresschgEvent.TrackingInfo;
public String GetPressureCap.getPresschgEvent.getTrackingInfo(){
    return TrackingInfo;
  }
public void GetPressureCap.getPresschgEvent.setTrackingInfo(String s){
    TrackingInfo = s;
  }……
/******code weaving GetPressureCap.getPressPresschgPlan**********/
  declare parents:GetPressureCap.getPressPresschgPlan implements
    ajcBasePlanInt;
  String GetPressureCap.getPressPresschgPlan.tracking;
  boolean GetPressureCap.getPressPresschgPlan.firstEvent = true;
  ajcBaseEventInt GetPressureCap.getPressPresschgPlan.handled_event;
public String GetPressureCap.getPressPresschgPlan.getTracking(){
    return tracking;
  }
public void GetPressureCap.getPressPresschgPlan.setTracking(String s){
    tracking = s;
  }
public boolean GetPressureCap.getPressPresschgPlan.isFirst(){
    return firstEvent;
  }
public void GetPressureCap.getPressPresschgPlan.setFirst(boolean b){
    firstEvent = b;
  }
public ajcBaseEventInt GetPressureCap.getPressPresschgPlan.getEvent(){
    return handled_event;
  }
public void
GetPressureCap.getPressPresschgPlan.setEvent(ajcBaseEventInt e){
    handled_event = e;
  }……
}
```

Figure 23. Introduction aspect

Then the FDIR system was compiled together with the `aspects`, including the relation-modifying aspect mentioned above and the one containing the `pointcuts`.[3] When the compiled system was run, a tracking tree, as shown in Figure 24, was displayed.



```
Tracking Window
Processes Tracking
Event LeakManager.DisplayLeakEvent
Event LeakManager.Leak
    Agent LeakManager
        Plan LeakManager.LeakAlarmPlan
            Event LeakManager.Ma...
                Agent LeakManager
                    Plan LeakManager.ManageLeakAlarmPlan
                        Event GetPressureCap.getPresschgEvent
                        Event Isolator.leakIsolEvent
                        Event LeakManager.ControlEstablish
FRCS-FU is leaking, finding out which part caused leak--Leak Isolated. 1-2-TANK-LEG caused leak.
                            Agent LeakManager
                                Plan LeakManager.ControlEstablish_dummyPlan--faile
                                Plan LeakManager.ControlEstablishPlan
```

Figure 24. A sample Tracking Tree

[3] Assume that the sample system has been compiled by the JACK compiler to generate Java code.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have explored general issues in agent-based FDIR systems, such as monitoring and recording the change of world's states in agents' belief, rule-based action-triggering, and composing plans based on decision trees. We have also studied the communication and interoperability among heterogeneous agents in a limited scale.

Furthermore, we have researched how to provide effective explanation component for an agent-oriented system. We have designed an agent reasoning tracking mechanism that tracks the sequence of events occurring in an agent system, as well as theplans used to handled the events and agents involved;these are dynamically displayed that to the use in form of a tree. By exploring the tree of events, plans and agents, the user can obtain more knowledge (problem-solving context, discourse history, etc.) of the system's behavior and understands the system better. We have also designed an approach for the system developers to provide domain specific and system dependent information for the tracking mechanism to use and shows to the users even more details about the system's activity.

In addition, we have done research on automatic insertion of the tracking mechanism to arbitrary agent systems. One way to do this is to use a preprocessing tool to parse the original code for an agent system and then insert calls to the tracking system into proper places. However, such a preprocessing tool is error prone, as it must deal with various programming styles by different programmers. Thus, we studied the possibility of using aspect-oriented programming paradigm. We have not only found a way to use aspect-oriented technique to automate the insertion of the tracking mechanism, but also obtained valuable experience of using aspect-oriented with agent-oriented paradigm together to achieve cleaner and more organizable programming. This experience can help us as well as other researchers in further research.

As shown above, the tracking mechanism can be reused, that is, plugged in to other JACK systems with little effort. However, the approach of using AspectJ on JACK systems is only reusable in limited situations, for instance, when one wants to build aspects related to event postings. Even then, the `advice` might have to be revised.

In the future, it should be possible to find a general approach which enables AspectJ to be used to program aspects easily in terms of JACK constructs. A possible way to do that is to develop a `pointcut` library that contains `pointcuts` capturing all critical `join points` in a typical JACK system (for example, `pointcut @subtask(..)` captures the join points that an event is posted using `@subtask(..)`); then system developer can reuse those pointcuts by referencing the library and write their own `advice`.

In conclusion, the principle contributions of this work are:

1. We have studied and created general rules for performing the FDIR tasks using agent-based technique. A platform-independent, agent-oriented, automated FDIR system can assist astronauts and flight controllers in performing fault detection, isolation, and recovery.
2. We have studied and designed a tracking mechanism for providing effective explanation to an agent system's behavior.
3. We have generalized the tracking mechanism so that other JACK agent systems can use this mechanism to track and display system activity. A preprocessing tool can be used for automatic insertion of the generalized

tracking mechanism. This allows separate development of the tracking mechanism and the agent system, and allows addition of an effective explanation component to an existing agent system with little effort.

4. We have studied how to use aspect-oriented and agent-oriented paradigm to obtain better performance. A plug-and-play agent system tracking mechanism have been designed using aspect-oriented paradigm. Our experience can help others further explore the possibility of the two different paradigms together. In the future, we are to continue our research on that.

REFERENCES

[1] M. P. Georgeff and F. F. Ingrand, "Final report, phase 2, Research on procedural reasoning systems," Artificial Intelligence Center, SRI International, Menlo, Park, CA, June 1990.

[2] M. Georgeff and F. Ingrand, “Decision-making in an embedded reasoning system”, presented at International Joint Conference on Artificial Intelligence, Detroit, MI, 1989.

[3] M. Georgeff and F. Ingrand, “Real-time reasoning: the monitoring and control of spacecraft systems”, presented at the Sixth IEEE Conference on Artificial Intelligence Applications, Santa Barbara, CA 1990.

[4] A. Cawsey, "Developing an explanation component for a knowledge-based system:discussion," *Expert Systems with Applications*, vol. 8, pp. 527-531, 1995.

[5] F. Chen, and R. Volz (2003), “Tracking and understanding automated space shuttle fault detection, isolation and recovery”, in proceedings of International Conference on Artificial Intelligence (IC-AI’03). Las Vegas, NV June 2003.

[6] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge, "A formal specification of dMARS," *Intelligent Agents IV*, vol. 1365, pp. 155-176, 1998.

[7] K. L. Myers, “A procedural knowledge approach to task-level control”, presented at the Third International Conference on AI Planning Systems, Edinburgh, Scotland, May 1996.

[8] M. Georgeff and A. Lansky, “Procedural knowledge”, the IEEE Special Issue on Knowledge Representation, vol. 74, pages 1383-1398, 1986

[9] M. Georgeff and A. Lansky, “A procedural logic”, presented at International Joint Conference on Artificial Intelligence, Los Angeles, CA, 1985.

[10] A. S. Rao and M. Georgeff, "BDI agents: From theory to practice," presented at First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, 1995.

[11] M. Wolverton, "Presenting significant Information in expert system explanation", presented at Seventh Portugese Conference on Artificial Intelligence (EPIA95), Funchal, Madeira Island, Portugal, 1995.

[12] Y. Shoham, “Agent oriented programming”, *Journal of Artificial Intelligence*, vol. 60, pp. 51-92, 1993.

[13] N. Howden, R. Rönnquist, A. Hodgson, A. Lucas, “JACK intelligent agents – summary of an agent infrastructure”, presented at the Fifth International Conference on Autonomous Agents, Montreal, Canada, 2001.

[14] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, pp. 29-32, 2001.

[15] R. Walker, E. Baniassad, G. Murphy, “An initial assessment of aspect-oriented programming”, presented at the 21st International Conference on Software Engineering**,** Los Angeles, CA, May1999

[16] C. V. Lopes, "AOP: A historical perspective", In *Aspect-Oriented Software Development*, Addison-Wesley, 2003

[17] C. Constantinides, A. Bader, T. Elrad, P. Netinant, M. Fayad, “Designing an aspect-oriented framework in an object-oriented environment”, *ACM Computing Surveys (CSUR),* vol. 32, no. 1es, pp. 41, 2000

[18] G. Kiczales, “Aspect-oriented programming”, *ACM Computing Surveys (CSUR),* vol. 28, no. 4es, pp. 154, 1996.

[19] K. Sycara, S. Widoff, M. Klusch and J. Lu, “LARKS: Dynamic matchmaking among heterogeneous software agents in cyberspace", *Autonomous Agents and Multi-Agent Systems*, vol. 5, pp. 173–203, 2002.

[20] K. Sycara, J. Lu, M. Klusch, and S. Widoff, “Dynamic service matchmaking among agents in open information environments”, *Journal ACM SIGMOD Record* , Special Issue on Semantic Interoperability in Global Information Systems, vol. 28, pp. 47-53, 1999.

[21] K. Sycara, J. Lu, M. Klusch, and S. Widoff, “Matchmaking among heterogeneous agents in the internet”, presented at AAAI Spring Symposium on Intelligent Agents in Cyberspace, Stanford, CA, 1999.

APPENDIX

A.1 Rules for tracking code insertion

This section contains an excerpt from a previous report to the research sponsor that describes the rules for traking code-insertion in greater detail.

A Preprocessor is used to automatically insert necessary code into JACK files so that the modified JACK systems will display the activities of itself to the users when it runs. When starting, the preprocessor will find all the JACK files that it needs to modify, and then use the parsers to analyze and insert the necessary code into the proper places.

The event parser will insert reference code to import the tracking agent and events, and add an extra parameter called TrackingInfo to each event.

The agent parser will insert the same reference code, and a declaration that two events, TrackingEvent and returnValueEvent, can be sent by the agent. If there is any event posted within the agent file[4], code will be inserted to generate the tracking information for the event the agent is posting.

The capability parser is similar to the agent parser except that it need not worry about the initialization of tracking information since no event will be posted within Capabilities.

The plan parser does the toughest work. It inserts reference code to import the tracking agent and events and declaration code as described above; it inserts code that maintains local tracking information within itself; and, it assigns proper tracking information to any event it posts. It also inserts code that creates instances of TrackingEvent and

[4] All events could be said to be posted by the agents, whether they are posted within a plan or a database or elsewhere; but here we are talking about the case in which there is a method of an agent, within which an event is posted.

returnValueEvent carrying proper information and sends them to TrackingAgent. Examples will be shown below.

A.1.1 Event parser

Event parser has three main tasks. First, it inserts the code referring to the TrackingAgent package right before entering the event class. Second, it inserts code declaring the extra field for the event, String TrackingInfo. Third, it inserts code that initializes the extra field in the event's posting function.

The following is an example showing how an event is modified by the parser.

```
package GetPressureCap;

public event getPressPresschgEvent extends BDIGoalEvent{
  String name;
  Boolean decreasing;
  Double pressure;
  String initialDisplay;
  String returnDisplay = "#name# pressure = #pressure#,
      pressure decreasing = #decreasing#";



     #posted as getValue(String n) {
       this.name = n;
       initialDisplay = "get pressure for " + name + ", and
   direction of change.";
     }
   }
```

Code 1. An event before modification

```
package GetPressureCap;

import TrackingAgent.*;//Added by Tracking-PreProcessor

public event getPressPresschgEvent extends BDIGoalEvent{

/********Added by Tracking-PreProcessor********/
String TrackingInfo;
logical String $TrackingInfo;
/********Added by Tracking-PreProcessor********/

 String name;
 Boolean decreasing;
 Double pressure;
 String initialDisplay;
 String returnDisplay = "#name# pressure = #pressure#, pressure decreasing = #decreasin
 #posted as getValue(String t//Added by Tracking-PreProcessor
,String n) {
/********Added by Tracking-PreProcessor********/
TrackingInfo = t;
/********Added by Tracking-PreProcessor********/

this.name = n;
initialDisplay = "get pressure for " + name + ", and direction of change.";
}
...
}
```

Code 2. An event after modification

If the event is an automatic event, i.e., it has a "#posted when methodName(…)" posting function, the parser will inserts the following code

```
            /********Added by Tracking-PreProcessor********/
              TrackingInfo = "" + TrackingAgent.NextNum();

            /********Added by Tracking-PreProcessor********/
```

where NextNum() is a static method of the TrackingAgent, which will generate sequential numbers beginning at 1. The method NextNum() will be called when an event is posted directly within an executive agent, or an automatic event is posted, to generate the tracking information for the event posted. NextNum() needs to be called at these points because the corresponding event starts a new tracking subtree from the outermost level; it must therefore have an index number incremented to distinguish it from other root level events. Below is the NextNum() method defined in TrackingAgent.agent.

```
private static int num = 0;

public static int NextNum(){
     return ++num;
}
```

A.1.2 Agent Parser

An agent parser also has three tasks to do. First, it inserts code that refers to the TrackingAgent package so that the tracking agent may be referenced. Second, it inserts code that declares that the TrackingEvent and returnValueEvent are sent by the agent; these events are used in the agent's plans to sent information to the tracking agent, but must be declared in the agent itself as well as the plan. Third, the agent parser inserts code that initializes tracking information by calling TrackingAgent.NextNum() and assign the tracking information to the event if any event is posted within the agent. Code 3 and Code 4 shows an agent before and after being modified by the agent parser.

```
package LeakManager;

import aos.jack.util.thread.BeliefReflection;
import aos.jack.jak.agent.Agent;
import aos.jack.jak.cursor.Cursor;
...
public agent LeakManager extends Agent{

   #global data PARTS parts("partOf.txt");
   #global data TYPES types("typeOf.txt");
   #private data DisplayLeak dl();
   #global data Who w();
   #private data MessageDialog messageDialog();
   #sends event leakIsolEvent lie;
   #posts event Leak le;
   ...
   #handles event ManageLeakAlarm mla;
   #uses plan ManageLeakAlarm_dummyPlan;     //always fails; for testing
   #uses plan ManageLeakAlarmPlan;
   #has capability GetPressure getP;

   String Name;
 ...
 public void display(String p){
         postEventAndWait(DL.leaking(p));
  }
}
```

Code 3. An agent before modification

```
package LeakMana
 ..
import TrackingAgent.*;//Added by Tracking-I
 public agent LeakManager extenc

#sends event TrackingEvent _tracking_Event;//Added by Track.
#sends event returnValueEvent _return_Value_Event;//Added by Trac
#global data PARTS parts("partC
#global data TYPES types("typeC
#private data DisplayLeak
#global data Who
#private data MessageDialog message
#sends event leakIsolEver
#posts event Leak
..
#handles event ManageLeakAla
#uses plan ManageLeakAlarm_du
#uses plan ManageLeakAla
#has capability GetPressur

 String Na
..
 public void display(Str

 postEventAndWait(DL.leaking("" + TrackingAgent.NextNum(),//added by Tr
 p))
 }
 }
```

Code 4. An agent after modification

A.1.3 Capability Parser

The capability parser works similarly to the agent parser, except that it need not worry about initialization of tracking information.

A.1.4 Plan Parser

The plan parser handles the toughest tasks. The plan parser inserts code that maintains a proper tracing information local to the plan itself and passes the tracking information to each event posted or sent within the plan. As a reminder, two methods of TrackingEvent help solve this problem. The method append(String trackingInfo) appends trackingInfo by ".1", for instance, append("1.2") results in "1.2.1". The other method increment(String trackinginfo) increments the last number (if only one number, then increments that number) by 1, for instance, increment("2.3.5") results in "2.3.6". The strategy for a plan to maintain a proper local tracking information is that each plan maintains a local variable, called _traking_info; at the very beginning of the plan body, this variable will be initialized to append(current_event.TrackingInfo), where current_event is the event handled by this plan; and each time an event is initialized(created) within this plan, _tracking_info is incremented by calling increment(_tracking_info), and the incremented value will be passed as TrackingInfo of the event being posted.[5]

First, the plan parser inserts code for referencing the tracking agent, as others do, right before entering the plan class. Second, right after entering the plan class it inserts code declaring that it will send the two events, and code declaring the variable _tracking_info; it also inserts code that defines the name for the TrackingAgent, which later will be used for sending the two events to the TrackingAgent. Third, the parser inserts code at the beginning of the plan body to initialize _traking_info, and sends to TrackingAgent an instance of TrackingEvent carrying proper information about current system activity, including proper tracking information. Fourth, it inserts code at proper places that maintains _tracking_info and passes it to the event wherever an event is posted. Code 5 and Code 6 show an example.

[5] The parser assumes that whenever an event is instantiated, it will then be posted.

```
package Isolator;
...
plan leakIsolPlan extends Plan {
  #reads data Who w;
  #uses data MessageDialog messageDialog;
  #handles event leakIsolEvent lie;
  #sends event Response re;
  #posts event getPressPresschgEvent pressPressChng;
  #posts event getPresschgEvent pressChng;
  #posts event secureEvent sec;
  ...
  #reasoning method
    descreasingOrBelowLimit(logical String what,double limit)
    {
        String sstring = lie.rcs + " Leaking----";
        getPressPresschgEvent ev = pressPressChng.getValue(sstring +  what.toString()
        @subtask(ev);
        ev.isDescreasing() || ev.getPressure() < limit;     // Fails method if false.
    }
    #reasoning method
    increasingAndAboveLimit(logical String what, double limit)
    {
        String sstring = lie.rcs + " Leaking----";
        getPressPresschgEvent ev = pressPressChng.getValue( sstring + what.toString()
        @subtask(ev);
        ev.isDescreasing() == false && ev.getPressure() >= limit;     // Fails method
false.
    }
body() {
String leakingPart;
 @subtask(sec.secure());
 if (increasingAndAboveLimit( $manf1, Limit) ) {
        if ( descreasingOrBelowLimit( $manf2, Limit) ) {
          leakingPart = $manf2.toString() ;
        }
        else {
            if ( increasingAndAboveLimit( $manf3, Limit) ) {
                                    ...
  @wait_for(messageDialog.display(leakingPart));
  @reply(lie, re.response());
  }
}
```

Code 5. A plan before modification

```
package Isolator;
...
import TrackingAgent.*;//Added by Tracking-PreProcessor
import aos.jack.util.thread.BeliefReflection;
plan leakIsolPlan extends Plan {
#sends event TrackingEvent _tracking_Event;//Added by Tracking-PreProcessor
#sends event returnValueEvent _return_Value_Event;//Added by Tracking-
PreProcessor
static String _tracking_info;//Added by Tracking-PreProcessor
 String trackingAgent = "Tracker";//Added by Tracking-PreProcessor
 #reads data Who w;
 #uses data MessageDialog messageDialog;
 #handles event leakIsolEvent lie;
 #sends event Response re;
 #posts event getPressPresschgEvent pressPressChng;
 #posts event getPresschgEvent pressChng;
 #posts event secureEvent sec;
...
 #reasoning method descreasingOrBelowLimit(logical String what,double
limit) {
 String sstring = lie.rcs + " Leaking----";
 getPressPresschgEvent ev = pressPressChng.getValue(_tracking_info =
_tracking_Event.increment(_tracking_info),//Added by Tracking-PreProcessor
sstring + what.toString());
 @subtask(ev);
 ev.isDescreasing() || ev.getPressure() < limit;
 }
 #reasoning method increasingAndAboveLimit(logical String what, double
limit) {
 String sstring = lie.rcs + " Leaking----";
 getPressPresschgEvent ev = pressPressChng.getValue(_tracking_info =
_tracking_Event.increment(_tracking_info),//Added by Tracking-PreProcessor
 sstring + what.toString());
 @subtask(ev);
 ev.isDescreasing() == false && ev.getPressure() >= limit;
 }
 body() {
/********Added by Tracking-PreProcessor********/
String agnt = getAgent().name();//Added by Tracking-PreProcessor
@send(trackingAgent, _tracking_Event = _tracking_Event.tracking
(lie.TrackingInfo,"leakIsolEvent","leakIsolPlan",agnt,lie.initialDisplay));
 _tracking_info = _tracking_Event.append(lie.TrackingInfo);//Added by
Tracking-PreProcessor

/********Added by Tracking-PreProcessor********/
 String leakingPart;
 @subtask(sec.secure(_tracking_info = _tracking_Event.increment
(_tracking_info)//Added by Tracking-PreProcessor
));
 if (increasingAndAboveLimit( $manf1, Limit) ) {
 if ( descreasingOrBelowLimit( $manf2, Limit) ) {
 leakingPart = $manf2.toString() ;
 }
 else {
 if ( increasingAndAboveLimit( $manf3, Limit) ) {
 ...
 @wait_for(messageDialog.display(leakingPart));
 @reply(lie, re.response(_tracking_info = _tracking_Event.increment
(_tracking_info)//Added by Tracking-PreProcessor
));
}
```

Code 6. A plan after modification

A.1.5 Value Display System

In the value display system, the same TrackingAgent is used; it handles an event called returnValueEvent, using a plan returnValuePlan. ReturnValueEvents are sent from each of the executive plans, carrying information about current agent belief, and an extra parameter, TrackingInfo, just as TrackingEvents do. The plan returnValuePlan will decode TrackingInfo to find the right nodes to associate with the information carried by returnValueEvent.

To automatically insert code that deals with value display, some requirement must be set upon the users of the preprocessor. In the events of their agent system, two extra fields must be defined. One is called initialDisplay, which generally shows the goal of the current event. The other, called returnDisplay, generally shows the result after the event has been handled successfully. When these have both been defined, the value display system is able to display the current system belief. ReturnDisplay should be defined in the following form if any field of the event is going to appear in it.

```
String returnDisplay = “ label1 = #field1#  lable2= #field2#”,
```

where the names between the pairs of # symbols must match the declared name of a variable to be displayed. Moreover, the variables may only be typed String, Integer, Long, Float or Double. Note that these are the class types, not the primitive types.

The event parser will then translate this form into a readable form, and put it into a method added to each event by the event parser itself, called trackValue(). This method returns the value of returnDisplay. Code that calls trackValue() will be inserted by the plan parser into each plan, right before the plan ends its execution (either succeeds or fails), to get the value of returnDisplay. This call has to be made at the end of the plan's execution, since some of the event's fields might not be initialized until then.

Since the plan parser, like its peers, works without a grammar of JACK and Java, it is difficult for it to find the exact end of plan execution, since there are plenty of ways in which a plan could end its execution at many different points in a plan. However, AOS has developed new features, which solve this problem. In the current release of JACK, a plan has two member methods, called pass() and fail(). If the plan succeeds, the former will be executed immediately before the plan returns, the latter will be executed before the plan returns otherwise. Both methods can be overloaded by the user to include the code to be executed.

Code 7 and Code 8 shows an event example and a plan example.

```
package GetPressureCap;
import TrackingAgent.*;//Added by Tracking-PreProcessor
public event getPressPresschgEvent extends BDIGoalEvent{

/********Added by Tracking-PreProcessor********/
String TrackingInfo;
logical String $TrackingInfo;
/********Added by Tracking-PreProcessor********/

String name;
 Boolean decreasing;
 Double pressure;
 String initialDisplay;

 String returnDisplay = "#name# pressure = #pressure#,
                pressure decreasing = #decreasing#";

 #posted as getValue(String t//Added by PreProcessor
,String n) {
/********Added by Tracking-PreProcessor********/
TrackingInfo = t;
/********Added by Tracking-PreProcessor********/

this.name = n;
initialDisplay = "get pressure for " + name + ",
              and direction of change.";

 }

  ...
 /********Added by Tracking-PreProcessor********/
public String trackValue(){
     return ""+name.toString()+" pressure =
      "+pressure.toString()+",
      pressure decreasing = "+decreasing.toString()+"";
}
}
```

Code 7. An Event After Modification

```
package Isolator;
...
import TrackingAgent.*;//Added by PreProcessor
import aos.jack.util.thread.BeliefReflection;
plan leakIsolPlan extends Plan {
#sends event TrackingEvent _tracking_Event;//by PreProcessor
#sends event returnValueEvent _return_Value_Event;// by PreProcessor
static String _tracking_info;//Added by PreProcessor
 String trackingAgent = "Tracker";//Added by PreProcessor
...
 #reasoning method increasingAndAboveLimit(logical String what,double
limit){
 String sstring = lie.rcs + " Leaking----";
 getPressPresschgEvent ev = pressPressChng.getValue(
_tracking_info=_tracking_Event.increment(_tracking_info)//by PrePro
 sstring + what.toString());
 @subtask(ev);
 ev.isDescreasing() == false && ev.getPressure() >= limit;
 }
 body() {
/********Added by Tracking-PreProcessor********/
String agnt = getAgent().name();//Added by Tracking-PreProcessor
@send(trackingAgent,_tracking_Event=_tracking_Event.tracking(
lie.TrackingInfo,"leakIsolEvent","leakIsolPlan",
agnt,lie.initialDisplay));
_tracking_info = _tracking_Event.append(lie.TrackingInfo);
/********Added by Tracking-PreProcessor********/
 String leakingPart;
 @subtask(sec.secure(
_tracking_info=_tracking_Event.increment(_tracking_info)//by PrePro
));
 if (increasingAndAboveLimit( $manf1, Limit) ) {
 if ( descreasingOrBelowLimit( $manf2, Limit) ) {
 leakingPart = $manf2.toString() ;
 }
 else {
 if ( increasingAndAboveLimit( $manf3, Limit) ) {
 ...
 @wait_for(messageDialog.display(leakingPart));
 @reply(lie, re.response(
_tracking_info=_tracking_Event.increment(_tracking_info)//by PrePro
));
}
/********Added by Tracking-PreProcessor********/
#reasoning method pass(){
   @send(trackingAgent, _return_Value_Event.returned
      (lie.TrackingInfo,lie.trackValue()));
}
#reasoning method fail (){
   @send(trackingAgent, _return_Value_Event.returned
     (lie.TrackingInfo,"a plan failed."));
}
/********Added by Tracking-PreProcessor********/
}
```

Code 8. A plan after modification

A.2 The Two base Interfaces for Events and Plans to implement

```
//All events need implement this interface in order to be tracked by the
//aspect-based tracking mechanism

public interface ajcBaseEventInt {
   public String getTrackingInfo();
   public void setTrackingInfo(String s);
   public String getResult();
   public String getGoal();
}
```

```
//All plans need implement this interface in order to be tracked by the
//aspect-based tracking mechanism

public interface ajcBasePlanInt {
  public String getTracking();
  public void setTracking(String s);
  public boolean isFirst();
  public void setFirst(boolean b);
  public ajcBaseEventInt getEvent();
  public void setEvent(ajcBaseEventInt e);
}
```

A.3 The Aspect with Pointcuts for Aystem Activity Tracking

The following is an `aspect` that contains the `pointcuts` which construct the system activity tracking mechanism, including capture related `join points`, constructing tracking window and tracking tree.

```
import aos.jack.jak.event.Event;
import aos.jack.jak.task.Task;
import aos.jack.jak.plan.Plan;
import aos.jack.jak.agent.Agent;
import aos.jack.jak.plan.PlanFSM;
import aos.jack.jak.fsm.FSM;
import aos.jack.jak.fsm.MaintainFSM;
import java.util.StringTokenizer;




aspect EventTracking_display {

  /*** for display ****/
  TrackDisplayer track;

  pointcut main(): execution(public static void main(..));
```

```
        before() : main(){
           track = new TrackDisplayer();
        }



        pointcut handled_event(Event e, Plan p): execution(public Plan createPlan(Event,
Task)) && args(e, ..) && target(p);



        /****** this pointcut initiates current plan's tracking to
TrackingInfo of the event it handles
                when a plan is created but before the plan really exe-
cutes *****/
        after(Event e, Plan p) returning(Plan pa): handled_event(e, p){
           if( p instanceof ajcBasePlanInt && e instanceof
ajcBaseEventInt){
              ajcBasePlanInt pai = (ajcBasePlanInt)pa;
              pai.setEvent ((ajcBaseEventInt)e);
           }
        }


        /****this pointcut captures the points when a plan enters its body, i.e., begin its
execution.
              Nodes for events/plans/agents are added to the tracking treee at this point.
************/
```

```
            before(ajcBasePlanInt pai): execution(PlanFSM body()) && target(pai){
                Plan pa = (Plan)pai;
               // System.out.println("=====Enter body()============");
                System.out.println("======" + "Event " + pa.handledEvent() + "====" +
pai.getEvent().getTrackingInfo() + "=======");
                pai.setTracking (Tracker.append(pai.getEvent().getTrackingInfo()));
                String planName = pa.toString();
                String agentName = pa.getAgent().name() ;
                StringTokenizer st = new StringTokenizer(planName);
                track.addNodes(pai.getEvent().getTrackingInfo(), "Event " +
pa.handledEvent(),
                  "Agent " + agentName, "Plan " + st.nextToken(), pai.getEvent().getGoal());
               // System.out.println("the event handled by current plan is " +
pai.getEvent().getTrackingInfo());
            }


            /***** this pointcut captures the points when an event is posted 1)within an
agent by the agent member method postEvent();
                 2)automatic posting ;    3) posting within beliefSets
****************************************/
            /***** the TrackingInfo of the event being posted is set
**********************************************/
```

```
        before(ajcBaseEventInt ev) : call(public * Agent.postEvent(..)) &&
args(ev,..)&& !this(PlanFSM){//this one is working   --11/25/02
            ev.setTrackingInfo(Tracker.NextNum() + "");
          //System.out.println("the tracking information of current event " + ev.toString()
+ " is " + ev.getTrackingInfo());
        }



        /******** this pointcut captures the points when an event is posted within an
agent by postEventAndWait() ******/
        /***** the TrackingInfo of the event being posted is set
*********************************************/
        before(ajcBaseEventInt ev) : call(public * Agent.postEventAndWait(..)) &&
args(ev,..)&& this(Agent){//this one is working   --11/25/02
            ev.setTrackingInfo(Tracker.NextNum() + "");
          // System.out.println("the tracking information of current event------ " +
ev.toString() + " is " + ev.getTrackingInfo());
        }



        /******** this pointcut captures the points when an event is posted within a
plan by @post() *************/
```

```
        /***** the TrackingInfo of the event being posted is set
*********************************************/
        after(ajcBaseEventInt ev, PlanFSM fsm) returning(): call(public *
Agent.postEvent(..)) && args(ev,..) && this(fsm)  { //this one is working   --12/03/02
            ajcBasePlanInt pa = (ajcBasePlanInt)fsm.getPlan();
            if(pa.isFirst()){
              ev.setTrackingInfo(pa.getTracking());
              pa.setFirst(false);
            }
            else {
              pa.setTracking(Tracker.increment(pa.getTracking()));
              ev.setTrackingInfo(pa.getTracking());
              }
          // System.out.println("the tracking information of current event " + ev.toString()
+ " is " + ev.getTrackingInfo());
        }


        /******* this pointut captures the points when an event is posted within a plan
by @send()   ********/
        /***** the TrackingInfo of the event being posted is set
*********************************************/
         before(ajcBaseEventInt ev, PlanFSM fsm): call(public * Agent.send(..)) &&
args(.., ev) && this(fsm)  {
```

```
            ajcBasePlanInt pa = (ajcBasePlanInt)fsm.getPlan();
            if(pa.isFirst()){
              ev.setTrackingInfo(pa.getTracking());
              pa.setFirst(false);
              //System.out.println("--------First: tracking information of the Message
Event" + ev.toString() + " is " + ev.getTrackingInfo());
            }
            else {
              pa.setTracking(Tracker.increment(pa.getTracking()));
              ev.setTrackingInfo(pa.getTracking());
              //System.out.println("--------Increment: tracking information of the Mes-
sage Event" + ev.toString() + " is " + ev.getTrackingInfo());
            }
          }



        /******** this pointcut captures the points when an event is posted within a plan
by @subtask() **********/
        /***** the TrackingInfo of the event being posted is set
**********************************************/
        pointcut pushCall(): call(public * Task.push(FSM)) ;
        after(ajcBaseEventInt ev, PlanFSM fsm) returning(): pushCall() && args(ev) &&
this(fsm)  {
            ajcBasePlanInt pa = (ajcBasePlanInt)fsm.getPlan();
            if(ev.getTrackingInfo() == null){
             if(pa.isFirst()){
               ev.setTrackingInfo(pa.getTracking());
               pa.setFirst(false) ;
             }
             else {
               pa.setTracking(Tracker.increment(pa.getTracking()));
               ev.setTrackingInfo(pa.getTracking());
             }
            }
           System.out.println("?????????the tracking information of current event " +
ev.toString() + " is " + ev.getTrackingInfo());
        }




        /******** this pointcut captures the points when an event is posted within a
plan by @insist() or achieve() ********/
        /***** the TrackingInfo of the event being posted is set
*********************************************/
```

```
        after(PlanFSM fsm) returning(FSM f): execution( public FSM genFSM(int)) &&
target(fsm)  {   //working as Feb. 24, 2003
            // System.out.println("!!!!!!!!!TestEvent has been caught in genFSM!!!!!!!!" );
             ajcBasePlanInt pa = (ajcBasePlanInt)fsm.getPlan();
             ajcBaseEventInt ev = (ajcBaseEventInt)f;
             if(pa.isFirst()){
               ev.setTrackingInfo(pa.getTracking());
               pa.setFirst(false) ;
             }
             else{
               pa.setTracking(Tracker.increment(pa.getTracking()));
               ev.setTrackingInfo(pa.getTracking());
             }

            //System.out.println("!!!!!!!!!!!!!!!!!!!!!the tracking information of current
event " + ev.toString() + " is " + ev.getTrackingInfo());
        }



        /***** this pointcut captures the points when a plan successfully complete its
execution and returns ********/
        /***** the result of the plan's execution is to be displayed
**********************************************/
```

```
            before(ajcBasePlanInt pa): call(* pass()) && target(pa){
                System.out.println("call pass()...");
                track.returnValueDisplay(pa.getEvent().getTrackingInfo(),
pa.getEvent().getResult());


            }


            /***** this pointcut captures the points when a plan fails and returns
********/
            /***** the fact that the plan fails is to be displayed
***********************/
            before(ajcBasePlanInt pa): call(* fail()) && target(pa){
                String planName = pa.toString();
                StringTokenizer st = new StringTokenizer(planName);
                System.out.println("call fail()... in Plan " + st.nextToken());
                track.returnValueDisplay(pa.getEvent().getTrackingInfo(), "--failed. try an-
other plan :(");


    }
}
```

VITA

Feilong Chen

2155 Cram Place Apt. 26, Ann Arbor, MI 48105

EDUCATION

M.S. in computer engineering, Texas A&M University, College Station, TX

08/03

B.S. in biochemistry, China Agricultural University, Beijing, China 07/09

EXPERIENCE

Research Assistant Department of Computer Science, Texas A&M University 08/01-08/03 Worked on projects funded by United Space Alliance

Session Co-chair, the Seventh World Multi-Conference on Systemics, Cybernetics, and Informatics (SCI 2003). Orlando, FL, 2003.

PUBLICATIONS

F. Chen and R. Volz (2002), "Intelligent assistant for space shuttle diagnostics" in proceedings of Mission Systems 2002: The Annual Conference for Technology in Space Operations, Houston, TX, May 2002.

F. Chen, and R. Volz (2003), "Tracking and understanding automated space shuttle fault detection, isolation and recovery", in proceedings of International Conference on Artificial Intelligence (IC-AI'03). Las Vegas, NV June 2003.

Feilong Chen and Richard Volz (2003), "Tracking automated shuttle fault isolation through use of aspects", in proceedings of the Seventh World Multi-Conference on Systemics, Cybernetics, and Informatics (SCI 2003). Orlando, FL, 2003.