TDR System - A Multi-Level Slow Intelligence System for Personal Health Care

Shi-Kuo Chang, JunHui Chen, Wei Gao and Qui Zhang
Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15238, USA
{schang, juc52, weg21}@pitt.edu, adora91@gmail.com

***Abstract*—This paper describes the design of an experimental multi-level slow intelligence system for health care, called the TDR system, consisting of interacting super-components each with different computation cycles specified by an abstract machine model. The TDR system has three major super-components: Tian (Heaven), Di (Earth) and Ren (Human), which are the essential ingredients of a human-centric psycho-physical system following the Chinese philosophy. Each super-component further consists of interacting components supported by an SIS server. This experimental TDR system provides a platform for exploring and integrating different applications in personal health care, emergency management and social networking.**

Keywords*—*slow intelligence system, distributed sensor networks, component-based software engineering.

1. Introduction

Recently there are growing interests in human-centric psycho-physical systems, especially in health care applications. Such human-centric psycho-physical systems have two common characteristics. From the decision-theoretic viewpoint these systems usually have multiple decision cycles such that the actions of slow decision cycle(s) may override the actions of quick decision cycle(s), resulting in poorer performance in the short run but better performance in the long run. From the architectural viewpoint these systems usually have multiple levels to monitor, control and manage many sensors and actuators.

The slow intelligence system is an approach to design such human-centric psycho-physical systems. A slow intelligence system (SIS) is a system that *(i) solves problems by trying different solutions, (ii) is context-aware to adapt to different situations and to propagate knowledge, and (iii) may not perform well in the short run but continuously learns to improve its performance over time.* The general characteristics of a slow intelligence system include enumeration, propagation, adaptation, elimination, concentration and multiple decision cycles [1]. In our previous work, an experimental test bed was implemented that allows designers to specify interacting components for slow intelligence systems [2].

DOI reference number: 10.18293/SEKE2016-054

To facilitate the design of complex slow intelligence systems such as human-centric psycho-physical systems, the concept of super-components is formulated [3]. A complex slow intelligence system basically consists of interacting super-components, which further consists of many interacting components supported by an SIS server. Communications in SIS are through the SIS server and the messages are *layered*, i.e., each message type has its hierarchical *scope*. A super-component can thus be viewed as a collection of components interacting by messages within the same scope. From an architectural viewpoint the result is a multi-level slow intelligence system as illustrated by Figure 1.1.

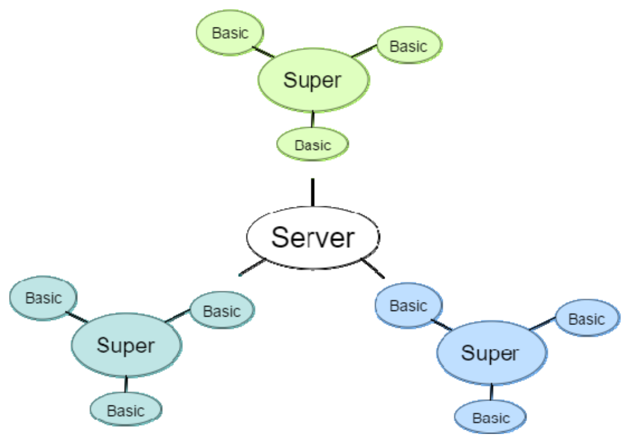

Figure 1.1. A multi-level slow intelligence system.

This paper describes the design of an experimental multi-level slow intelligence system for health care, called the TDR system, which mainly consists of three super components: Tian, Di and Ren. According to the Chinese philosophy these three super-components are the essential ingredients of a human-centric psycho-physical system. They can be thought of as human beings (Ren) interacting with the environment consisting of heaven (Tian) and earth (Di). Decision making in TDR system is through multiple computation cycles involving the super components to increase the chances of survival of human beings. Any action based on only one aspect of the environment without considering the other aspects could reduce the chances of survival, thus iterative, multiple computation cycles are crucial for the TDR system.

The paper is organized as follows. Section 2 presents an abstract machine model for the computation cycles. The TDR system architecture is described in Section 3. The Tian super-component is described in detail in Section 4. Since the Ren super-component has been described in the first author's previous paper on slow intelligence system for health care [4], it will not be repeated here. A user-friendly GUI for the TDR system is described in Section 5. The TDR system was implemented in Java and GUI implemented in PHP. This test bed for TDR system thus offers an experimental platform for exploring and integrating different applications in personal health care, emergency management and social networking, some of which will be discussed in Section 6.

2. The Abstract Machine Model for Computation Cycles

As mentioned in Section 1 an SIS typically possesses at least two decision cycles. The first one, the *quick decision cycle*, provides an instantaneous response to environmental changes. The second one, the *slow decision cycle*, tries to follow the gradual changes in the environment and analyze the information acquired from the environments or peers or past experiences. The slow/quick decision cycles enable the SIS to both cope with the environment and meet long-term goals.

Complex SISs may possess multiple slow decision cycles and quick decision cycles. Most importantly, actions of slow decision cycle(s) may override actions of quick decision cycle(s), resulting in poorer performance in the short run but better performance in the long run.

To model such decision cycles we introduce an abstract machine model of multiple computation cycles in Section 2.1, and then specify the computation cycles for the TDR system in Section 2.2. In Section 2.3 we describe the steps to compile the abstract machine model into working components of the TDR system.

2.1. The Abstract Machine Model

The Abstract Machine Model is specified by: (P, S, P0, $Cycle^1$, ...,, $Cycle^n$), where

P is the non-empty problem set,
S is the non-empty solution set, which is a subset of Po,
P0 is the initial problem set, which is a subset of P,
$Cycle^1$, ...,, $Cycle^n$ are the computatin cycles.

Each computation cycle will start from an initial problem set and apply different operators such as **+*adap*$_{Aij}$=**, **-*enum*<**, **>*elim*-**, **=*prop*$_{Aij}$ +** and **>*conc*=** successively to generate new problem sets from old problem sets until a non-empty solution set is found. If a non-empty solution set is found, the cycle is completed and later the same computation cycle can be repeated. If on the other hand no solution set is found, a different computation cycle is entered.

As an example the problem set P consists of problem elements p1, p2, p3, ..., pn, and each problem element pj is specified by a vector consisting of attributes Aij. A computation cycle x will attempt to find a solution set by first adapting based upon input from the environment: P^x0 **+*adap*$_{Aij}$=** P^x1 is to adapt based on attribute Aij, for example, by appending Aij to each element in P^x0 to form P^x1. Then it may try to find related problem elements: P^x1 **-*enum*<** P^x2 where P^x2 = { y: y is related to some x in P^x1, e.g. d(x,y) < D}

Next it may try to eliminate the non-solution elements:
P^x2 **>*elim*-** P^x3 where P^x3 = {x: x is in P^x2 and x is in S}

Finally the solution elements (or alert messages if there are nosolutions) may be propagated to peers: P^x3 **=*prop*$_{Aij}$+** P^x4 is to export/propagate attribute Aij to peers.

Therefore this computation cycle can be specified succinctly as follows: $Cycle^x$ [guard x,y]: P^x0 **+*adap*$_{Aij}$=** P^x1 **-*enum*<** P^x2 **>*elim*-** P^x3 **=*prop*$_{Aij}$+** P^x4

The above expression is a specification of the computation cycle, not a mathematical equation. This expression should be read and interpreted from left to right.

If P^x4 is non-empty, the Abstract Machine will complete this cycle of computation and terminate at the end of $Cycle^x$, and it may later resume at the beginning of $Cycle^x$. Otherwise P^x4 is empty and the Abstract Machine will jump to a different $Cycle^y$. This is specified by [guard x,y] where x is the current computation cycle if a solution set is found (P^x4 is non-empty), and y is the computation cycle to enter if no solution set is found (P^x4 is empty). Before an Abstract Machine completes its current computation cycle, it will propagate the solution set (or alert messages) to its peers.

In the above, the elimination operator can be replaced by the concentration operator, whenever the solution set is not known apriori. The concentration operator applies a predefined threshold to filter out problem elements below the threshold:
P^x1 **>*conc*=** P^x2 where P^x2 = {x: x is in P^x1 and th(x) above a predefined threshold t}

2.2. Multiple Computation Cycles of TDR System

For the TDR system, a problem element is a combination of Tian, Di and Ren attributes. Those problem elements that are favorable for human survival are in the solution set S. The problem set P consists of problem elements p1, p2, p3, ..., p^n, and each problem element is specified by a vector consisting of the attributes from Tian (heaven), Di (earth) and Ren (human being), i.e.,

pj = (t1j, t2j, ..., d1j, d2j, ..., r1j, r2j, ...)

For example, the Tian attributes tij are atmospheric variables such as amount of sunlight and water level, the Di attributes dij are residential variables such as ambient temperature and humidity, and the Ren attributes rij are personal health indicators such as blood pressure, spo2 value, heart rate, etc.

pj = (sunlightj, waterlevelj, tempj, humidityj,
bloodpressurej, spo2valuej, heartratej)

Initially some attributes may not be assigned any value and some may already have pre-assigned values. After most attributes have been assigned values one can decide whether the problem element is in the solution set. (The simplest case is that each attribute Aij has a solution range Rj, and if every attribute Aij falls within the solution range Rj then the problem element pj is in the solution set S).

In the TDR system, there are continuous interactions among the three super-components Tian, Di and Ren. Each super-component has its own computation cycle, which is basically the following: Starting from some problem set P0, the super-component first adapts to the input from the environment as well as from other peer super-components. It then tries to find related problem elements by enumeration. After those problem elements not in the solution set have been eliminated either using the elimination operator or using the concentration operator, the termination condition can be tested. The termination condition is expressed by [guard x, y] where Cycle x is the current cycle and Cycle y is the cycle to jump to. Whenever one super-component completes its computation cycle, if a solution is found the computation ends, otherwise the control is transferred to the next super-component. Since there are three super-components, we will have three computation cycles.

The Tian super-component has computation Cycle1:

Cycle1 [guard1,2]: $P^1 0$ ***+adap***$_{\boldsymbol{Aij}}$= $P^1 1$ ***-enum***< $P^1 2$ ***>elim-*** $P^1 3$ ***=prop***$_{\boldsymbol{Aij}}$+ $P^1 4$

Likewise, the Di super-component has computation Cycle2:

Cycle2 [guard2,3]: $P^2 0$ ***+adap***$_{\boldsymbol{Aij}}$= $P^2 1$ ***-enum***< $P^2 2$ ***>elim-*** $P^2 3$ ***=prop***$_{\boldsymbol{Aij}}$+ $P^2 4$

Finally, the Ren super-component has computation Cycle3:

Cycle3 [guard3,1]: $P^3 0$ ***+adap***$_{\boldsymbol{Aij}}$= $P^3 1$ ***-enum***< $P^3 2$ ***>elim-*** $P^3 3$ ***=prop***$_{\boldsymbol{Aij}}$+ $P^3 4$

Notice the three computation cycles together form a higher-level computation cycle. High-level computation cycles are essential for a complex human-centric psycho-physical system such as the TDR system. In Section 6 we will discuss applications to personal health care.

2.3. A Compiler for the Abstract Machine Model

The Abstract Machine Model is a formal specification of the computation cycles of a slow intelligence system. Once the abstract machine model is provided, a compiler can be constructed to generate the components. In what follows we describe the major steps of the generic Abstract Machine Compiler (AMC) and the components it generated in pseudo codes.

Step 1: Adapt input from the environment

The AMC will first generate the basic components to gather input from the environment (see box below).

```
Basic Component:
        //initialize
        threshold = user input();
        while (true) {
                //adapt input from environment
                currentData = collectEnvironmentData();
                if (currentData exceed threshold) {
                   send alert message to Controller and/or Advertiser;
                } else {
                   send normal message to Controller on demand
            }
}
```

AMC Controller maintains the state and makes decisions based upon different states (see box below).

```
Controller:
//maintain the state within controller. Make decision based on different state.
create and run stateMachine;
while (true) {
        msg = getMsgFromSocket();
      do something that is not related to state machine
      stateMachine.perform(msg); //based on different states, perform
   differently when given input
}
```

For each Controller, when given some input, the State Machine will determine the action and the output. It may give several tries. For example, two solutions can be applied to one certain state when given certain input (see box below).

```
State Machine:
//define the states
enum Status {
     State0,
     State1,
     …;}
Status currentStatus = State0;//initial state
void perform(Message msg) {
        //based on different states, perform differently when given input
        switch (currentStatus) {
                case 'State0':
                based on message type and purpose, perform action or
                change state
                        break;
                case 'State1':
                based on message type and purpose, perform action or
                change state
                        break;
                ….
        }
}
```

Step 2: Enumerate and find related problem elements

For Step 2 and Step 3 the AMC is custom designed to handle different patterns from a pattern knowledge-base. For example, if the pattern is "picnic" the initial problem set may be as follows: P0={([0,10], [0,10], [10,20], [60,120], [60,80], [50,80])} where pj = (flower1j, flower2j, tempj, bpj, spoj, ekgj).

To answer the question "Is today a good day for picnic?" the temperature sensor is first used to measure temp. Depending on the results of the measurement, either enumeration operator or elimination operator can be applied.

Suppose the temp is 25. Since the temp is normal it cannot be used to eliminate other problem elements and therefore after enumeration P1 = {([0,10], [0,10], 25, [60,120], [60,80], [50,80])}. More computation is needed.

Step 3: Eliminate non-solution elements

Suppose the temp is 40. Since the temp is too hot, other problem elements are eliminated and therefore after elimination P1 is empty. Either the conclusion is “today is not a good day for picnic” or another computation cycle may be entered (to find an indoor location for picnic, for example).

Step 4: Propagate solution elements to peers

Once a solution is obtained, the abstract machine will propagate the solution to its peers. For example, several super components may do the work at the same time. Once one super component gets the solution, the rest of them can stop work. An Advertiser will then inform the other super components (see box below).

```
Advertiser pseudo code:
  while (true) {
        msg = getMsgFromSocket();
        switch (msg.type) {
                case 'Alert':{
              uploadAlert(); //upload alert message to database
              propagateAlert(); //propagate alert message to its peers }
              …
              };
          if( solution_set != null )
            propagateSolution(); //propagate solution to its peers
            terminateCycle(); //terminate computation cycle}
     else  switchCycle(); //switch to a new computation cycle }
}
```

3. The TDR System Architecture

As mentioned in Section 1, the TDR system is a multi-level slow intelligence system consisting mainly of three super-components: the Tian super-component, the Di super-component and the Ren super-component. The TDR System architecture is illustrated by Figure 3.1.

The TDR system has a common SIS server to support multi-level messaging. There is an integrated database to store TDR records, and a web GUI that supports the reception and sending of messages. Each super-component has its own sensor(s) to collect information from the environment. For example as shown in Figure 3.1 the Tian super-component has two plant sensors: Parrot Flower 1 and Parrot 2, the Di super-component has an ambient temperature sensor, and the Ren component has a blood pressure sensor. Each super-component furthermore consists of the following components: a monitor component to make sure the information collected by the sensor(s) is within certain acceptable range, a GUI component to interact with the user, an Uploader component to upload the collected information to the next higher level and last but not least a controller component to control the activities of the various components to realize the computation cycles described in the previous section.

When there are multiple controllers in a super-component such as the Tian super-component, a coordinator component can be introduced to coordinate the activities of the controllers and collect the information provided by the controllers. Generally speaking both the controller component and the coordinator component are essentially controllers, which should possess both the abilities to coordinate and to control the sub-components.

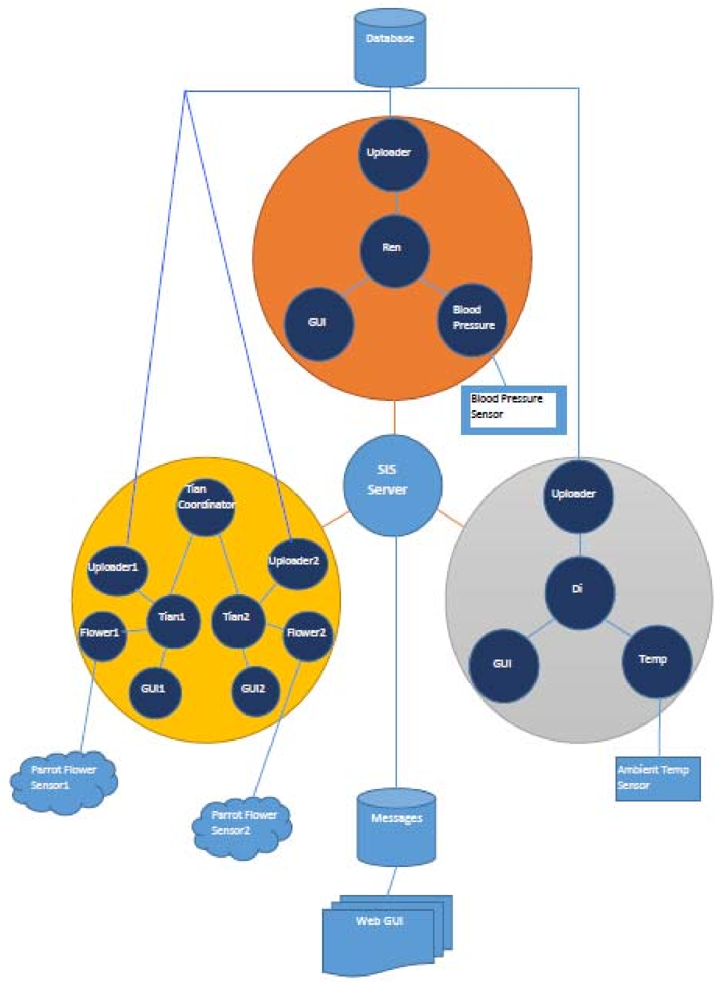

Figure 3.1. The TDR system architecture.

In the experimental test bed, the following functionalities are provided:

3.1 Define a Component

A component should have predefined scope, predefined role and unique name within its predefined scope, these can be described in the component’s Register message, which is stored as an XML document under xml/InitXML. All parameters are defined as Key-Value pairs. The scope defines where outgoing messages from this component can go and the scope of incoming messages this component can receive. Role defines the type of component that can only handle certain types of incoming/outgoing messages. Among all components with the same name within a certain scope, only one of them can be active, i.e., the component name must be unique. There are currently six predefined roles: Basic, Monitor, Advertiser, Controller, Coordinator and Debugger.

3.2 Create a Component

Once the designer knows how a component should behave, he can start to implement it under the Components folder. Components with all kinds of roles have similar templates for implementation. There are two places containing information that the designer should pay attention to: (i) xml/initXML where all predefined Register messages for all available components are stored. (ii) For new components the designer should use the same scope and name in both XML definition and for constants in codes under Components / NEW_COMPONENT_NAME_HERE folder (SCOPE, NAME). The implementation of each role is far from being different from each other. As long as the designer doesn't add extra message types to the collection of acceptable incoming messages, he can simply replace all scopes and names (folder name, SCOPE, NAME, class name, java source code CreateXXX, XML under initXML folder) and create a new working component almost immediately. The six different roles of components are as follows:

3.2.1 Basic Component

For a Basic component such as Blood Pressure in Figure 3.1, no changes are necessary for main method. Method "initRecord" is provided as a place for putting initialization code. Method "componentTask" is provided as a place for putting periodically executed code, such as collecting data. Method "ProcessMsg" is provided as a place for handling different types of messages. Other variables can be added if needed, but the framework should suit the general purpose for implementing a Basic component that sends out Readings which are collected from a data source.

3.2.2 Monitor Component

For a Monitor component such as any GUI in Figure 3.1, it can be designed as a general monitor or a visual console to display data. Other variables can be added if needed.

3.2.3 Advertiser Component

For an Advertiser component such as any Uploader in Figure 3.1, it can be designed as a tool to process the Readings and send anything outside the system via emails, sockets, etc.

3.2.4 Controller Component

For a Controller component such as Ren in Figure 3.1 to process combination of TempBloodPressure measurements, it can be broken down into code segments similar to the TempBloodPressure segment. Five types of code segments are under the ControllerComponents/TempBloodPressure folder: "initial.java" contains all initialization code of extra variables , "helper.java" contains all helper methods used and "helperClass.java" contains all user defined classes. By default Controller components only process Alert messages from Basic components. Alert messages must have unique names. The same names are used to create code snippets under the TempBloodPressure folder.

3.2.5 Coordinator Component

The Coordinator component such as Tian Coordinator in Figure 3.1 processes the messages from controller components and other components and coordinates the activities of controller components and other components.

3.2.6 Debugger Component

The default Debugger is the PrjRemote.exe tool. It can be replaced by a customized Debugger. However, when a component is assigned the Debugger role, it will get a copy of all messages within the scope that it is in.

3.3 How to Run a Component

Scripts for the Controller component will be automatically generated. For all other roles customized scripts must be provided under the Scripts folder. For Basic or Monitor or Advertiser component, one can simply copy the BloodPressue or the GUI or the Uploader component, respectively, and do some name replacement.

3.4 Scoping

There can be multilevel scopes, each of which contains components that collaborate with each other or are related to each other. Scoping provides a way to further divide the components. By default messages will only be sent within current scope, but one can add ("Broadcast", "True") and ("Direction", "[Up/Down]") to enable broadcast of messages.

3.5 Trouble-Shooting

If a component cannot be connected to the SISServer, one should check SCOPE and NAME in both code and xml definition. If a message is not delivered, check if the message is sent to a target that does not process this type of message. It is also possible that one forgets to add certain parameters to the message such as valid Scope, Sender, Purpose, etc.

4. The Tian Super-component

4.1 System Structure

A plant is heavily dependent on the environment. According to Chinese philosophy, we may consider the plants' status as Tian (heaven), which will indicate environmental status to some degree. The plant sensors made by Parrot are used in our experiment, which can gather such data as amount of sunshine, moisture, temperature and amount of fertilizer in a plant's environment (see Figure 4.1).

In Tian super-component, we include two different parrot-flower-sensors for plants in different locations. Since they are located in different places, they can gather data from two different environments. Figure 4.2 illustrates the interactive Tian super-component. Notice there are two Tian controllers coordinated by the Tian coordinator.

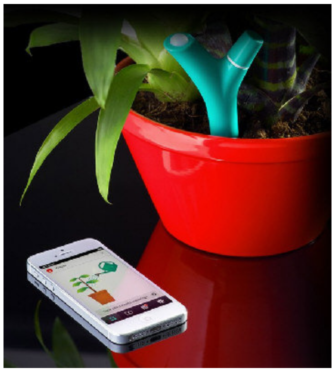

Fig 4.1. The plant sensor Parrot Flower Power.

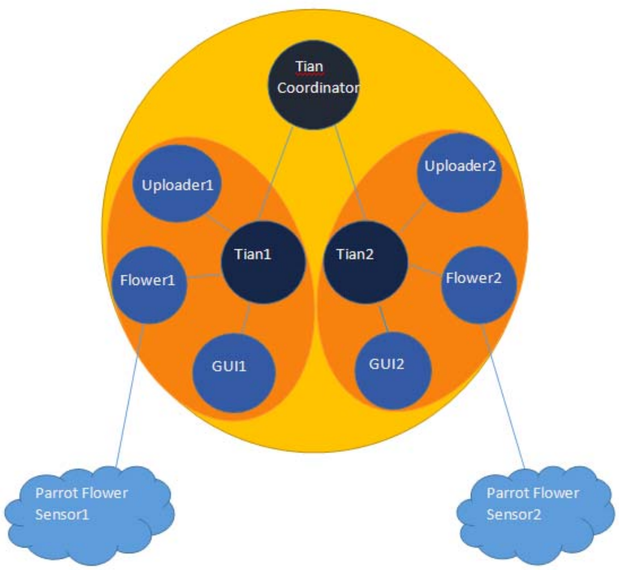

Figure 4.2. The Tian Super-component.

Thus the Tian Super-component has three layers:

- **Top Layer**: Tian coordinator
- **Middle Layer**: Tian1/Tian2 controllers
- **Bottom Layer**: Flower1/Flower2, GUI1/GUI2, Uploader1/Uploader2

Bottom layer is in charge of getting data from the sensor, displaying data to user, and uploading the data to database. Middle layer is in charge of logically activating/deactivating bottom layer components based on instructions from higher level. Top layer is in charge of aggregating data from lower layers. Top layer coordinator will also communicate with other super-components' top-layer.

4.2 Data Path

There are two paths for the data. One is for data going upwards, which is done through Uploader1 and Uploader2. The other is for data going downwards, which is handled by Tian1 and Tian2. Tian1 and Tian2 are both controllers and can make their own decisions such as activate or deactivate the corresponding Flower1 and Flower2 components.

4.3 Control Message Definition

1. Activate all components
   Sender: Web GUI
   Receiver: Tian1/Tian2
   Purpose: activate all components under Tian1/Tian2 sub-system

2. Deactivate all components
   Sender: Web GUI
   Receiver: Tian1/Tian2
   Purpose: deactivate all components under Tian1/Tian2 sub-system

3. Active Flower1/Flower2 component
   Sender: Tian1/Tian2
   Receiver: Flower1/Flower2 component
   Purpose: activate Flower1/Flower2 component

4. Deactivate Flower1/Flower2 component
   Sender: Tian1/Tian2
   Receiver: Flower1/Flower2 component
   Purpose: deactivate Flower1/Flower2 component

4.4 AMC Compiler Steps for Tian

Compared to the generic Abstract Machine Compiler AMC described into Section 2.3, the Tian Compiler has these following steps: Step 1 and Step 4. The other two steps do not exist. In what follows we describe how the Compiler generates the various components in pseudo codes.

In Tian compiler, there are two alert states and four inputs: Alert Flower1, Alert Flower2, Activate all components and Deactivate all components. Since the problem vector will have one and only one state element to be 1 and one and only one input element to be 1, so there are a total of C(1, 2)*C(1, 4) = 8 different problem vectors.

For example,

$p_0 = (1, 0, 0, 1, 0, 0)$.

This specifies when in normal state, given input Alert Flower1, how the abstract machine should perform.

Step 1: Create flower component to adapt input from the environment

The Flower1 and Flower2 monitors will gather environmental data (Sunlight, Moisturizer, Temperature, and Fertilizer) stored in parrot cloud which is updated by parrot sensors. The Flower1 and Flower2 monitors will generate alert message when new environmental data come in. Only Flower1 component and Tian1 component will be described below.

```
Flower1 Component:
        //initialize
while (true) {
        //adapt input from environment
        currentData1 = collectDataFromSensor1();

        //send alert message if exceed threshold
        if (currentData1!=nulls) {
                sendAlertMsgTo(Controller);//inform
                Controller that new data comes in
        sendAlertMsgTo(Advertiser);//inform
        Advertisser to propagate new data
        } else {
                //no new data comes in }
}
```

```
Tian1:
//maintain the status within controller. Make decision based on different status.
    //create and run stateMachine;
    msg = getMsgFromSocket();
  switch (msg.type) {
        case 'Alert:
          if(msg.sender == "Flower1"){
                Tian1DataArray.add(msg);
                 sendEmergencyMessageTo("Tian Coordinator");
          }
          break;
        case 'Setting':
          stateMachine.perform(msg);
          break;
    }
```

```
StateMachine:
Status currentState = ALERT;//initial state
void perform(Message msg) { //There are two kinds of messages:
//a). environment data message from Tian Components, including alert flower
//b). control message from WebGUI, including activate and deactivate all
      components
        switch (currentState) {
                case ALERT:
switch (msg.type) {
                case 'Setting'://control message from webGUI
switch (purpose) {
case 'Activate':
        activate Tian components.s
case 'Deactivate':
        deactivate Tian components  } break;  ] break;
    }  }
```

```
Coordinator:
    switch (msg.type) {
        case 'Emergency:
          if(msg.sender == "Tian1"){Tian1Array.add(msg);
          }
          if(msg.sender == "Tian2"){Tian2Array.add(msg);
          }
          break;
}
```

Step 2: (does not exist)
Step 3: (does not exist)
Step 4: Create upload component to propagate solution elements to peers.

The Advertiser component can upload necessary messages to the database and propagate to its peers. If a solution is found an Advertiser will color-code its banner in a tranquil color such as "blue" and inform the other components. If no solution is found, its banner is color-coded "red" and control is switched to a new computation cycle (see box below).

```
Advertiser pseudo code:
  while (true) {
          msg = getMsgFromSocket();
          switch (msg.type) {
                  case 'NewFlower':{
                uploadAlert(); //upload alert message to database
                propagateAlert(); //propagate alert message to its peers }
                ...
                };
      if( solution_set != null ) {color-code("blue");
       //this component  is color-coded "blue"
              propagateSolution(); //propagate solution to its peers
              terminateCycle(); //terminate computation cycle}
      else  {color-code("red"); //this component is color-coded "red"
            switchCycle(); //switch to a new computation cycle }
  }
```

5. The Web GUI

The dashboard is the main GUI interface of the TDR system. As illustrated by Figure 5.1 it provides a high-level overview of the data in the system.

On the left side, it has a menu panel that contains all the actions the user can perform, including activating and deactivating components. For the super user, this menu will also include addition, deletion and modification of regular users. The activation and deactivation messages are sent utilizing a message database (MDB) and the TDR components will actively fetch the incoming messages from the MDB (see Figure 3.1).

Figure 5.1. The dashboard for super user.

There is a carousel that displays all components in rotation, four at a time for the PC screen and only one for the smart phone screen. This vividly demonstrates the idea of computation cycles in the TDR system. A component's banner is in *tranquil* state (blue color) until an alert is received and then it changes to red color. Below the carousel panel, a table will be displaying all records that belong to the current user. For each entry, it contains the date and time of a record, the sensor type, the data type, the actual reading of the data, and the originator. This scheme allows flexibility and scalability, as in the future there might be more and more sensors added to the TDR system.

By clicking the “find similar” button at the lower right part of the dashboard, messages such as M3 will be sent to the MDB to be fetched by the similarity retrieval component to find other user’s records similar to the current user’s record. The messages sent are exactly the same as the ones used in the TDR system, hence the web interface can be viewed as one remote component of the TDR system.

After clicking one specific component on the carousel panel, it will show a detailed list of records that are from the component. If the user is communicating remotely with his/her doctor, a user might want to specify the record ID so that the doctor knows exactly what entry he/she is referring to. A graph showing the data-to-day changes of a selected data iitem can also be displayed by the GUI for visualization purpose.

6. Discussion

In the formulation of the computation cycles for the TDR system, one can start with the computation cycle of any one of the three super-components. For example the TDR system may start with Ren, i.e., the human conditions are first taken into consideration. Then the atmospheric environmental attributes from Tian and surrounding residential attributes from Di are considered so that an overall solution can be found to enhance the chances of survival of the human being. The TDR system then tries to find appropriate Tian attributes tij representing atmospheric environmental variables such as sunlight and water level, etc. and Di attributes dij representing surrounding residential variables such as ambient temperature and humidity, etc.

Alternately the TDR system may also start with the Tian or the Di computation cycles. Different constrained optimization algorithms can be formulated depending on the structure of multi-level computation cycles for Tian, Di and Ren to obtain the “best” solution, i.e., the solution that increase the probability of human survival the most. Finally instead of (or in addition to) constrained optimization algorithms we can also manually set certain variables by human-centric interactions or through social interactions.

One of our main goals is to expand the TDR system for the computation of Chi (also spelled as Qi in Chinese transliteration system HanYu PinYin). The Chi super-component is regarded as at a higher level. It has attributes including both objective measurements and subjective evaluations. Some researchers propose to employ electrical measurements to estimate Chi [5]. Other researchers propose to combine objective measurements with subjective evaluation into an evaluation matrix to estimate Chi [6]. This makes the Chi super-component both pro-active and adaptive at multiple levels.

The dashboard for TDR system can be further refined. When one clicks on “view details” for the Chi super-component, a list of attributes for Chi is shown. The objective measurements in this list is filled by the multi-level computation cycles based upon actual measurements. The subjective evaluations are entered by the principal user himself/herself based upon his/her subjective feelings.

The experimental TDR system provides a versatile platform for exploring and integrating applications such as personal health care, emergency management and social networking, etc. These applications are currently being investigated at our research laboratory, with major emphasis on an experimental TDR system to estimate Chi. We will also further investigate the theoretical issue to define and characterize the *resonance state* of a system with multiple, multi-level computation cycles.

References

[1] Shi-Kuo Chang, "A General Framework for Slow Intelligence Systems", *International Journal of Software Engineering and Knowledge Engineering,* Volume 20, Number 1, February 2010, 1-16.

[2] Shi-Kuo Chang, Yingze Wang and Yao Sun, "Visual Specification of Component-based Slow Intelligence Systems", Proceedings of 2011 International Conference on Software Engineering and Knowledge Engineering, Miami, USA, July 7-9, 2011, 1-8.

[3] Shi-Kuo Chang, Yao Sun, Yingze Wang, Chia-Chun Shih and Ting-Chun Peng, "Design of Component-based Slow Intelligence Systems and Application to Social Influence Analysis", Proceedings of 2011 International Conference on Software Engineering and Knowledge Engineering, Miami, USA, July 7-9, 2011, 9-16.

[4] Shi-Kuo Chang, Sen-Hua Chang, Jun-Hui Chen, Xiao-Yu Ge, Nikolaos Katsipoulakis, Daniel Petrov and Anatoli Shein, "A Slow Intelligence System Test Bed Enhanced with Super-Components", Proceedings of 2015 International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, USA, July 6-8, 2015, 51-63.

[5] Ming-Feng Chen, Hsi-Ming Yu, Shu-Fang Li and Ta-Jung You, “A Complementary Method for Detecting Qi Vacuity”, *BMC complementary and alternative medicine*, Vol. 9, No. 12, 2009.

[6] Ke-Feng Huang, *Effects of Energy Absorption on Meridian System* (能量攝取對經絡系統影響之效應), Doctoral Dissertation (in Chinese), Institute of Biomedical Engineering, National Yang-Ming University, Taiwan, June 2011.