# A Software Development Framework for Various Anticipatory Reasoning-Reacting Systems

Kai Shi, Yuichi Goto, and Jingde Cheng
Department of Information and Computer Sciences, Saitama University
Saitama, 338-8570, Japan
+81-48-858-3785 - {shikai, gotoh, cheng}@aise.ics.saitama-u.ac.jp
- http://www.aise.ics.saitama-u.ac.jp

**Abstract**
Building anticipatory reasoning-reacting systems (ARRS) is difficult. An ARRS consists of many components and works based on some fragments of logic systems and formal theories. Besides, there is no general development process to build a practical ARRS. On the other hand, different ARRSs have common functions and similar working process. The development of these common parts should not be repeated when we build different ARRSs. To this end, this paper proposes the first development framework for ARRSs, which is a reusable, semi-finished ARRS that can be customized by developers to produce custom ARRSs for various target domain. By using the framework, we can use a general development process to build a particular ARRS.
**Keywords** : anticipatory reasoning-reacting system, software development, framework, multi-agent system, soft system bus.

## 1 Introduction

The traditional reactive systems are passive, i.e., the systems only can perform those operations in response to instructions explicitly issued by users or application programs, but have no ability to do something actively and anticipatorily by themselves [4]. In order to make reactive systems anticipate, Cheng proposed a new kind of highly reliable and secure computing systems called *anticipatory reasoning-reacting systems* (ARRS) [4], which have anticipatory ability by making qualitative predictions of future and taking proper actions, therefore it can forestall attacks and disasters.

Building an practical ARRS is not an easy task. An ARRS consists of many components and it must work based on some fragments of logic systems and formal theories. Besides, there is no general development process to build a practical ARRS by now. On the other hand, various ARRSs have many common functions and similar working process. The development of these common parts should not be repeated when we build different ARRSs.

In order to facilitate the development and maintenance of ARRSs, Goto et al. [12] proposed a development and maintenance environment for ARRSs, identified and classified common components that an ARRS should have, and drew up the basic requirements of the development and maintenance environment. However, they did not discuss how to develop a practical ARRS for a target domain. Is there a unified development process/method which can build various practical ARRSs for any target domain?

To this end, this paper proposes a development framework [11] which is used to build ARRSs for various target domains. The framework is a reusable, semi-finished ARRS that can be customized by developers to produce custom ARRS for a specific target domain. By using the framework, we can use a general development process to build a particular ARRS. Besides, because the previous ARRS are not suit to construct such a framework, this paper also proposes a new ARRS which can manage the differences of different target domains.

The rest of the paper is organized as follows. Section 2 reviews current ARRSs, and discusses the limitation of the ARRSs. Section 3 shows what the new ARRS is, and its necessity. Section 4 presents the software development framework for ARRSs, the process to build a practical ARRS based on the framework, and implementation issues of the framework. Finally, section 5 gives some concluding remarks.

## 2 Anticipatory Reasoning-Reacting Systems

### 2.1 Overview of Anticipatory Reasoning-Reacting Systems

An *anticipatory reasoning-reacting system (ARRS)* is a computing system containing a controller C with capabilities to measure and monitor the behavior of the whole system, a traditional reactive system RS, a predictive model PM of RS and its external computing environment, and an anticipatory reasoning engine ARE such that according to predictions by ARE based on PM, C can order and control RS to carry out some operations with a high priority [4].

As a certain kind of anticipatory systems, to be anticipatory, ARRSs should behave continuously and persistently without stopping its running [10]. Thus, persistent computing systems should be as an infrastructure of computing anticipatory systems [10]. Cheng proposed *Persistent computing* as a new methodology, aiming to develop continuously dependable and dynamically adaptive reactive-systems, called “persistent computing systems”, which have two key characteristics/fundamental features: (1) persistently continuous functioning, i.e., the systems can function continuously and persistently without stopping its reactions, and (2) dynamically adaptive functioning, i.e., the systems can be dynamically maintained, upgraded, or reconfigured during its continuous functioning [5, 6, 7].

As a computing anticipatory system with the ability to predict and take next actions, an ARRS is complex. The working process of ARRSs mainly includes two phases. The first phase is *prediction*, to make some future events known in advance, especially on the basis of special knowledge, or statements about the future events. And the second phase is choosing anticipatory actions according to the predictions, and taking these actions.

Goto et al. [12] re-defined the architecture of the ARRS shown in figure 1, as well as

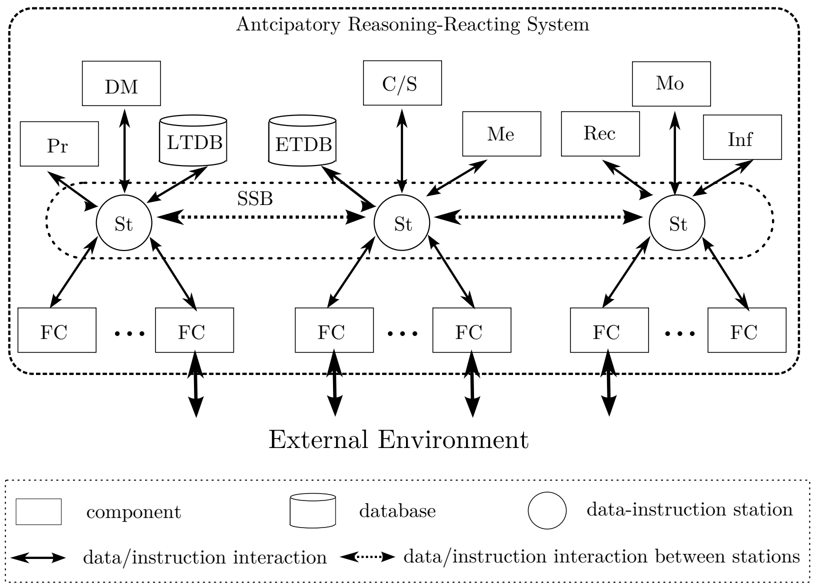
Antcipatory Reasoning-Reacting System
DM
C/S
Mo
Pr
LTDB
ETDB
Me
Rec
Inf
SSB
St
St
St
FC
FC
FC
FC
FC
FC
External Environment
component
database
data-instruction station
data/instruction interaction
data/instruction interaction between stations

**Fig. 1**: The Architecture of an Anticipatory Reasoning-Reacting System

identified and classified common functional/non-functional components and application-independent data in all ARRSs. They also explained how these components work and cooperate. Common components in all ARRSs are classified into two kinds of components: *PCS-core components* and *ARRS-core components*. *PCS-core components* are common components in all persistent computing systems. PCS-core components include central control components, such as a central controller/scheduler (C/S), a central measurer (Me), a central recorder (Rec), a central monitor (Mo), and an central informant (Inf), as well as the soft system buses [3] which provides a communication channel with the facilities of data/instruction transmission and preservation to connect components. *ARRS-core components* are common components in all ARRSs, but not in all persistent computing systems. ARRS-core components include a predictor (Pr), a decision maker (DM), a logical theorem database (LTDB), and an empirical theory database (ETDB). A *predictor* receives several kinds of data and outputs predictions. A *decision-maker* receives predictions from a predictor, and outputs instructions to an ARRS. The logical theorem database stores fragments of logic systems underlying anticipatory reasoning or reasoning about actions. The empirical theory database stores empirical theories of a target domain as predictive models, behavioral models, or world models. The data involving in an ARRS are raw input data, fragments of logic systems, a world model, a predictive model, a behavior model, translation rules, calculation rules, interests, and priority. Only fragments of logic systems are application-independent data.

## 2.2 Weakness of Original Anticipatory Reasoning-Reacting Systems

The motivation of original ARRS is to make the ARRS itself have the ability to anticipate, but the original ARRS are not suitable for the new target domains of ARRSs. In an original ARRS, the anticipatory entities are the system itself, i.e., the ARRS can predict potential attacks or failure, and take anticipatory action to deal with such problems. However, some new target domains of ARRSs were proposed, such as the ARRS for air traffic control [8, 14]. In such target domains, the anticipatory entities are not the ARRS itself but the entities outside of the ARRS, e.g., the aircrafts in air traffic control. Although the original ARRSs are suitable for the early target domains, it can not tackle the new target domains in the same way.

An original ARRS needs to re-build all current status as logical formulas over time. In an original ARRS, for each prediction, we need gather all current status of the target problem, and translate all status to logical formulas, then use these formulas to predict. Similar process is taken when to choose actions. Because an original ARRS handles all current status together, when the current status changes over time, we have to re-build all current status as logical formulas, re-predict by using these logical formulas, and re-choose actions by using these logical formulas. However, in the real world, only part of status changes over time, i.e., most of entities stay unchanged. Thus such re-building for all status is unnecessary.

The amount of anticipatory entities affects the efficiency of an original ARRS. In an original ARRS, there is only one predictor and one decision-maker, and the ARRS uses all current status to predict and choose action. Thus, when the amount of anticipatory entities increases, the efficiency of the ARRS reduces. Such effect may be tolerable in the early target domains of ARRSs, because there are few anticipatory entities such as components and services. However for the new target domains, the number of anticipatory entities increases a lot. As a highly reliable computing system, the ARRS should not be affected by the number of anticipatory entities.

The computing of prediction and choosing actions is centralized in original ARRSs. Because there is only one predictor and one decision-maker in an ARRS, we have the following problems. First, if the only predictor breaks down, how does the ARRS behave continuously and persistently? Second, the current architecture of ARRSs can not support parallel computing/distributed computing, because of the only one predictor and decision-maker, which spends most of computing time, i.e. the predictor and decision-maker are the bottleneck of the ARRS.

The original ARRSs do not take into account system flexibility. As software, an ARRS can not avoid business changes. Besides, as a persistent computing system, an ARRS should not stop when it changes its business. By now, it is unsolved how to change an ARRS's business without stopping it.

The original ARRSs do not take into account to support for different systems/devices outside the ARRSs. An ARRS especially for new target domains often deals with a lot of devices outside the ARRS, such as aircrafts, vehicles, and radars. However, it is unsolved how to handle these different devices in a unified way in ARRSs. Moreover, the ARRS should not stop when we add/remove/modify such a system/device.

## 3 New Anticipatory Reasoning-Reacting Systems

### 3.1 The Conceptual Model of New Anticipatory Reasoning-Reacting Systems

Due to the weakness of the original ARRS, it is not suitable to construct a development framework based on the original ARRS. Therefore this paper proposes a new kind of ARRSs to underline the development framework for various ARRSs. The conceptual model of the new ARRS is an agent-based system. In the middel of figure 2, it is the conceptual model of an ARRS, or called the virtual world on an ARRS. And on the top of figure 2, it is the real world of target domains for the ARRS. In the real world, there are different kinds of *active entities*, which have ability to act, i.e., have ability to change the status of the real world, e.g., computing systems, software components, computing services, airplanes, vehicles, while there are also some *passive entities* which do not have ability to act, e.g., runways, taxiways. So we propose a virtual world on ARRSs to handle the differences of the entities, by mapping each entity in the real world of target domain to an *agent* in the virtual world. There are two kinds of agents in ARRS: *active agents* and *passive agents*. In figure 2, in the virtual world on an ARRS, a circle stands for an active agent, and a square stands for a passive agent. For any active entity in the real world, we map it to an active agent in the virtual world. For any passive entity, we map it to a passive agent. In the virtual world, each agent collects its related current status by itself. Such related current status includes the information about itself, and information about its related agents, i.e., the relationships between itself and other related agents (In figure 2 a two-way arrow stands for the relationship). And each agent can find out which other agents could affect it, and also knows whether there are some unrelated agents become related with it when time passes by, vice versa. Besides, each agent can automatically update all above information. For each active agent, besides the above ability, it has the ability to anticipate. It means each active agent knows how to predict, how to choose actions, and how to take the actions. For the passive agents, because they have not ability to act, they can not anticipate. Thus the passive agents do not need the ability to predict and to choose actions.

If an ARRS can work like the virtual world that has been stated above, we could make up for original ARRSs' weakness. First, because we map different type of anticipatory entities to similar agents, the new ARRS can handle all of the anticipatory entities in the same way, i.e., the new ARRS can handle both the original target domain and new target domain of ARRSs. Second, because each active agent only manage its related current status, if an agent's status changes over time, only this agent and its related agents need to re-build the current status, while other unrelated agents do not need to to re-build the current status. Third, because each active agent has ability to predict and to choose actions by itself, the amount of anticipatory entities can not affect the efficiency of the ARRS. Fourth, because each active agent has computing resource to predict and choosing actions by itself, the new ARRS supports parallel computing inherently, and it is easy to deploy an ARRS in a distributed environment. Besides, the failure of any agent could not affect other agents. Fifth, agent-based systems show flexibility to face the changing business

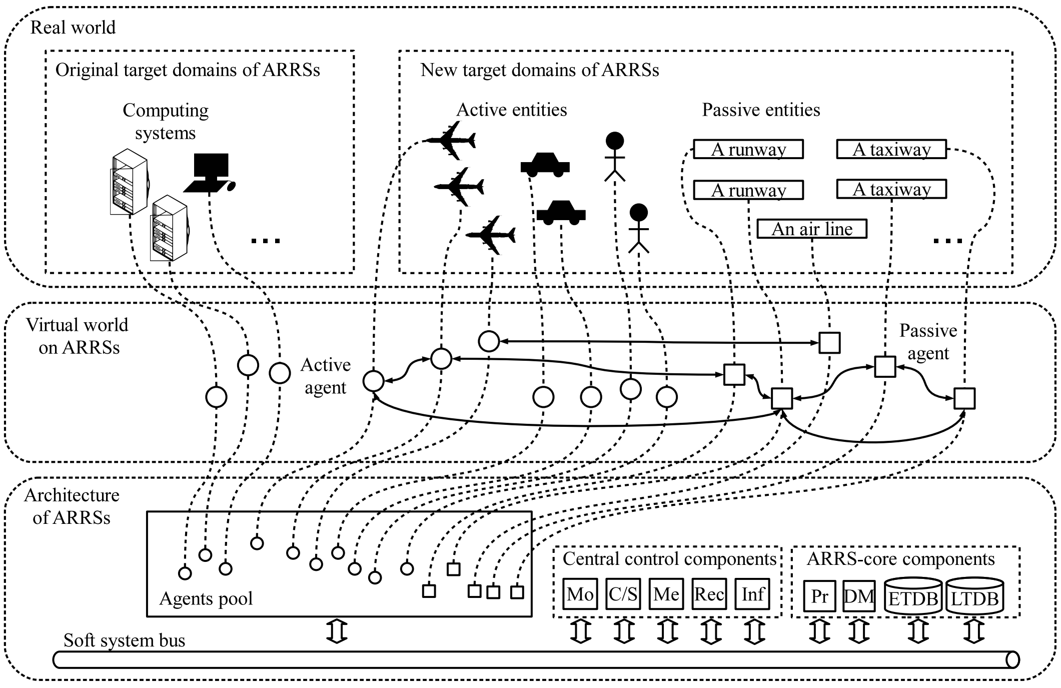

**Fig. 2**: View of the new anticipatory reasoning-reacting systems

[1, 13]. Because we encapsulate the business into agents, we could add/remove/modify agents dynamically to meet the changing business.

## 3.2 Architecture of New Anticipatory Reasoning-Reacting Systems

Figure 3 shows the architecture of the new ARRS. This architecture satisfies: (1) the architecture is a practical implementation of the conceptual model of a new ARRS, (2) the architecture can be used directly to build a software development framework for ARRSs. A new ARRS consists of PCS-core components, ARRS-core components, agents pools, and I/O adapters. In a new ARRS, PCS-core components are as same as the current ARRS presented in section 2.1. ARRS-core components are also as same as the current ARRS, however, there is no longer only one predictor and one decision maker. A new ARRS provides redundant predictors and decision makers. Each predictor or decision maker actually provides a kind of computing service. When other components/objects in the ARRS request such a service, a unoccupied predictor or decision maker will be called. The new ARRS provides redundant predictors and decision makers, because: (1) when one predictor/decision maker fails, we can use other predictors/decisions, (2) when one predictor/decision maker is occupied, other predictors/decision makers could be used, which improves the concurrency of the ARRS.

In a new ARRS, an *agent* is an object which encapsulates data and behavior, and running in an agents pool. Each agent has data structure *facts container* and *observers' list*. A *facts container* stores *facts* which are logical formulas to represent the status of the

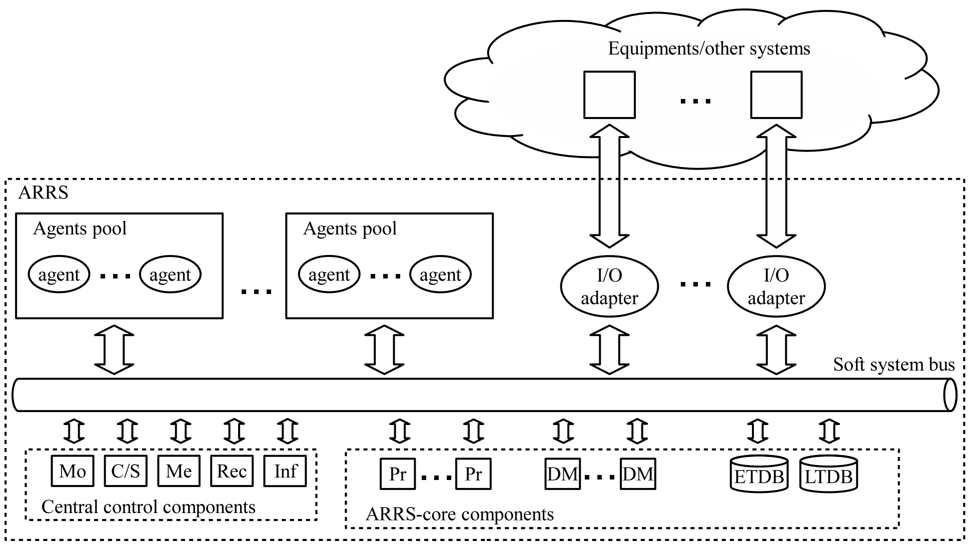

**Fig. 3**: Architecture of a new ARRS

agent itself or relationship between the agent and other agent. An *observers' list* records all of other objects who want to observe the change of this agent. Each agent can find out which other agents are related with itself, accumulates the current status about the agent itself and relationship between itself and other agents, and notifies other agents who are on its observers' list once the agent's properties changes. Agents are divided into two categories: *active agents* and *passive agents*. For an active agent, it has additional data structures *predictions container* and *actions container*. A *predictions container* stores intermediate candidates of prediction. An *actions container* stores intermediate candidates of next actions. An active agent also has some additional functions: to predict by calling a predictor, and to choose actions by calling a decision maker. Besides, for each active agent, we can implement its every possible (anticipatory) action as a behavior (function). Each behavior includes the detail instructions about how to take an action. In the conceptual model of the new ARRS, each active agent has ability to predict and choose action, while in the actual architecture, an agent does not have computing resource to predict and choose action inside itself, but only calls a predictor and a decision-maker. We remove the computing of prediction and choosing action form active agents, and make the prediction and choosing action as computing services which is used for agents to call, because: (1) active agents may be different, however the functions of prediction and choosing action are the same, so we should distinguish commonality and individuality, (2) in many cases, active agents does not change its status most of time, then the computing resource is not necessary.

An *agents pool* is a component which is a running environment of agents. An agents pool provides memory, CPU time, and other computing resource to agents running inside it, manges the life cycle of agents, and interacts with the agents. Thus one agent can not run outside the agents pools. We propose agents pools for the following reasons: (1) because an agent needs several resource, the agents pool can control the allocation and deallocation of these resource. (2) we can deploy several agents pools on different computer servers, while an agent can move among these different agents pools, when one agents pools fail, the agents in this container can move to other agents pools, (3) Load balancing can be achieved by the same way.

An *I/O adapter* is a component which maps to a functional component, a device/equipment, or an reactive system. An I/O adapter can inputs from or outputs to a functional component, a device/equipment, or an reactive system. In the conceptual model of the new ARRS, each functional component, device/equipment, or reactive system is mapped to an agent directly, while in the actual architecture, each functional component, device/equipment, or reactive system is mapped to an I/O adapter, then that I/O adapter is mapped to an agent. The reason why we need I/O adapters is as follows. Although we can map each entity in real world to an agent in ARRSs, we may not realize this map directly, because there are many kinds of entities in the real world, while one entity may involve several systems or equipments. For example, if we want to gather the information of an aircraft, we may deal with radars, GPS on the aircraft, geographic information databases, or other systems. Besides, when the ARRS predicts and chooses actions for the agent aircraft, the ARRS needs to give instructions to some equipment of the aircraft or display the warnings or actions suggestions on both the monitor of aircraft and the monitor of the control tower. Because there are kinds of input data and output instructions for ARRSs, we should decouple these input data and output data from the representation of data in ARRSs. We can shield these different and provide a unified interface inside the ARRS by encapsulating the differences of devices of real world in I/O adapters. In an ARRS, it is regular to add/remove/modify input data type, such as adding a new kind of sensor and retirement of outdated equipments. We handle these variation by add/remove/modify a corresponding I/O adapter.

Now we present the working process of the new ARRS. For each agent, it first gathers all related current status including both the information about itself and the relationships between itself and other related agents. In order to gather and update the relationship information in time, one agent needs to observe its related agents to find out whether their statuses change. If one agent wants to observe another related agent, it registers in that agent's observers' list. Once that agent's properties change, it notifies all of the agents in its observers' list to update. After one agent updates its current status, it sends the status to a predictor, gets the translated logical formulas, and stores them in its facts container. Then an active agent sends its current status to a predictor, gets the predictions from the predictor, and stores them into the predictions container. At this moment the active agent could give the ARRS users an alarm/a warning about the prediction. After that, active agent sends the predictions to a decision-maker, gets next actions from the decision-maker, and stores the candidates in its actions container. After the active agent gets next actions, it is time to "act". Because each active agent implements its every possible action as a behavior (function), the active agent can call the corresponding function directly. Each function includes the detail instructions about how to take the action. The detail instructions may calls the corresponding I/O adapters and functional components. Then the I/O adapters and functional components handle these instructions, and “act”.

## 4 Framework Based on New Anticipatory Reasoning-Reacting Systems

### 4.1 Framework Overview

As a reusable, semi-finished application that can be customized by developers to produce custom applications [11], a software development framework is an ideal solution to build complex software for different target domains, such as ARRSs. The developers should not build an ARRS for a target domain from scratch. Besides, the developers should not focus on the ARRS itself, such as how to predict, how to choose actions, how to ensure the security, and how to improve the efficiency, but to focus on the business of the target domain. In order to “stand on the shoulders of giants”, the developers need not only the common components of ARRSs, but also a skeleton which specifies how these components cooperate. Furthermore, the developers also need a mechanism to add their own code based on the specific target domain into these common parts of ARRSs. Thus, a software development framework with modularity, re-usability, and extensibility is suitable for this purpose.

The software development framework for ARRSs is a semi-finished ARRS that can be customized by developers to build a practical ARRS for a target domain. The framework consists of *frozen spots* and *hot spots*. Frozen spots remain unchanged in any instantiation of the application framework, such as the architecture of the ARRS, its basic components and the relationships between them [15, 16]. Hot spots represent those parts where the developers can add their own code of specific functionality to their own ARRS for the target domain [15, 16]. Because the new ARRS is proposed to construct a software framework for ARRSs, the architecture of the framework is as same as a new ARRS’s. Next we will distinguish between the frozen spots and hot spots of the framework.

### 4.2 Frozen Spots

Frozen spots are implemented parts of the semi-finished ARRS, which are the same in any ARRS. The Frozen spots of the framework for ARRSs include: (1) the architecture of new ARRSs, (2) the components, including PCS-components, ARRS-core components, agents pools, and I/O adapters, (3) the specification for components to handle errors, to exchange data, and to invoke operations on each other, (4) implemented parts of abstract class of active agents and passive agents, such as facts container, observers list, predictions container, actions container, method to formalize the current status, method to predict, and method to choose action of active agent, as well as facts container and observers list of passive agents.

## 4.3 Hot Spots

Hot spots characterize the incomplete parts of the semi-finished ARRS, which are different in different target domains. These hot spots can be adapted to meet specific target domain requirements. The differences between different ARRSs are: (1) world model, predictive model, and behavioral model for the target domain, refers to section 4.5, (2) active agents, (3) passive agents, and (4) I/O adapters. The models for the target domain are not belong to hot spots, because they are just logical formulas as data. Thus in the framework, the hot spots rely on active agents, passive agents, and I/O adapters. The framework defines the abstract class of active agents, passive agents, and I/O adapters. By implementing the abstract methods of these abstract classes, the developers of ARRSs could construct their own agents, passive agents, and I/O adapters for the target domain. The abstract methods include: (1) For the abstract class of active agent, its abstract methods include the method to gather current status about itself, including to determine which other agents have relationship with itself, and the method to handle the effective next actions. (2) For the abstract class of passive agent, its abstract method is to gather current status about itself, including to determine which other agents have relationship with itself. (3) For the abstract class of I/O adapter, its abstract method is to communicate (input and output methods) with other components in the ARRS.

## 4.4 Supporting Tools

In order to build a practical ARRS based on the framework, we also need some supporting tools. Theses supporting tools include database management, model constructing, system configuration, and system management.

## 4.5 Building a Practical ARRS Based on the Framework

In order to build an ARRS for the target domain, we must have the knowledge of the real world, predictive laws, and behavioral patterns, while all these information are represented as logical formulas of a specific logic (or logics). These models describe the specific theories or facts about the target domain. In ARRSs, the models are divided into three kinds called world model, predictive model, and behavioral model, which represent the information of the real world, predictive laws, and behavioral patterns correspondingly. However, this paper does not discuss how to construct these models.

Based on the models, we design and implement the agents. The implement of agents is similar with traditional object-oriented programming. Each agent has properties and methods. Each agent has capability to update their properties by communication with I/O adapters or functional components. Each agent must implement the abstract methods. Based on different input/output devices and cooperative systems involving in the problem, we design and implement the corresponding I/O adapter for each device/system.

Now we discuss briefly how to implementation of an ARRS for runway safety by using the framework. The ARRS for runway safety belongs to the new target domain of ARRSs shown in section 2.2. The ARRS for runway safety involves a lot of entities, equipments, and software systems outside the ARRS. The entities include aircrafts, vehicles, runways, and taxiways. The equipments include the equipments on aircrafts, equipments on vehicles, radars, and other equipments of the airports. The softare systems refer to the currently available software systems using by aircrafts and airports. To implement an ARRS for runway safety, the developers of the ARRS need to construct the world model, predictive model, and behavioral model for runway safety. Next, the developers implement the agents. In this case, aircraft and vehicle are active agents, while runway and taxiway are passive agents. We take aircraft for an example. The developers implement a concrete agent class called $Aircraft$ by extending the abstract class of active agent. The class $Aircraft$ contains the properties that an aircraft should have, and the methods to represent the actions of a aircraft should have such as takeoff and landing. Besides, the developers implement the abstract methods in abstract class of active agent, e.g., the method to gather related status. After that, the developers implement an I/O adapter for each equipment or existing system. Finally, the developers configure the ARRS. Then the ARRS for runway safety is built.

## 4.6 Implementation Issues of the Framework

The soft system buses (SSB) [3] are the infrastructure of the framework, which connects all components of the framework, by providing: (1) the way to identify and specify any component in an SSB-based system, (2) the way of self-measuring, self-monitoring, and self-recording of system states, (3) the way for components to send data/instructions to and receive data/instructions from data-instruction stations, (4) the way for components to specify partners in cooperation and communication, (5) the way for control components to manage functional components including, adding, upgrading, replacing, and removing the functional components and starting or stopping the running of them, (6) the facility of data/instruction preservation and retransmission, (7) the facility of authentication, (8) the facility of encipherment, and (9) the way to stop the interaction between an SSB-based system and its outside world [6]. A structured p2p based SSB [17] is being developed.

A predictor consists of a formula generator, a forward reasoning engine, a prediction choose, and a calculator [12]. A decision-maker consists of a formula generator, a forward reasoning engine, an action chooser, and a calculator [12]. The core of both a predictor and a decision-maker is a forward reasoning engine. FreeEnCal [9] as a forward reasoning engine with general-purpose, is a hopeful candidate.

The implementation of the agent and the agents pool is similar to the implementation“the servlet and the web container” or “the EJB and the EJB container” [2]. It is a mature technology. And the detail about the design and implementation issues can refer to [2].

The use of the (I/O) adapters is also a widely used technology. And the detail about how to design and implementation such a adapter can refer to [18].

## 5 Concluding Remarks

This paper proposed a software framework for developers of ARRSs to develop various practical ARRSs. In order to design such a framework which can be suitable to develop various ARRSs, this paper also proposed new ARRSs, which also could cover the weakness of original ARRSs to some extent. However, if there are new target domains of ARRSs emerge, it is uncertain whether the framework can handle the new target domains of ARRSs.

By using the framework, the developers of ARRSs could understand the ARRS easily, design and implement a specific ARRS easily. Because the framework specifies a template for the ARRS, based on such a template, we could develop new components of ARRSs easily.

The framework is part of development and maintenance environment for ARRSs [12]. The work is ongoing. The next step is to implement the common components and the supporting tools.

## References

[1] H. J. Ahn, H. Lee, and S. J. Park. A flexible agent system for change adaptation in supply chains. *Expert Systems with Applications*, 25(4):603–618, 2003.

[2] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *Proc. ACM Sigplan Notices*, volume 37, pages 246–261. ACM, 2002.

[3] J. Cheng. Soft system bus as a future software technology. In *Proc. 8th International Symposium on Future Software Technology*. Software Engineers Association of Japan, 2004.

[4] J. Cheng. Temporal relevant logic as the logical basis of anticipatory reasoning-reacting systems. In *Proc. Computing Anticipatory Systems: CASYS - 6th International Conference, AIP Conference Proceedings*, volume 718, pages 362–375. AIP, 2004.

[5] J. Cheng. Comparing persistent computing with autonomic computing. In *Proc. 11th International Conference on Parallel and Distributed Systems*, pages 428–432. IEEE Computer Society Press, 2005.

[6] J. Cheng. Connecting components with soft system buses: A new methodology for design, development, and maintenance of reconfigurable, ubiquitous, and persistent reactive systems. In *Proc. 19th International Conference on Advanced Information Networking and Applications*, pages 667–672. IEEE Computer Society Press, 2005.

[7] J. Cheng. Persistent computing systems as continuously available, reliable, and secure systems. In *Proc. 1st International Conference on Availability, Reliability and Security*, pages 631–638. IEEE Computer Society Press, 2006.

[8] J. Cheng, Y. Goto, and N. Kitajima. Anticipatory reasoning about mobile objects in anticipatory reasoning-reacting systems. In *Proc. Computing Anticipatory Systems: CASYS - 8th International Conference, AIP Conference Proceedings*, volume 1051, pages 244–254. AIP, 2008.

[9] J. Cheng, S. Nara, and Y. Goto. FreeEnCal: A forward reasoning engine with general-purpose. In *Proc. 11th International on Conference Knowledge-Based Intelligent Information and Engineering Systems, Part II, Lecture Notes in Artificial Intelligence*, volume 4693, pages 444–452. Springer-Verlag, 2007.

[10] J. Cheng and F. Shang. Persistent computing systems as an infrastructure of computing anticipatory systems. *International Journal of Computing Anticipatory Systems*, 18:61–74, 2006.

[11] M. Fayad and D.C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[12] Y. Goto, R. Kuboniwa, and J. Cheng. Development and maintenance environment for anticipatory reasoning-reacting systems. *International Journal of Computing Anticipatory Systems*, 24:61–72, 2011.

[13] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1):7–38, 1998.

[14] N. Kitajima, Y. Goto, and J. Cheng. Development of a decision-maker in an anticipatory reasoning-reacting system for terminal radar control. In *Proc. 4th International Conference on Hybrid Artificial Intelligence Systems, Lecture Notes in Artificial Intelligence*, volume 5572, pages 68–76. Springer-Verlag, 2009.

[15] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1994.

[16] W. Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. *Object-Oriented Programming*, pages 150–162, 1994.

[17] M. R. Selim, Y. Goto, and J. Cheng. A low cost and resilient message queuing middleware. *International Journal of Computer Science and Network Security*, 8(8):225–237, 2008.

[18] S. Van den Enden, E. Van Hoeymissen, G. Neven, and P. Verbaeten. A case study in application integration. In *Proc. the OOPSLA Business Object and Component Workshop, 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.