Int J Adv Manuf Technol (2005) 27: 211–224
DOI 10.1007/s00170-004-2169-5

ORIGINAL ARTICLE

Y.-E. Nahm · H. Ishikawa

An Internet-based integrated product design environment. Part I: a hybrid agent and network architecture

Received: 24 September 2003 / Accepted: 6 February 2004 / Published online: 2 February 2005
© Springer-Verlag London Limited 2005

Abstract During the past two decades, concurrent or collaborative engineering (CE) has presented new possibilities for successful product development. In addition, the advances in computer networks and information technology have brought engineering design context into a new era. As a promising approach to accommodate the radical pace of changes in the context of engineering design, agent technologies have been attracting public attention and are being used in an increasingly wide variety of applications. However, little attention has been paid to multi-agent system (MAS) frameworks for CE environments that enable systematic and timely design integrations in both hierarchical and heterarchical design topology. This is the first of a two-part paper proposing a MAS framework for integrated product design in a computer network-oriented CE environment. Part I first proposes a hybrid agent network architecture to develop lightweight, dynamic and large-scale distributed systems, and then proposes a hybrid agent architecture based on our finite state automata (FSA) formalism that can exhibit both hybrid behaviours and hybrid interactions. Finally, some ideas for building an integrated product design environment are presented using the proposed agent and network architecture. Part II discusses the applications of the proposed MAS framework to concurrent engineering design.

Keywords Concurrent engineering · Design decomposition and integration · Finite state automata · Multi-agent system · Network-oriented CE environment

1 Introduction

Since the late 1980s, *concurrent* or *collaborative engineering* (CE) has brought new possibilities for realising faster product development, higher quality, lower costs, improved productivity, better custom values, and so on [1, 2]. CE attempts to incorporate various product lifecycle functions from an earlier stage of the product development process. This approach is thus intended to encourage the design engineers, to consider all elements of the product lifecycle from the outset, thus imposing a heavy burden on the design engineers [3–5]. As a result, the CE approach increases the complexity of design problems and necessitates different expertise from a variety of engineering fields.

Due to the increasing complexity of the design problem, it becomes more difficult for a single designer to propose a design solution as global optima that satisfies all of the requirements within a given timeframe. Design decomposition has been a common practice to reduce the complexity and facilitate the concurrency of the design problem [4]. A complex engineering design problem is recursively decomposed into a manageable number of sub-problems, each representing sub-systems (or units, parts) or different perspectives (or disciplines). Then, the design decomposition causes the *hierarchical* and *heterarchical* design topology. The decomposed sub-problems are assigned to different design participants who are geographically dispersed and diverse in their disciplines. Since each participant has a different point-of-view and has limited expertise, the sub-problems are usually interrelated to each other to exchange various sources of design information necessary for individual design problems (i.e. heterarchical design topology). Also, a large problem can be formulated in terms of interacting smaller sub-problems that encapsulate specific aspects of the large problem (i.e. hierarchical design topology). Then, the results of smaller problems are recombined to formulate solutions to the larger problems, along with the problem hierarchy. Accordingly, a complex engineering design problem should be solved by a network of linked design problems with hierarchical and heterarchical interdependencies.

In addition, the advances in computer networks and information technology have brought engineering design context into a new era. The Internet has been validated as an information infrastructure for collaborative engineering and has been utilised as a facilitator for collaborative activities [6]. Therefore, it is becoming common that the decomposed design problems are

Y.-E. Nahm · H. Ishikawa (✉)
Department of Mechanical Engineering and Intelligent Systems,
The University of Electro-Communications,
1-5-1 Chofugaoka, Chofu-shi, 182-8585 Tokyo, Japan
E-mail: ishikawa@mce.uec.ac.jp
Tel.: +81-424-435422
Fax: +81-424-843327

distributed over a networked environment such as the Internet, intranet or World Wide Web (WWW). In such a distributed design environment, different aspects of a problem are modelled and solved by different people who have special expertise. In solving individual design problems, design participants use different engineering resources (e.g. problem-specific engineering models, applications, pre-existing software tools, legacy codes, design data, etc.) that are intrinsically heterogeneous. Since a wide variety of engineering resources are used at geographically distributed locations, the heterogeneous resources need to be integrated or translated to facilitate interoperation and collaboration over the networked environment.

Consequently, the engineering product development process in such a network-oriented CE environment can be viewed as: problem decomposition of the product design project and aggregation of sub-problems by multi-disciplinary or multi-talented teamwork; a network of linked decisions with hierarchical and heterarchical interdependences; resource-sharing and the exchange of information between team members; and design conflict detection and process coordination. There have been many research efforts to address all of these basic tracks. Recently, agent technology has been considered as an important approach for developing industrial distributed systems [7, 8], and has already been applied to the many intelligent manufacturing systems for the last ten years [9]. In particular, the approach of multi-agent systems (MAS) has been proven to be an effective way to develop large distributed systems [10, 11].

Many agent-based approaches have been applied to information sharing and teaming [12–18], multi-enterprise integration [10, 11], supply chain management (SCM) [9, 19], concurrent design and manufacturing [20, 21], planning and scheduling [22–25], negotiation and coordination [4, 19, 25–27], tools for multi-media conferences [28], and so on. However, few significant research results have been reported for applying agent technology to the integrated product design environment that enables the hierarchically and heterarchically distributed team members to integrate various engineering resources in a systematic and timely fashion. Our claim is that this problem is mainly due to the lack of agent architecture suited to various engineering resources and their interactions modelling, and of agent network architecture more adaptive to the dynamic random changes.

This paper proposes a MAS framework, called the agent network concurrent design environment (ANetCoDE), for integrated product design in the computer network-oriented CE environment. The need for new agent and network architectures is first addressed by identifying the shortcomings of state-of-the-art research in Sect. 2. Sect. 3 proposes the agent network architecture for building a lightweight, dynamic and large-scale MAS, as well as a clear formalism for the agent architecture that enables agents to exhibit hybrid (both continuous and discrete) behaviour and hybrid interaction, by using finite state automata (FSA) theory. Based on the proposed architectures, the ANetCoDE is developed, and several ideas for systematic and timely design integration are addressed in Sect. 4. Sect. 5 gives an overview of an implementation perspective, and Sect. 6 concludes the paper.

2 Need for integrated product design environment

2.1 Integrated product design environment

As shown in Fig. 1, a complex engineering design problem can be decomposed into a manageable number of sub-problems, based on the product decomposition (e.g. product modularity and

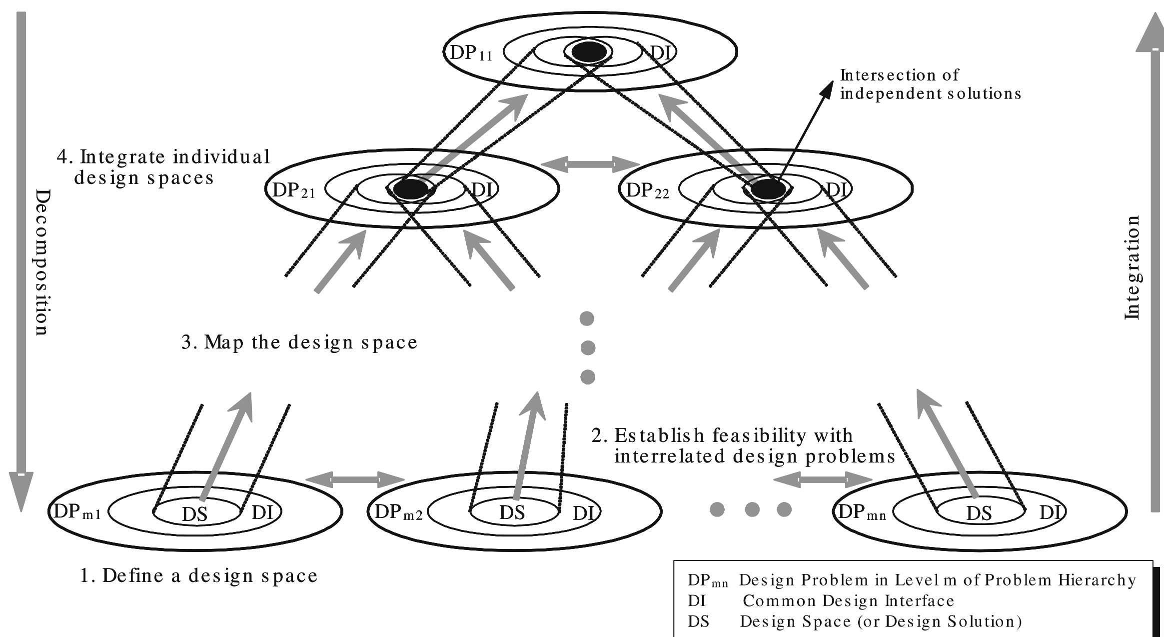


Fig. 1. Hierarchical and heterarchical design topology

structural decomposition) and process decomposition (including upstream and downstream processes, or multidisciplinary perspectives such as structures, kinematics, dynamics, etc.) [29]. Sub-problems are interrelated and may share the same kinds of design interfaces (DI), by representing subsystems (or units, parts) or different perspectives (or disciplines). Therefore, a design problem (DP_{11}) is made up of the smaller sub-problems (DP_{21} and DP_{22}) and one or more design interfaces. Then, a non-terminal design problem (e.g. DP_{21}) can be formulated as a decision network of multiple interacting design aspects (or a manageable set of design variables) and defines the feasible design space (or solutions). When defining the design space, the design problem establishes the feasibility of the defined solutions by communicating the design possibilities with the neighbouring design problems and by integrating the design spaces mapped from its child design problems. Further, the design solutions are recursively mapped onto the parent design problem in order to create a global solution that can be accepted from all of the sub-problems. As a result, design decompositions cause the hierarchical and heterarchical design topology.

Although design decomposition facilitates the concurrency of product lifecycle functions from an earlier stage of design and reduces the complexity of the design problem, it necessitates various integration efforts for recombining the hierarchically and heterarchically decomposed design problems. Then, the hierarchical and heterarchical design integrations can be made by mapping the individual design solutions onto commonly interacting design spaces and by communicating the design possibilities through accessible design interfaces with neighbouring design problems, respectively.

2.2 Shortcomings of state-of-the-art researches

2.2.1 Network architectures for MAS

The aforementioned integrated product design environment involves interactions between different people or organisations with different (possibly conflicting) goals and proprietary information. The MAS approach is ideally suited to represent problems that have multiple problem-solving methods, multiple perspectives and/or multiple problem-solving entities. Since agents are relatively independent pieces of software interacting with each other through message-based communication, system development, integration and maintenance become easier and less costly. The multi-agent-based solutions can be designed and implemented as several interacting agents. When talking about MAS, two components thus have to be described: the agent architecture and network architecture.

As shown in Fig. 2, the existing network architectures for MAS can be classified into three categories: centralised approach; federation approach; and autonomy-based approach.

The centralised architecture probably is the most widely applied approach for developing the network-oriented collaboration systems as well as MAS [9, 30, 31], although it may be criticised for its bottleneck problem of data and control [13], inadequacy with regard to loose and flexible collaboration in

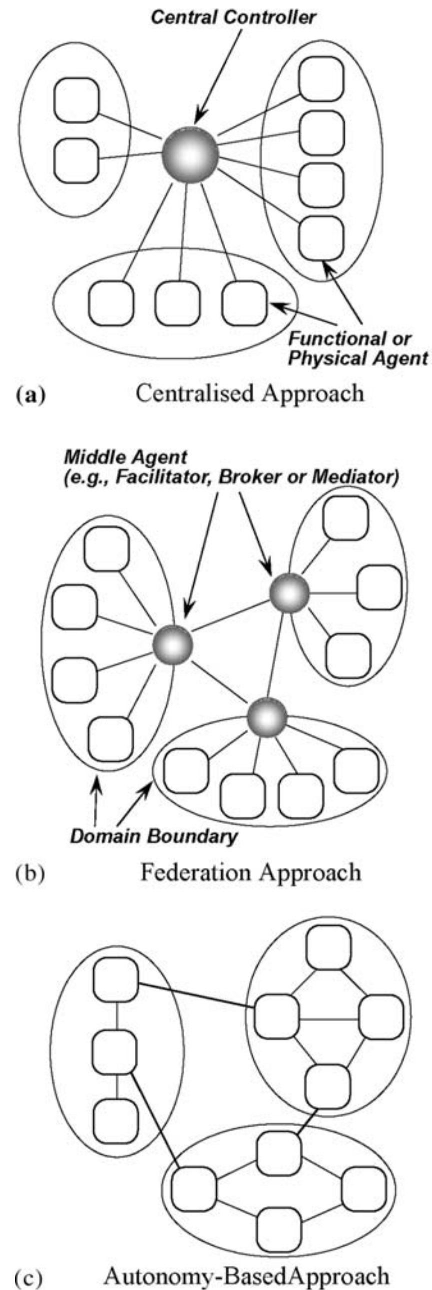


Fig. 2. Existing network architectures for distributed systems

an open environment [31, 32], low scalability [13], and the system safety and stability problem of being shut down by a single point of failure [9, 32], etc. An example of this type of approach is a distributed and integrated environment for computer-aided engineering (DICE) project [33]. The DICE project provides a centralised multi-user system for cooperation and coordination among multiple designers working in separate engineering disciplines using the shared workspace model and object-oriented database management systems.

Next, the federation architecture is a distributed system approach, where usually there is no central module. Instead, this

approach coordinates multi-agent activities via facilitation as a means of reducing overheads, ensuring stability and providing scalability [9]. Communication and coordination between a local collection of agents and remote agents are therefore made only via middle agents (e.g. facilitators, brokers, or mediators) that usually provide the following services: (1) message routing and translation; (2) agent behaviour monitoring and notification; and (3) coordination between multi-agent activities. Using this approach, the Palo Alto Collaborative Testbed (PACT) experiments demonstrate how encapsulating engineering tools and frameworks by using agents that exchange information and services through facilitators enable them to interoperate with one another, even though they were developed with no anticipation of subsequent integration [13]. Here, an agent is regarded as a computer program that communicates with external programs exclusively via predefined protocols such as KQML [14] and KIF [15]. Although they provide a good foundation for developing open, scalable multi-agent-based network architectures, many interpreters are required between different domain systems that adopt the different communication and content-level languages [34]. As in the centralised approach, the bottleneck problem still remains if a lot of agents are connected to a facilitator.

More recently, the autonomy-based approach has been proposed based on the concept of autonomy, using distributed computing and component technologies [31, 34]. The distributed object modelling and evaluation (DOME) framework forms a service exchange network by connecting interacting components that encapsulate specific aspects of a design problem [31]. Although powerful in integrating various engineering models transparently, the framework is likely to be only suited to the well-structured and conflict-free problem. Since it mainly focuses on how an integrated model can be modelled by combining a number of distributed modules, the topology of the module network is fixed and remains unchanged as the system operates. Because all of the problems cannot be always well-structured or formalised, design participants seek to discover new relationships and configure themselves as the system operates. A similar approach is found in the component agent-based design-oriented model (CADOM) program [34]. In this system, the component agents (CAs) encapsulating common building elements (such as the wall, beam, slab, etc.) are distributed over the Internet, and the assembly shops (ASs) assemble the predefined and distributed CAs to produce a particular project using the web-based interface manager and design session manager. Since that system basically follows the ideas of DOME, the similar drawbacks also exist. Also, by encapsulating engineering tools and human specialists as an agent, the distributed intelligent design environment (DIDE) architecture attempts to integrate engineering tools, like CAD/CAM tools, database systems, or knowledge-based systems, in an open environment [17].

Since there is no central controller or middle agents, the functional or physical agents are empowered to manage most of the activities related to their own goals and tasks through intensive inter- and intra-agent communication. This type of architecture is well-suited for developing distributed intelligent design systems in an open, dynamic environment. However, it is not easy to test the overall behaviour of the MAS [18]. Also, it is difficult to produce a globally optimal solution since each local agent has an equal right to compete with one another. No agent has the right to decide on the best strategy [30]. What is worse, the circular dependencies among interacting agents may give rise to serious trouble. It is difficult to find a circular dependency when distributed agents are utilised since there is no central observer [31]. Therefore, there is no theoretical guarantee that the process will ever converge.

Based on the product decomposition and marketplace dynamics, the responsible agent for product-process integrated design (RAPPID) framework forms a hierarchical topology of component agents, each with specified characteristics (or a definable attribute or parameter of a component) [35, 36]. The component agents buy and sell units of these characteristics, taking into account its own resources. Although it presents a promising vision to the market-based design environment, it is not clear how the heterogeneous design tools can interoperate with one another, or how a variety of engineering information and multidisciplinary knowledge can be defined, shared and managed among heterarchically distributed team members.

2.2.2 Agent architectures for modelling engineering resources and their interactions

In addition to the network architecture, building agent-based solutions require the agent architecture to model properties or behaviours of the agent itself. The ideas of finite state automata (FSA) theory had and still have strong influence in different areas of human activities like technology, sociology and biology. FSA could be also considered canonical representations of computational agents. FSA are a widely known and simple but powerful formalism to model the behaviour of a system as a sequence of transitions.

Examples include Mealy automata, Moore automata, timed automata [37], I/O automata [38], hybrid I/O automata [39], interface automata [40], and so on. Many Mealy-type, Moore-type or hybrid hidden Markov models (HMMs) have been proposed to model agents and their interactions [41]. Alur and Dill [37] proposed timed automata, which impose timing constraints on state transitions, to model the behaviour of real-time systems over time. Lynch and Tuttle [38] introduce an event-driven model of asynchronous distributed computation, the input-output automaton. The I/O automata consist essentially of states, start states and discrete transitions. Discrete transitions are labelled by discrete actions that include mutually disjoint sets of input, output and internal actions, where input actions are required to be enabled from every state (i.e. input-enabled). In other words, since a process must be able to accept any input at any time, a very strong distinction is made between those actions locally-controlled by the system itself (output and internal actions) and those actions controlled by the system's external environment (input actions). More recently, the I/O automata model is augmented by adding explicit trajectories [39]. The state of the hybrid I/O automata can therefore change in two ways: discrete transitions, which make it possible to syn-

chronise the transitions of different hybrid I/O automata, and change the state atomically and instantaneously; and trajectories, which describe the evolution of the state over intervals of time.

Models of computation can be classified into two categories: component models that specify how components behave in an arbitrary environment, and interface models that specify what the components expect from the environment. Component models like (hybrid) I/O automata are deemed as pessimistic in that they assume the environment is free to behave as it pleases, and two components to be composed are compatible only if no environment can lead them into an error state. Alfaro and Henzinger proposed a lightweight formalism, interface automata, to model component interfaces with the capability of capturing temporal aspects of interfaces [40]. The interface automata are based on an optimistic approach in which components are usually designed under assumptions about the environment, and two components are compatible so long as there is some environment in which they can work together.

Although a couple of variations of FSA have been proposed to model components and their interactions, there is no FAS model or framework that is able to explicitly represent both hybrid (i.e. continuous and discrete) behaviour and hybrid interaction, and to dynamically adapt its behaviour and interaction style to the environmental random changes.

2.3 Need for a new agent and network architecture

As mentioned above, a complex engineering design problem should be solved by a network of linked design problems with hierarchical and heterarchical interdependencies. Since a large problem can be formulated in terms of interacting smaller subproblems, the results of smaller problems need to be systematically recombined to formulate solutions to the larger problems. On the other hand, the dependencies between the heterarchically distributed design problems seem to occur more dynamically. Although there already exist a lot of network architectures for building MAS, most approaches consider either hierarchical or heterarchical topology. Also, there is no lightweight architecture that can provide flexibility for agent consideration, plug-and-play and agent interaction while adapting to dynamic topological changes.

In addition, the difficulties in integrating a variety of engineering resources mainly stem from the heterogeneities of interfaces to access them and internal models of computation to describe their behaviour. As in most research, the interfacing problem could be solved by providing functional agents, where different engineering models or tools are encapsulated with transparent interfaces network-accessible from other agents. However, heterogeneous internal computational models still make the integration difficult. Models of computation include the continuous-time model, discrete-event model, the discrete-time model and the synchronous/reactive model [42, 43]. In addition to the network architectures, much agent-based research has suggested a lot of agent architectures for modelling either a continuous or discrete behaviour. However, for most real-world engineering tools or models, neither a purely continuous behaviour model nor a purely discrete behaviour model is appropriate.

Due to these shortcomings of current agent and network architectures, most agent-based solutions are still ad hoc, and therefore, they may not be appropriate for different systems and domains. As a result, few comparable research results have been reported for applying agent technology to the integrated product design environment that enables systematic and timely design integrations in the hierarchical and heterarchical design topology. Therefore, there is strong need to develop a new agent and network architecture.

Our solution to agent network architecture is to hybridise the autonomy-based and federation approaches, and doing so could compensate for the limitations between the two (see Sect. 3.1). In addition, the most promising solution to agent architecture is the hybrid agent model that can exhibit a combination of continuous and discrete behaviour. Besides the hybrid agent behaviour, the agent should be able to change its interaction mode (i.e. either continuous or discrete communication) according to the agent behaviour or unexpected environmental changes (see Sect. 3.2). Such a new agent and network architecture can effectively support the development of a MAS framework for implementing the integrated product design environment (see Sect. 4).

3 A multi-agent system architecture

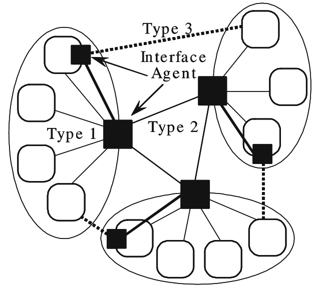
3.1 Proposed hybrid network architecture

This section proposes a hybrid agent network architecture that marries the best aspects of the federation and the autonomy-based approaches into the so-called autonomy-based federation approach. While most federation and autonomy-based approaches adopt either message-based or invocation-based communication models, the proposed approach enables a combination of message-based and invocation-based communication models that are dynamically switched according to the random environmental changes.

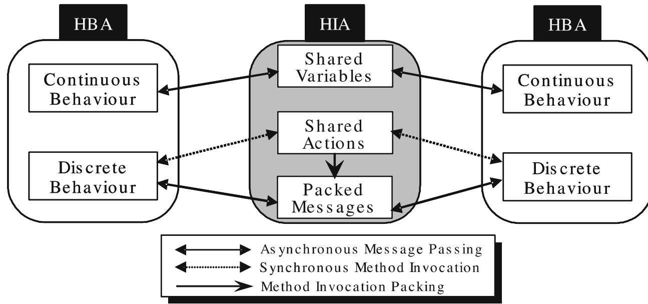
As shown in Fig. 3a, we first employ a lightweight middle agent, called an "interface agent", for continuous or discrete interactions between local agents (type 1), between a collection of local agents and a collection of remote agents (type 2), and between a local agent and a remote agent (type 3). Therefore, agent interactions are made only via the interface agent.

A multi-agent organisation can also be structured in a hierarchical form depending on the size of the organisation and complexity of design problem. Then, the interface agent serves as a bridge between hierarchically and heterarchically distributed agents or agent teams. Any functional agents are thus pluggable into the different domains, only if the interface agents that defines how to access the functional agents are network-accessible. As compared with facilitators, brokers and mediators, the interface agent can be characterised by the following features:

- Unlike the federation approach, discrete and continuous interactions are distributed over a lot of lightweight interface



(a) An Autonomy-Based Federated Network Architecture



(b) A Hybrid Agent Architecture

Fig. 3. Proposed network architecture and agent architecture

agents. In the case of type 1, it simply provides shared variables for message-based interactions, instead of providing information routing and translation services as in the existing middle agents. Message is stored directly on the shared variables of interface agent and viewed directly by the functional agent. Information need not be explicitly passed to and from functional agent since the agent can access it directly. The different agents can see current variable values without update delays. Exchanging the messages is therefore the responsibility of autonomous functional agents, not the interface agent, thereby reducing the communication overheads of the interface agent. During the message-based interactions of type 2, a registration mechanism is provided for finding appropriate agents or services, thereby enabling dynamic system reconfiguration. That is, the important feature of interface agent is to know how to find other agents, thus serving as a yellow page server. Once the information (e.g. physical address, role, interface, etc.) about other agents is obtained, functional agents can directly make invocation-based interactions through peer-to-peer conversations (type 3), thus also eliminating the communication overheads of the interface agent.

- Real-world design environments are highly dynamic because of diverse frequently changing situations. For example, there are no obligations for functional human or computerised agents to remain with a design network for a certain time period [19]. In addition, network-based large-scale systems are more prone to random changes of their environment (e.g. temporary unavailability of servers or service agents, transmission channel alterations or break-downs) [46]. Our approach can change the type of communication between the communicating agents. For example, when the synchronous invocation-based communication (type 3) has failed, our interface agent initiates an asynchronous message-based communication with the remote agent. As drawn by bold lines in Fig. 3a, it intercepts the invocation, packs it under the form of message and stocking it in the different interface agent. In this manner, functional agents are able to interact in a highly dynamic way. In our approach, functional agents can join in, stay, or leave the system. When a design task is allocated, a virtual design network between functional agents may emerge from the system through autonomous design process between functional agents.

Consequently, our approach can be viewed as a lightweight and flexible network architecture for a MAS framework that can provide flexibility for agent consideration, plug-and-play and dynamic agent interaction in a networked environment.

3.2 Proposed hybrid agent architecture

Real-world systems are intrinsically hybrid systems that exhibit a combination of discrete and continuous behaviour [39]. For a more complete model of real-world systems, a system component should therefore be able to represent both discrete and continuous behaviour. The component then needs to make discrete and continuous interactions with different components, in order to perform the hybrid behaviour. Therefore, we believe that hybrid behaviour model and hybrid interaction model are indispensable in describing real-world components. This section proposes a hybrid agent model, called Hagent, which consists of hybrid behaviour automata (HBA) for modelling individual agents (i.e. the component itself) and hybrid interaction automata (HIA) for modelling the interaction of agent team (i.e. the component interface and environment).

3.2.1 Hybrid behaviour automata (HBA)

In this paper, an agent is defined as a reactive and proactive system that has the data and code encapsulation, its own thread of control, and the ability to execute autonomously (or responsively) without (or with) external invocation. Agents are also resource-bounded reasoners that have some ability to reason about their beliefs, based on the finite computational resources and a set of explicitly represented beliefs [44]. To realise an intelligent behaviour with the resource-bounded reasoning capability, we advocate that it is necessary for the agent to possess a set of explicitly represented states and a repertoire of possible behaviours.

Also, interactions between different agents can be generally classified into two models: (local or remote method) invocation-based interaction and message-based interaction. Using a hybrid combination of the two types of interaction models, our HBA model attempts to exhibit a combination of discrete and continuous behaviour. First, from the viewpoint of agent reactiveness, it seems to be persuasive that discrete behaviour can be described by the triplet <input_action, internal_action, output_action> by considering synchronous (or possibly asyn-

chronous) invocation-based interactions that consist of the following steps: 1) the caller invokes an input action (i.e. function or subroutine) on the callee; 2) the input action triggers some internal actions for internal computation; and 3) the callee invokes an output action in order to inform the caller of the termination of the task or (possible) requested information [29]. By adopting a black box approach, the triple can be rewritten by the pair <input_action, output_action>. Since the input/output action pair alters the internal states, a discrete transition can be denoted by the quad (previous_state(s), input_action(i), output_action(o), next_state(s)). We call this possible transition a repertoire fragment. A repertoire is then constructed from a finite sequence $s_0 i_1 o_1 s_1 \ldots s_{k-1} i_k o_k s_k$ of alternating states and actions beginning with a start state (s_0) such that each quad $(s_j, i_{j+1}, o_{j+1}, s_{j+1})$ is a repertoire fragment.

Second, from the viewpoint of agent proactiveness, continuous behaviour means that an agent can autonomously update its internal state variables and external state variables over intervals of time, without being invoked externally. That is, continuous behaviour is proactive, rather than reactive (or interactive) as in discrete behaviour. While the input actions of discrete behaviour are triggered by the external agents, all actions of continuous behaviour are locally controlled. Continuous behaviour is based on asynchronous (or possibly synchronous) message-based interaction. Therefore, continuous behaviour can be described by the triplet <previous_state, action, next_state>, not by the input/internal/output action triplet. Then, the internal or external state vector ($s(t) \in S$) can be determined by an internal action or external action (i.e. input and output actions) ($a \in A$), respectively, at time t ($\dot{s}(t) = a(s(t))$). Although a continuous transition may be commonly described in differential equations, it is more tractable to design it as a discrete transition that changes the state at regular intervals of δt, $s(t + \delta t) = a(s(t))$, when realising it using digital circuits and computers [45]. Our HBA model thus defines a continuous transition as the quintuplet (start_time(t), previous_state(s), action(a), next_state(s), end_time(t)). The repertoire fragment for continuous behaviour is slightly different from that of discrete one. A repertoire fragment (rf) is described as a finite sequence $t_0 s_0 a_1 s_1 t_1 \ldots t_{k-1} s_{k-1} a_k s_k t_k s_0$ of alternating states and actions beginning with a start state (s_0) at time t_0, ending with a state (s_k) at time t_k, and returning to the start state (s_0), which can be simulated by a single-loop program. Then, the duration of time ($\delta t'$) required to a repertoire fragment execution is calculated by $t_k - t_0$. A repertoire (R) can have one or more repertoire fragments with regular intervals of time (δt) that can be explicitly specified by users, denoted by the form of $R = < \{rf_1, rf_2, \ldots, rf_i\}, \delta t >$. In particular, we call the repertoire fragment of $s_k = s_0$ the "final fragment", which means that a repertoire is fully completed. As a result, an HBA triggers a possible repertoire fragment at regular intervals of δt, and δt may be replaced by $\delta t'$ if $\delta t - \delta t' < 0$. According to required behavioural aspects, more than one repertoire can be embedded into the HBA.

Definition 3.1: An HBA $A = < I, O, P, S, S_{\mathrm{INIT}}, A_D, A_C, R_D, R_C, W, \Delta T >$ consists of the following elements:

- I, O and P are mutually disjoint sets of input, output and internal (or private) variables, respectively. For a clearer separation with internal variables, we denote the set of all input and output variables by a set $E(= I \cup O)$ of external variables. Strictly speaking, E may be local copies of a part of "external world" state variables, ES. Also, the set of all variables is denoted by $V = I \cup O \cup P$.
- S and $S_{\mathrm{INIT}}(\subseteq S)$ is a set of states and a set of initial states, respectively. S is partitioned into a set $S_I(\subseteq \mathrm{assign}(P))$ of internal states and a set $S_E(\subseteq \mathrm{assign}(E))$ of external states. assign(P) is used to denote the value assignments into a set P of variables.
- A_D and A_C are a set of action primitives for discrete behaviour and a set of action primitives for continuous behaviour, disjoint from each other. A_D (A_C) is further partitioned into a set A_{DI} (A_{CI}) of input actions, a set A_{DO} (A_{CO}) of output actions and a set A_{DP} (A_{CP}) of internal actions. Then, A_D and A_C are denoted by $A_D = A_{DI} \cup A_{DO} \cup A_{DP}$ and $A_C = A_{CI} \cup A_{CO} \cup A_{CP}$, respectively.
- R_D ($\subseteq S_I \times A_{DI} \times A_{DO} \times S_I$) is a set of discrete transitions, with the property that, for a given pair $< a_{di}(\in A_{DI}), a_{do}(\in A_{DO}) >$ and a state $s(\in S_I)$, there is a transition of the form (s, a_{di}, a_{do}, s). An element (s, a_{di}, a_{do}, s) of R_D is referred to as a repertoire fragment (rf) of repertoire (R).
- $R_C(\subseteq T \times S \times A_C \times S \times T)$ is a set of continuous transitions at time T. If $a \in A_{CI}$, A_{CO} or A_{CP}, then (t, s, a, s', t') is called an input, output, or internal transition step, respectively. Slightly different from the discrete transition, a finite sequence $t_0 s_0 a_1 s_1 t_1 \ldots t_{k-1} s_{k-1} a_k s_k t_k s_0$ of R_C is referred to as a repertoire fragment, where s_0, t_0 and t_k are a start state, start time and end time, respectively. Note that the HBA removes the dual use of external variables (E) for discrete and continuous transitions. If a discrete transition could change an input or output variable, it could cause an incorrect change of continuous transition.
- W makes a dynamic choice of repertoire, through collaboration with HIA.
- ΔT is a timing interval specified by users. In the continuous behaviour mode, HBA continuously change the internal and external states at regular intervals of ΔT. This interval of time can be dynamically changed in comparison with the real duration of time required to the repertoire fragment execution.

3.2.2 Hybrid interaction automata (HIA)

As mentioned above, both invocation-based models and message-based models are popular communication mechanisms for modelling interactions between distributed systems or system components [46], through shared actions (i.e. public procedures or functions to be invoked) and shared variables (i.e. public variables to be referenced) that can be used to model both discrete and continuous interaction [39].

In the previous section, we mentioned the importance of the representation of hybrid behaviour. Since a component can exhibit a combination of discrete and continuous behaviour, a

component should be dynamically able to switch its communication mode according to the behaviour type executed by different components. In order to offer the capability to adapt the communication model to the execution context of an component as well as changes in its communication environment, instead of using the direct communication approach between components (e.g. I/O automata and hybrid I/O automata [38, 39]), we employ a kind of interface automaton, HIA, through which components indirectly communicate with each other.

The proposed HIA model is somewhat similar to interface automata [40] in that an interface component is explicitly defined to model interactions between components, but offers quite different functionalities. The interface automata aim at automating the component composition for component-based design, via automatic compatibility checks between component interfaces. It captures both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. On the other hand, HIA focus on modelling interaction protocols for collaboration between components (i.e. cooperation, competition and coordination), thus facilitating agent-based design. Such interaction protocols can be modelled by shared variables or shared actions. Thus, a shared variable itself can be a finite state automaton. For example, shared variables could play a critical role in integrating different types of solutions or the same types of solutions with different values from multiple HBA; that is, solution integration (see Sect. 4.3.1).

In addition, for a better and more transparent understanding of component behaviour, a clearer separation is made between the mechanisms used to model discrete and continuous interaction between components. That is, the dual use of shared actions (or shared variables) for discrete and continuous interaction is not allowed. Our HBA use shared actions for synchronous discrete interaction and shared variables for asynchronous continuous interaction (see Fig. 3b). HIA also provide a mechanism for dynamic switching of the communication models used between HBA. Although it is similar to [46], our HIA provide a clear formalism. When a caller HBA tries to invoke an input action of a callee HBA via a shared action of the HIA, and the callee HBA is under the continuous behaviour, the HIA pack the invocation under the form of a message and stock it in a message queue. When the callee HBA is later ready to accept invocations, it can contact the message queue and thereby retrieve the packed invocation message. After unpacking the message, it can return the invocation result to the caller HBA.

Definition 3.2: An HIA $I =< V, S, S_{\mathrm{INIT}}, A, R, Q >$ consists of the following elements:

- V is a set of shared variables, through which HBA autonomously make continuous interactions by communicating their own data at regular intervals of time. V may be further finite state automata that consist of possibly multiple input data (V_{IN}), single output data (V_{OUT}), a set of states (V_S), a set of actions (V_A), and a set of state transitions ($V_T \subseteq V_{\mathrm{IN}} \times V_S \times V_A \times V_S \times V_{\mathrm{OUT}}$) denoted by $V =< V_{\mathrm{IN}}, V_{\mathrm{OUT}}, V_S, V_A, V_T >$.
- $S(\subseteq \mathrm{assign}(V))$ and $S_{\mathrm{INIT}}(\subseteq S)$ is a set of states and a set of initial states, respectively. $\mathrm{assign}(V)$ is used to denote the value assignments into a set V of variables.
- A is a set of shared actions, through which HBA make discrete interactions with each other.
- $R(\subseteq S \times A \times S)$ is a set of state transition steps.
- Q is an intercept mechanism for dynamic switching of communication models used between interacting HBA. Q consists of a sequence of actions (Q_A) for packing an invocation under the form of a message and a set of message queues (Q_M) for stocking the packed message.

4 ANetCoDE framework for integrated product design

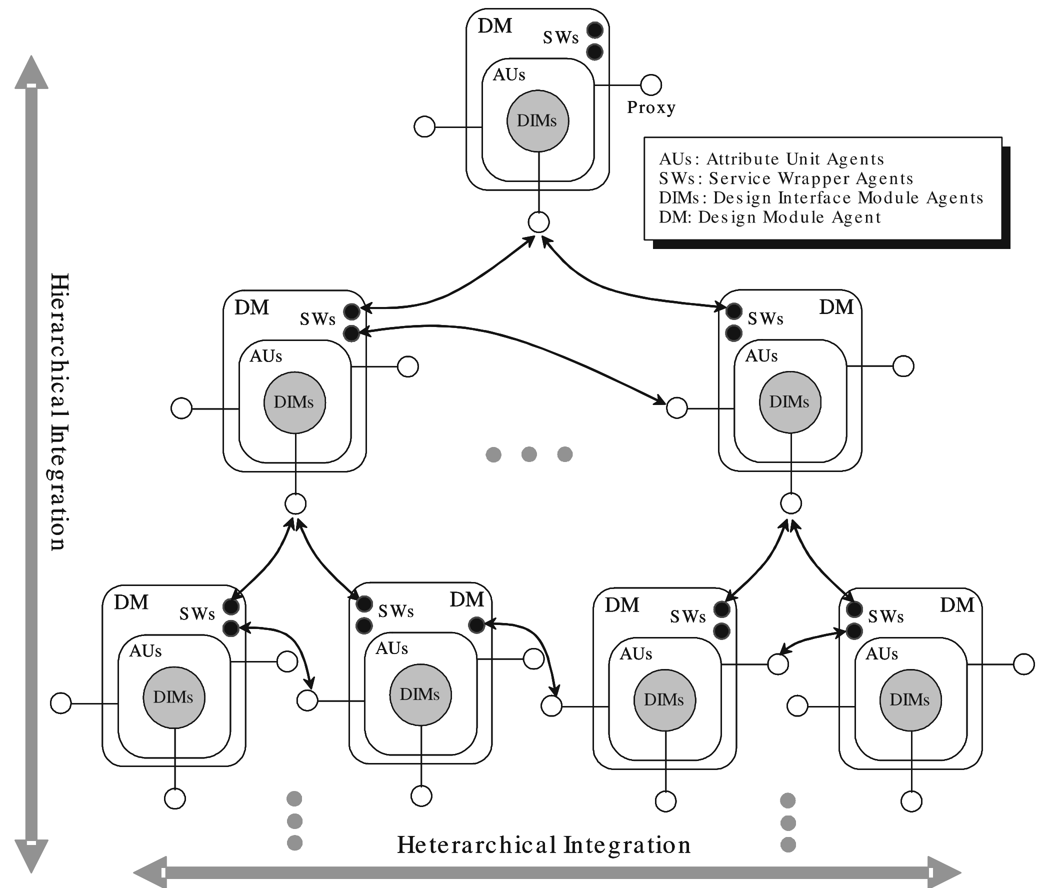
Based on the proposed agent and network architecture, we develop a MAS framework, ANetCoDE, for the integrated product design in a networked environment.

4.1 Design agents

This section gives an overview of the design agents for the ANetCoDE. As shown in Fig. 4, it is assumed that decomposed subproblems can be structured in a network form of four different types of design agents in the hierarchical and heterarchical design topology:

- An attribute unit agent (AU-Agent) is a kind of HBA that represents a specific aspect of product and process knowledge, and by default initiates continuous behaviour for its own design task including design synthesis, selection, evaluation, critique, and so on. The continuous and discrete interactions between AU-Agents are made through shared variables and shared actions that could be described as slot variables and public methods of a design interface module agent (DIM-Agent), respectively. In the continuous behaviour and interaction mode, AU-Agent can be viewed as an n-bit processor, whose design task is dependent upon the active bits on the shared bus by the other agents (cooperation) and the other agents having the same bits active (competition). When a competition occurs, DIM-Agent may initiate a sequence of coordination process. Since such a process is interactive rather than transaction-based, DIM-Agent makes the conflicting AU-Agents shift their behaviour and interaction mode to discrete one. In this paper, continuous and discrete behaviour are thus regarded as transaction-based and interactive process, respectively. Figure 5 shows the pictorial representation of a repertoire for continuous behaviour that is intended to describe the cooperation between AU-Agents in the local problem domain. For example, the repertoire fragment, rf1, of this repertoire continuously checks whether the required inputs are available at its real-time interval. If exist, the next repertoire fragment, rf2, is executed. This continuous behaviour is asynchronously performed. In another

Fig. 4. Agent network concurrent design environment (ANetCoDE) for hierarchical and heterarchical design integrations



repertoire, designers can also publish their own AU-Agents to the remote design domains by making the AU-Agents available and accessible over the Internet.

- A design interface module agent (DIM-Agent) is a kind of HIA for establishing inner-domain or inter-domain communication channels between AU-Agents. A DIM-Agent explicitly defines interaction spaces (i.e. a set of shared variables, V, of HIA) for the shared and network-accessible agent interactions, integrates a number of design solutions assigned to the shared variables, and coordinates the conflicting design solutions through a coordination protocol (i.e. a set of state transition steps, R, of HIA) that are modelled using shared actions (A). The interaction space itself is further a finite state automaton that provides the integration functionalities such as suppression mapping, inhibition mapping, joint mapping and functional mapping (see Sect. 4.3.1). As mentioned in Sect. 3.1, the DIM-Agent is a lightweight middle agent and also serves as a message server by intercepting a failed method invocation and packing it under the form of the message, thus enabling dynamic switching of communication models used by AU-Agents.
- A service wrapper agent (SW-Agent) is a kind of HBA that serves as a bridge between the different domains. With an SW-Agent, designers can define both input and output action, through which shared actions or shared variables of local or remote agents are called. Thus, SW-Agent facilitate the integration of inter-domain tasks by plugging the remote AU-Agents published from the different divisions into the local domain (local AU-Agents or local DIM-Agents) or providing its own data objects with the remote AU-Agents or DIM-Agents. The SW-Agent then ensures the interoperability between the different types of data objects by defining a set of transformation functions between input and output.
- A design module agent (DM-Agent) is a kind of HBA that corresponds to a design division which takes charge of a partial portion of the entire design problem. The DM-Agent contains several functional modules for its own tasks. Such functional modules include a set of HBA for design process control, problem-solving, decision-making, etc., and a set of HIA for interactions between the HBA. This paper focuses on the problem-solving characteristics of the DM-Agent. During the design process, the DM-Agent produces an evolving collection of AU-Agents. The DM-Agent recursively aggregates a set of runnable AU-Agents, based on the input-output compatibilities between AU-Agents and the active data objects from their shared variables. That is, if all the required inputs of an AU-Agent are available, the AU-Agent can run immediately. It is therefore assumed that the problem-solving structure of a DM-Agent can be represented in the form of a network of AU-Agents.

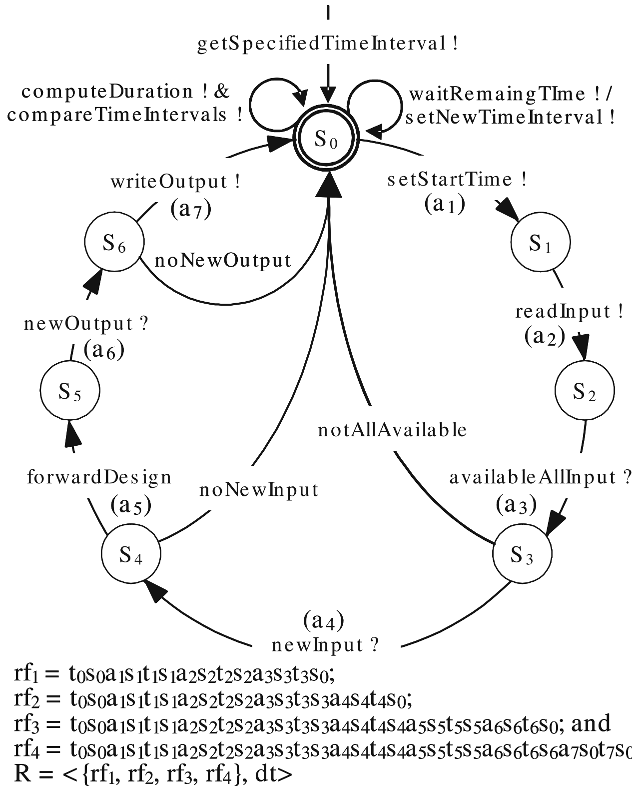


Fig. 5. An example of repertoire for continuous behaviour, called inner-problem cooperation

4.2 Communicating design objects

Most research in engineering design context takes a design variable as a triplet that consists of a unique name, type and value. Instead, a new design object communicated between the agents is proposed to provide richer semantics, thus enabling design agents to make intelligent inferences. Table 1 shows the attributes of design objects.

The design objects are classified into four types: a decision design object on which design agents are allowed to make decisions independently during the design process and under their controls; a performance design object which is used in the design process that measures the performance of current design; a specification design object which is specified usually in advance of a design process that needs to be satisfied for a feasible or acceptable design; and an intermediate design object. Designers can also express a preference structure on a design object. If a designer, for example, imposes the trapezoid function as a preference structure onto a set-valued design object $< v_1, v_2, v_3, v_4 >$, the values between v_2 and v_3 will always be preferable to the designer. The degree of influence describes the sensitivity of the data relative to the remaining data objects, which will be of importance later when a redesign process is required. For example, suppose that an AU-Agent implements the following two expressions as the performance measure (total mass M and safety factor S): $M = \rho(htL + w^2\sqrt{x^2 + y^2})$; $S = \sigma th^2/12P(L - x)$. Note that mass is linear in both t and h, while the safety factor for the bending member is linear in t but quadratic in h. Thus, as long as no other design objects reach the maximum acceptable dimension, it will always be preferable to increase h rather than t. When a set-valued assignment is made, the design object can also be either existentially-quantified or universally-quantified.

Table 1. Attributes of the data object

Attribute	Description
Id	a unique identification name
Handler	agent creating the data object
Type	(“decision”|“performance”|“specification” |“intermediate”)
Value	of type DataObjectValue
Priority	(“-1”|“0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”)
Degree of influence	(“none” | “linear” | “quadratic” | “cubic” | “quartic” | “more”)
Logic quantifier	(“none” | “universal” | “existential”)
Preference function (or Preference graph)	(“none”|“monotonic_decreasing” |“monotonic_increasing”|“triangular” |“reverse_triangular”|“trapezoid” |“reverse_trapezoid”|“monotonic_increasing_flat” |“flat_monotonic_decreasing”|“flat”)
Unit	determined by the agent based upon the types it depends upon

4.3 Vision to integrated product design

4.3.1 Agent interaction spaces for hierarchical design integration

In order to integrate hierarchically distributed design problems, DIM-Agents of the parent problem define network-accessible interaction spaces into which the design solutions from child problems are assigned. Then, the interaction space is represented by a finite state automaton that consists of six states and four types of integration functions, as shown in Fig. 6:

- The unbound state. This means that the space does not have any assigned value.
- The Single_Input_Offered state. This means that the space has a single agent that provides a design object.
- The bound state. This means that there is both a provider agent that assigns a design object to the space, and one or more consumer agents that need the assigned data object for their own design tasks.
- The Conflicted_Input_Offered state. This means that there are two or more provider agents that assign the same types of data objects to the space. That is, the data object is possibly in conflict.
- The Multiple_Input_Offered state. This means that there are two or more provider agents that assign different types of data objects to the space. This implies that the space is intended to create a new design object, based on the individual design objects.
- The Unspecified_Mapping_Type state. This is to allow input actions from users to specify the mapping operation type.

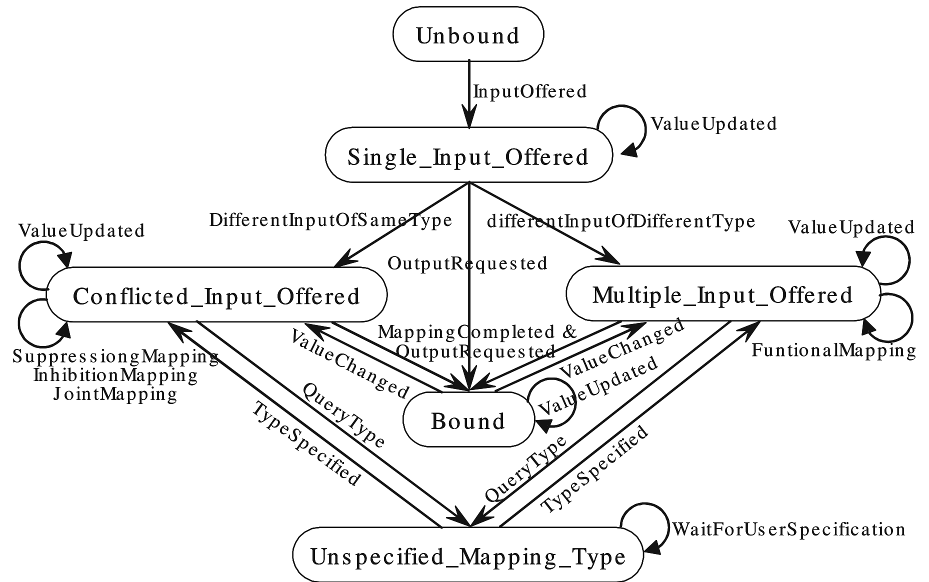


Fig. 6. An example of interaction space for hierarchical design integration

In addition, four different types of data mapping operations are proposed to integrate design objects:

- The SuppressionMapping operation. The old data is suppressed by the new one. This operation always releases the newest data object.
- The InhibitionMapping operation. This mapping has a single priority value that is used for arbitration. When more than one data object is connected to this mapping operation, the value of the domain with the highest priority value will be set to the value of this mapping. For example, the solution to the "design an automotive suspension" problem may be the solution to the problem "design a hydraulic suspension". The designer may eliminate the mechanical suspension option with the lower priority from consideration (at least temporarily).
- The JointMapping operation. This operation represents the logical relationships between the individual design objects of the child design problems. For example, to make the robust design decision against future product variations, a body design team may select a body compatible with any of the possible power trains mapped from the power train design team (i.e. a logical AND operation).
- The FunctionalMapping operation. Unlike JointMapping, this operation represents the arithmetic relationships between the design objects. For example, the cost of the automotive suspension will be the summation of the costs of the valve, sensor, harness, etc., and the total cost of the automobile will further be the summation of the costs of the suspension, body, engine, chassis, brake system, automotive seat, and the like. A lot of built-in functions are implemented, including min, max, a set of average operators, etc.

4.3.2 Service usage scenarios for heterarchical design integration

In the ANetCoDE, functional agents like AU-Agents can be used to represent various engineering models related to products or processes and encapsulate the pre-existing software tools, by defining input and output actions transparent to the remote users. These agents are also network-accessible and pluggable. Then, the agents can be directly connected to each other through proxy agents (a special type of HIA that contains only shared input and output actions) in a networked environment like the Internet.

Since a design team can predict the design outcomes from its child sub-problems, the shared interaction spaces for hierarchical design integration can usually be defined in advance. On the other hand, the dependencies between the heterarchically distributed design problems seem to occur more dynamically. As a mechanism for the heterarchical design integration, we present here the idea of publishing and subscribing the AU-Agents (see Fig. 7). When an AU-Agent is published to the neighbouring domains via the DIM-Agent of the parent problem domain, three usage options of the published AU-Agent are available, including Service_Supplier, Service_Consumer and Service_Simulator.

- The Service_Supplier option. If an AU-Agent is published as a service supplier, the designer can take its output services for his or her local usage through the agent proxy. The out-

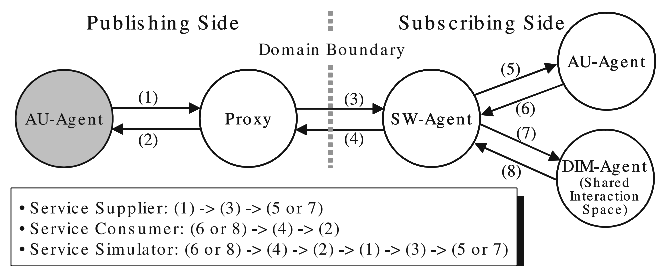


Fig. 7. Publishing and subscribing mechanism for heterarchical design integration

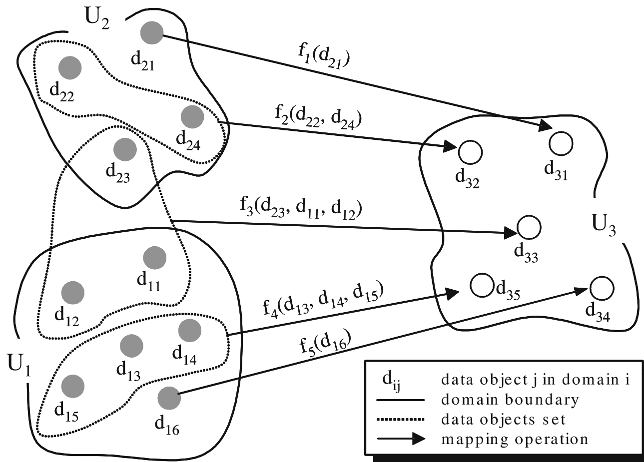


Fig. 8. Transformation processes for interoperability between incompatible design objects

put design objects are passed onto the local AU-Agent or DIM-Agent through the mediation of an SW-Agent. When introducing more than one agent to the local domain, a designer can also make the remote design objects compatible with his or her local domain after some transformation processes. Fig. 8 shows a pictorial representation of such mapping operations. U_1, U_2 and U_3 are the "universal set" of all the assignments to attributes in domain 1, 2 and 3, respectively. Then, the designer can create a new design object by assembling or customising the remote design objects accessible over the Internet.

- The Service_Consumer option. Unlike the Service_Supplier option, this option implies that the publishing side requires the input to its AU-Agents from the remote domains. By the Service_Supplier and Service_Consumer options, the designer tries to reveal the local design decisions to remote domains by exchanging local design objects, and to examine whether his or her current design solutions are acceptable or feasible to the remote domains or how the solutions influence the remote design solutions. When some conflicts are caused by the different viewpoints of each domain, a finite sequence of a resolution process to negotiate or trade off the conflicts is made. That is, these options are intended to incorporate the synchronous collaborative activities.
- Service_Simulator usage. Publishing AU-Agents as Service_Simulator means that the subscribing side can use the AU-Agents as a means to share their models of computation only for local purposes. By using this option, the designer can pre-estimate the current design performance from the viewpoint of different disciplines or exploit how the current design should be modified to reach an agreement of all members. By doing so, the designer can minimise the discrepancies between the different domains in advance, and finally can get a globally optimal solution with dramatically reduced backtracking processes. Therefore, the input to this AU-Agent by the subscribing side has no effect on the design of the publishing side.

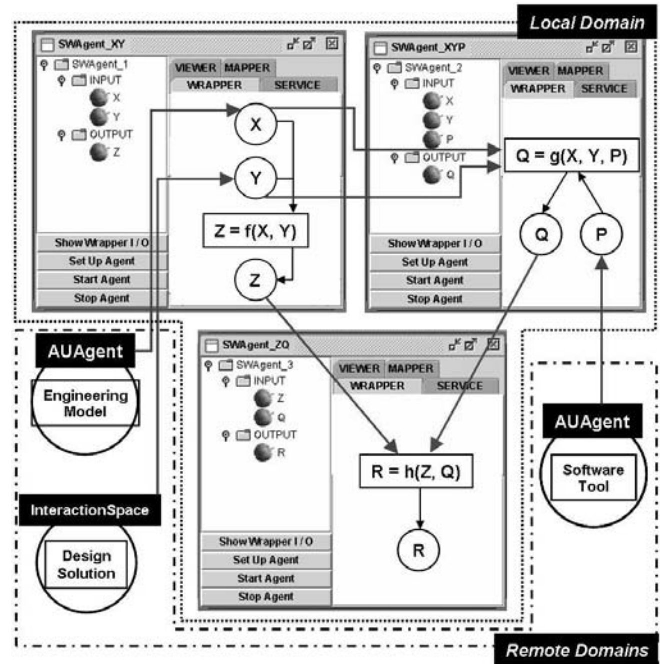


Fig. 9. An illustrative example of design integration

Finally, designers can make either input or output interface, or both interfaces of the AU-Agent accessible from the remote domains.

4.3.3 Plug-and-play for concurrent global design network

Figure 9 illustrates how the services from remote AU-Agents and interaction spaces are integrated. Design integration can be made by defining how the design objects from remote services are related to one another. When introducing the remote AU-Agents or interaction spaces to the local domain, the designer first creates SW-Agents by defining the interfaces to access remote services (i.e. input actions), making the services compatible for his or her purposes through a set of transformation functions (i.e. internal actions), and defining the destinations of the transformed remote services (i.e. output actions). After plugging the remote services into the local domain, SW-Agents can play the remote services.

Suppose that two AU-Agents and an interaction space are published by the remote domains and are now accessible over the Internet. Then, the designer can define a lot of transformation functions (i.e. embedded models) by instantiating SW-Agents, based on the data objects obtained from published AU-Agents and interaction space services. In this example, the three SW-Agents (SWAgent_XY, SWAgent_XYP and SWAgent_ZQ) are created to define three embedded models including $Z = f(X, Y)$, $Q = g(X, Y, P)$ and $R = h(Z, Q)$. For example, SWAgent_XY takes the two input data objects, X and Y, from the AU-Agent and interaction space, respectively, and generates a new data object, Z, by using a transformation function, $Z = f(X, Y)$.

These plug-and-play processes finally form an extended design chain of design agents that emerges from the system in a

dynamic manner. Then, the resultant concurrent design network represents an integrated global perspective for the given design project.

5 Implementation

A prototype version of the proposed framework has been implemented in Java2SE. In the previous version, the common object request broker architecture (CORBA) standard for distributed computing environments is utilised to add distributed communications capabilities to design agents. The software module has been recently replaced by the Java remote method invocation (RMI), due to its simple interaction protocol, the automatic and transparent serialisation for exchanged data and objects, the standard infrastructure for the distributed Java programming, and so on. The Java native interface (JNI) programming was also used to integrate pre-existing software programs that were written in the different types of languages such as C and C++.

Figure 10 illustrates the software modules for constructing the Internet-based integrated product design environment. The system components in grey represent those which have been prototyped in our laboratory as part of architecture. The core component is the ANetCoDE kernel including agent architecture and network architecture. All the design agents are implemented as the concurrent threads, by extending java.lang.Thread or by implementing java.lang.Runnable. The agents then exhibit a combination of continuous and discrete behaviour at their regular intervals. These intervals can also be specified by the designer. In addition, the DIM-Agent and AU-Agent have the adapter objects (i.e. a kind of proxy) implementing the RMI-compliant interfaces, to which their clients (SW-Agents) can connect for use. The SW-Agents get the appropriate proxy objects by the dynamic class loading through an HTTP server. In the current implementation, the proxy objects use RMI's default socket-based transport. As required, they can therefore be interconnected to form an integrated design network through the Internet.

The AU-Agent is used for local or remote use, and for a third party application. An application programming interface (API) is provided so that third party software wrappers plug into the ANetCoDE system. The AU-Agent transforms pre-existing domain-specific models or applications to ANetCoDE-compliant entities and provides the interoperability between the entities. The DIM-Agent can provide a number of shared interaction spaces to which the SW-Agents with permission to access can connect. The SW-Agents provide a mapping between the remote design objects and local design objects.

6 Conclusions

This paper develops a multi-agent system framework, ANetCoDE, for an integrated product design environment that enables systematic and timely integrations in the hierarchical and heterarchical design topology. To build the ANetCoDE, we first propose a lightweight agent network architecture, called the autonomy-based federation approach, to provide flexibility for agent consideration, plug-and-play and dynamic agent interaction in a networked environment. Second, a hybrid agent architecture is proposed for a more complete model of real-world system components, engineering models or tools. It can model both hybrid (continuous and discrete) behaviour and hybrid interaction, and dynamically adapt its behaviour and interaction modes to the environmental random changes. Finally, some ideas for enabling the hierarchical and heterarchical design integrations are presented, including the network-accessible agent interaction spaces and pluggable AU-Agents.

References

1. Prasad B (1996) Concurrent engineering fundamentals, Volume I: integrated product and process organization, and Volume II: integrated product development. Prentice-Hall, New York

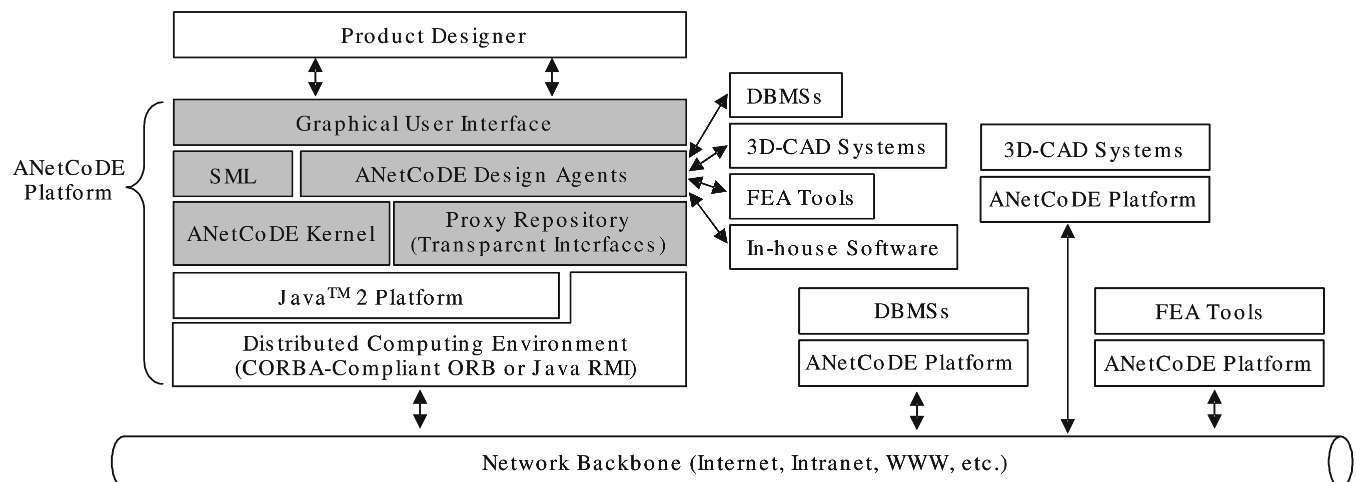


Fig. 10. System components of ANetCoDE

2. Willaert SSA, De Graaf R, Minderhoud S, Collaborative engineering: A case study of concurrent engineering in a wider context. J Eng Technol Manage 15:87–109
3. Kusiak A, Wang J (1995) Decomposition of the design process. Trans ASME J Mech Des 115:687–695
4. Kusiak A, Wang J (1994) Negotiation in engineering design. Group Decis Negotiation 3:69–91
5. Kusiak A, Larson N (1995) Decomposition and representation methods in mechanical design. Trans ASME J Mech Des 117:17–24
6. Cutkosky MR, Tenenbaum JM, Glicksman J (1996) Madefast: an exercise in collaborative engineering over the Internet. Commun ACM 39(9):78–89
7. Jennings NR, Sycara K, Wooldridge M (1998) A roadmap of agent research and development. Auton Agents Multi-Agent Syst 1:7–38
8. Wooldridge M, Jennings NR(1995) Intelligent agents: theory and practice. Knowl Eng Rev 10(2):115–152
9. Shen W, Norrie DH (1999) Agent-based systems for intelligent manufacturing: a state-of-the-art survey. Knowl Inf Syst 1(2):129–156
10. Peng Y et al. (1998) An agent-based approach for manufacturing integration – the CIIMPLEX experience. J Appl Artif Intell 13(1/2):39–63
11. Peng Y et al. (1998) A multi-agent system for enterprise integration. Int J Agile Manuf 1(2):201–212
12. Kuokka DR et al. SHADE: knowledge-based technology for the reengineering problem. http://www-ksl.stanford.edu/knowledge-sharing/
13. Cutkosky MR et al. (1993) PACT: an experiment in integrating concurrent engineering systems. IEEE Comput 26(1):28–37
14. Finin T et al. (1992) Specification of the KQML agent-communication language. Tech Report EIT TR 92-01, Enterprise Integration Technologies, Palo Alto, California
15. Genesereth MR et al. (1992) Knowledge interchange format, version 3.0 reference manual. Tech Report Logic-92-1, Computer Science Dept, Stanford University, Palo Alto, California
16. Toye G et al. (1993) SHARE: a methodology and environment for collaborative product development. In: Proceedings of the Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp 33–47
17. Shen W et al. (1997) Agent-based approaches for advanced CAD/CAM systems. In: Proceedings of the Fifth International Conference on CAD/Graphics, Shenzhen, China, pp 609–615
18. Maturana F et al. (1997) Multi-agent architectures for concurrent design and manufacturing. In: Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing, Banff, Canada, 27 July to 1 August, pp 355–359
19. Chen Y et al. (1999) A negotiation-based multi-agent system for supply chain management. In: Proceedings of Autonomous Agents, Workshop on Agent-Based Decision-Support for Managing the Internet-Enabled Supply-Chain, Seattle, Washington
20. Zha XF, Lim SYE, Fok SC (1999) Development of expert system for concurrent product design and planning for assembly. Int J Adv Manuf Tech 15:153–162
21. Zha XF (2002) A knowledge intensive multi-agent framework for cooperative/collaborative design modelling and decision support of assemblies. Knowledge-Based Sys 15:493–506
22. Zhao FL, Tso SK, Wu PSY (2000) A cooperative agent modelling approach for process planning. Comput Ind 41:83–97
23. Maturana F, Balasubramanian S, Norrie DH (1996) A multi-agent approach to integrated planning and scheduling for concurrent engineering. In: Proceedings of the Third ISPE International Conference on Concurrent Engineering, Toronto, Canada, 26–28 August 26-28, pp 272–279
24. Flores RA, Kremer RC, Norrie DH (2000) An architecture for modelling Internet-based collaborative agent systems. In: Proceedings of Agents' 2000 Workshop on Infrastructure for Scalable Agent Systems, Barcelona, Spain
25. Kim KS et al. (2000) Compensatory negotiation for agent-based project schedule coordination. CIFE Working Paper #55, Centre for Integrated Facility Engineering, Stanford University
26. Danesh MR, Jin Y (2001) An agent-based decision network for concurrent engineering design. Concurrent Eng: Res Appl 9(1):37–47
27. Petrie CJ, Webster TA, Cutkosky MR (1995) Using Pareto optimality to coordinate distributed agents. Artif Intell Eng Des Anal Manuf 9(4):313–323
28. Pena-Mora F et al. (2000) CAIRO: a concurrent engineering meeting environment for virtual design teams. Artif Intell Eng 14: 203–219
29. Berardi D et al. (2003) Finite state automata as conceptual model for E-services. In: Proceedings of the 7th Conference on Integrated Design and Process Technology, Special Session on Modelling and Developing Process-Centric Virtual Enterprises with Web-Services
30. Ou-Yang C, Lin JS (1998) The development of a hybrid hierarchical/heterarchical shop floor control system applying bidding method in job dispatching. Robot Comput-Integr Manuf 14:199–217
31. Pahng G-D, Bae S-H, Wallace D (1998) Web-based collaborative design modeling and decision support. In: Proceedings of the 1998 ASME Design Engineering Technical Conferences, Atlanta, Georgia, 13–16 September
32. Rosenman M, Wang F (2001) A component agent based open CAD system for collaborative design. Automat Constr 10:383–397
33. Sriram D, Logcher R (1993) The MIT dice project. IEEE Comput 26(1):64–65
34. Rosenman M, Wang F (1999) CADOM: a component agent-based design-oriented model for collaborative design. Res Eng Des 11:193–205
35. Parunak HVD, Ward A, Fleischer M, Sauter A (1997) A marketplace of design agents for distributed concurrent set-based design. In: Proceedings of the 4th ISPE International Conference on Concurrent Engineering: Research and Applications
36. Parsons MG, Singer DJ, Sauter JA (1999) A hybrid agent approach for set-based conceptual ship design. In: Proceedings of the 10th International Conference on Computer Applications in Shipbuilding
37. Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235
38. Lynch NA, Tuttle MR (1987) Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, pp 137–151
39. Lynch N, Segala R, Vaandrager F (2003) Hybrid I/O automata. Inf Comput 185(1):105–157
40. de Alfaro L, Henzinger TA (2001) Interface automata. In: Proceedings of the 9th Annual ACM Symposium on Foundations of Software Engineering
41. Noda I (2002) Hidden Markov modelling of multi-agent systems and its learning method. In: Proceedings of RoboCup 2002 International Symposium
42. Chang W-T, Ha S-H, Lee EA (1997) Heterogeneous simulation – mixing discrete-event models with dataflow. J VLSI Signal Process 15:127–144
43. Liu J et al. (2001) Interoperation of heterogeneous CAD tools in Ptolemy II. J Model Simul Microsyst 2(1):1–10
44. Wooldridge M (1992) The logical modelling of computational multiagent systems. PhD Thesis, Department of Computation, The University of Manchester
45. Oka T, Tashiro J, Takase K (1999) Object-oriented BeNet programming for data-focused bottom-up design of autonomous agents. Robot Auton Syst 28:127–139
46. Budau V, Bernard G (2002) Synchronous/asynchronous switch for a dynamic choice of communication model in distributed systems. In: Proceedings of the 9th International Conference on Parallel and Distributed Systems, 17–20 December