The Missing Layer — Virtualizing Smart Spaces

Marc-Oliver Pahl and Georg Carle
Chair for Network Architectures and Services
Technische Universität München
Garching bei München, Germany
{*pahl, carle*}*@net.in.tum.de*

***Abstract*—With the Virtual State Layer (VSL), an abstraction for software-based orchestration of smart spaces is presented. The aim of the VSL is to ease the programming of orchestration software while not limiting its functionality.**

The VSL primarily provides: the virtualization of that part of the real world that can be orchestrated via sensors and actuators, a base for the creation of portable orchestration services, mechanisms to easily couple services, and the availability of state over time.

The VSL approach is conceptually situated between ontology-based systems and protocol-translating systems for orchestrating smart spaces.

The concept and its implementation — a Java-based distributed P2P publish-subscribe system — are presented.

***Keywords*-autonomous computing; distributed computing; smart space; app store**

I. Introduction

Orchestration of spaces by humans can be decomposed into perception ("it is dark"), cognition ("I would feel more comfortable if I could see something; there is a light, and it seems to be off"), and action ("turn the light on") according to models of cognitive psychology [1].

For automated space-orchestration, this process can be mapped into software as a series of event-condition-action (ECA) rules. The orchestration control flow described above appears in natural language as "when it is dark outside and someone enters $roomA$, switch the lights on":

$$\left\langle \begin{array}{c} \text{event} \\ \text{condition} \\ \text{action} \end{array} \right\rangle : \left\langle \begin{array}{c} \text{motionDetector}_{roomA} = \top \\ \text{lightSensor}_{outside} < 5000\text{Lux} \\ \text{lights}_{roomA} := \top \end{array} \right\rangle \quad (1)$$

Rule-based orchestration workflows are often the right level of abstraction for smart space automation. It is straightforward to realise them in software and to adapt them by reconfiguring the rule set to changed needs.

Sometimes more complex processing, such as machine learning, is needed to accomplish orchestration goals. Programming languages and execution hardware allow a high level of complexity to be expressed in software. Software is not the limiting factor in smart space orchestration.

Automated space orchestration is limited by hardware. The sensors and actuators a space contains define which physical state can be measured or changed, and thereby which part of the real world can be orchestrated by software. Connecting hardware and software is a fundamental challenge for smart space orchestration.

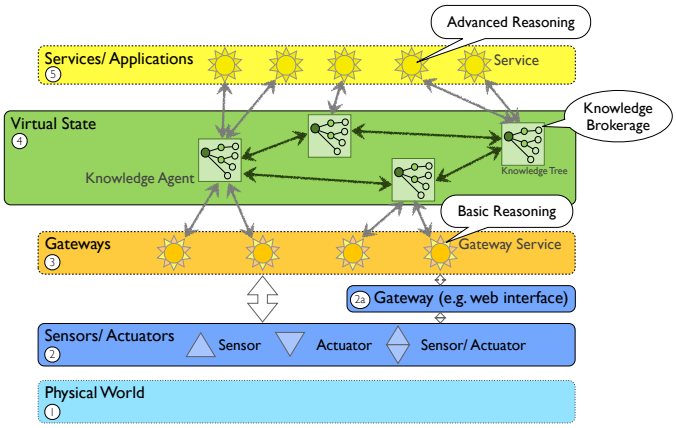

Figure 1. Logical layering of smart space orchestration.

One goal of this paper is to present an abstraction for sensors and actuators in the software domain. It should fit the needs of orchestration software and not introduce limitations. Additionally it should enable site-independent portable orchestration services. The Virtual State Layer (VSL) presented here does so by virtualizing real world states, sensors and actuators interact with. See Fig. 1.

Orchestration (done by machines or humans) can be expressed as an Input-Processing-Output (IPO) model:

1) Input: Obtain relevant state.
2) Processing: Reason on the obtained state.
3) Output: Possibly change state.

Obtaining and changing state are common operations when orchestrating spaces. The presented VSL aims to alleviate the programming of services by offering support for common tasks. The virtualization of physical state takes the IPO tasks 1 and 3 out of the services' domain (layers 1-3 in Fig. 1). Without need for writing code to obtain sensor state or control actuator state, programmers can focus on the orchestration logic. The availability of virtualized real-world state simplifies smart space services significantly.

Allowing software to be the origin of states makes it possible to structure orchestration functions. Complex services can be decomposed into better-manageable (reusable) small services, which provide context for one another. Using the VSL, software modules can be coupled and reused at need, leading to a service-oriented architecture (SOA).

978-1-4673-5077-8/13/$31.00 ©2013 IEEE

The availability of past state makes it possible to express changes over time. Using the VSL, this becomes possible even for simple ECA-based software, as all state present in the abstraction can be used in rules.

In the following a middleware concept is presented that uses state exchange as a communication principle between smart space entities. The concept presented leads to spatial and temporal decoupling of services and sensors/actuators. It allows the full modularisation of functionality within the software part of smart spaces.

II. Concept Virtual State Layer

The new layer introduced in this work is called the Virtual State Layer (VSL). Its purpose is to store and provide the state used by smart space orchestration services.

The VSL is the only concept for inter-process communication in the control plane of a smart space. Non-control traffic can be exchanged directly over the data plane; for instance, software making music follow the inhabitant of a house will not usually use the VSL for audio traffic.

The VSL is an extension of the tuple space interprocess communication concept [2]. A tuple space is a shared data structure among processes. Producers can create typed tuples and put them into the tuple space. Consumers can search for tuples and get notified when tuples become available that match their search criteria. Consumed tuples are removed from the tuple space.

The VSL adapts the tuple space concept to the distributed-computing-environment smart space. The most significant extensions to [2] are the introduction of hierarchical unique addresses and permanent distributed storage of tuples.

The tuples inside the VSL will be called (information) nodes in the following. They represent state information of a smart space. The state can represent physical state of the orchestrated space that gets measured by hardware or inferred information created by software.

VSL nodes have unique addresses. The addresses are hierarchically structured, generating a logical tree of nodes. The hierarchy expresses the relationships between information. Different information nodes can form a device, for instance, with the device ID as a parent node and its components as children. The same applies for service representations: a device may contain multiple sensors, a room may contain different devices, or a gateway may interface with different devices. Services can transparently access nodes by their addresses. Each process gets an address space that it can freely organize.

Nodes inside the VSL can have more than one data type. Types can be used by services to search for certain information, e.g. temperature sensors within the smart environment.

The hierarchical addresses and the use of multiple data types provide extended semantics to the original tuple space concept.

VSL nodes have additional attributes to reflect the context of smart space information. Version numbers and time stamps are especially useful in the context of permanently stored information and asynchronous coupling where when information was produced is of interest. For instance, it might be relevant for a heating service that the temperature in the living room was measured yesterday night.

Access rights reflect the security and privacy needs of smart spaces. For example, the VSL as basic abstraction for smart space orchestration must provide means to prevent the *blinkenlights* service[1] from opening the front door. At the same time, the values of motion sensors inside the bedroom should not be accessible to the facility manager service that automatically reports broken lights to an external company. As with file systems, read and write access rights can be set for identities (e.g. user or group IDs). Having an access control scheme as part of the VSL as elementary information manager allows security and privacy by design.

As distributed version of a tuple space, the VSL contains a publish/subscribe system [3]. Nodes can be subscribed by *readerIDs*. As soon as a node changes its value, the *readerID* gets notified by the VSL. The hierarchical addressing allows subscription of parent nodes that lead to notification of updates in any readable child.

The introduction of the complex data types that goes along with hierarchical addressing makes it necessary to allow transactions on subtrees. Changing the IP address of a network interface card without changing the broadcast address would make no sense, for instance. Transactions are realized via locks on subtrees. Every ID that has read or write permission on a node can lock its subtree.

The permanent storage of tuples allows full spatial and temporal decoupling of processes. It makes the history of a space remain accessible.

Tuple spaces generally allow synchronous communication via publish/subscribe: a consumer can subscribe to an address and read out new tuples as soon as it is notified.

With the persistence of the VSL, this method of tight coupling leads to unnecessary latency. The VSL has to store a published value; then, it sends a notification to the receiver. Finally, the receiver retrieves the value. Even if the storage took no time, the round-trip time for the information to be passed is doubled (notification + retrieval). This latency is unnecessary and might be critical for applications requiring low latency.

To provide fast synchronous interaction between entities, the concept of virtual nodes is introduced. A virtual node is an address that does not point to a node inside the VSL but instead to a service acting as a consumer (*set*) or as a producer (*get*) of information. The equivalent construct to a virtual node in an operating system is a pipe. Access to a virtual node is transparent for services.

[1] http://blinkenlights.net/

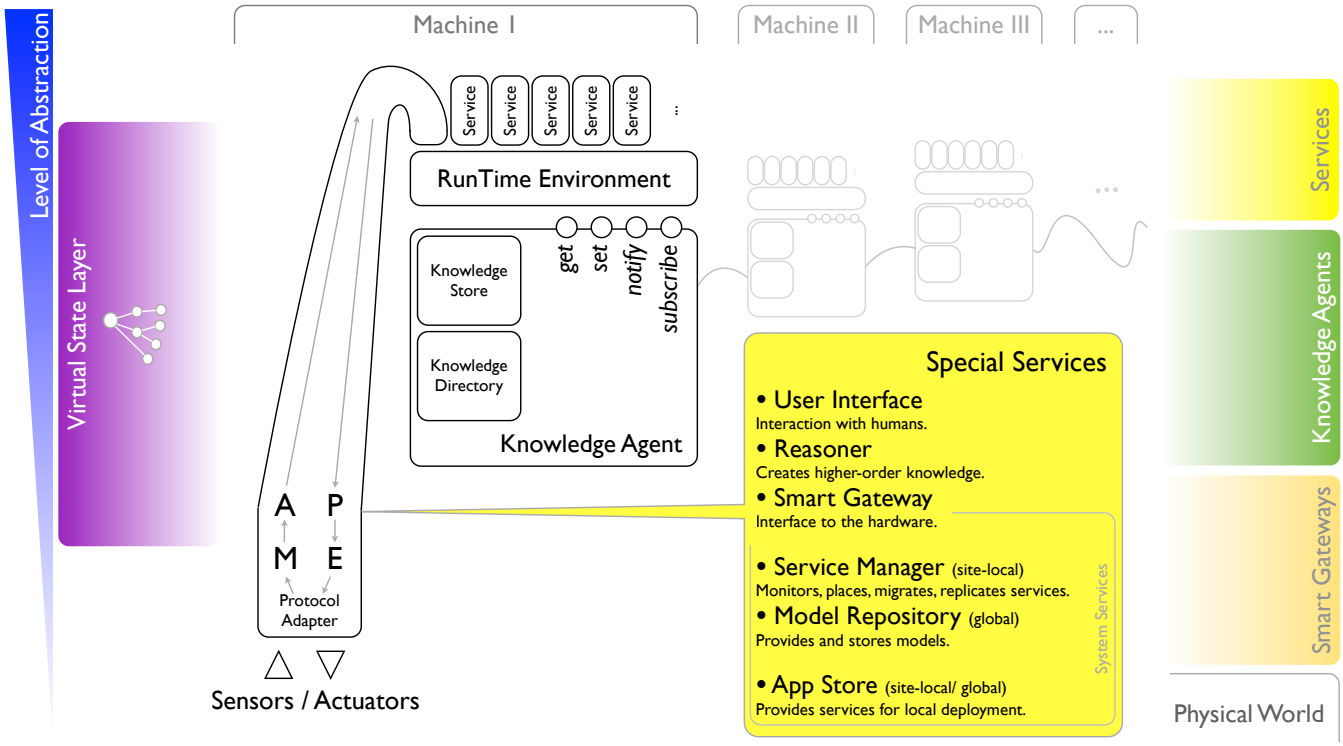

Figure 2. Architecture of the Distributed Smart Space Orchestration System with the VSL in the centre.

III. Implementation

The VSL is the foundation of the Distributed Smart Space Orchestration System (DS2OS). See Fig. 2.

Smart space hardware is heterogeneous. The VSL is implemented in Java to make it runnable on various hardware platforms. Services using the VSL can be implemented in other programming languages. Currently all services mentioned in Fig. 2 are implemented in Java, but the user interface service. It is running in PHP (see III-C4).

A. Peer-to-Peer Overlay

The VSL is spanned by a Peer-to-Peer (P2P) overlay of so-called *Knowledge Agents* (KA). One *Knowledge Agent* running inside a smart space makes the system work, as all services can use it as node-exchange point.

To make use of the available resources, the VSL is meant to run on all nodes of a space that are powerful enough. In combination with node replication, this distribution provides enhanced resilience to using only one powerful machine. Distributed access nodes (KAs) lower the latency of VSL accesses to information that is on close agents. Caching can be used to enhance the performance for accessing remote information. With caching enabled, the VSL realises a kind of a content-centric network [4], autonomously retrieving and enhancing access to information nodes based on their hierarchical addresses.

When slow links partition areas of devices with fast links, the VSL can partly cover the negative effects (e.g. latency, bandwidth), as a KA in the fast domain acts as proxy for the slow-link devices.

Finally, distribution makes the VSL scale.

The P2P overlay is agnostic to underlying communication protocols. In the current implementation, three transports are provided, and more transports can be easily added. The first transport is OSGI remote service invocation [5]. It is used by services on a local machine to access their KA that runs as registered OSGI service. Services running as OSGI bundles discover the agent and use the OSGI framework for fast communication on the local machine. When OSGI is not available, the knowledge agent can run as plain Java program. The second transport is TCP. It can be used by local services, remote services, and remote agents. The third transport is XMPP. It is primarily used as communication protocol between agents.

Interagent communication is always encrypted, as the data of the VSL has to be protected. Requests for information nodes are always authenticated. As agents send requests on behalf of services, joining the overlay is a security-critical operation. A malicious knowledge agent may, at a minimum, eavesdrop on the traffic of its directly connected services and act on behalf of them.

For discovering other knowledge agents, IP multicast in IPv6 subnets and IP broadcast in IPv4 networks are

used. When another agent is found, the agents authenticate mutually using signed certificates containing each agent's public key.

Each site has a local certificate authority (CA). In an initial branding process it creates a unique public/private key pair for each agent and issues a certificate with the agent's public key. Each agent holds its private key, its signed certificate, and the public key of the site's CA to verify certificates of other agents. The initial branding helps assure the integrity of the VSL, as only authenticated agents can join the overlay.

To authenticate, a KA sends its certificate and the remote agent verifies it. The verified public key of the new agent is used to establish a session key and to exchange the current shared encryption key of the overlay. The session key is unique for the new agent. It is used in the periodic rekeying message to encrypt the new shared overlay communication key. Using the indirection over a KA's session key allows the exclusion of a KA from the overlay, as it can only decrypt the new shared key when it is encrypted with its session key. After a successful join, agents communicate using the overlay's shared communication key.

For bootstrapping, the agent that is already part of the overlay sends the mappings between agentIDs and transport addresses to the new agent. The KA will use this information to retrieve information from remote nodes on request of a connected service. Additionally, it sends the address space of the whole VSL containing all types (see *Knowledge Register* in Fig. 2). This *Knowledge Register* is duplicated on all *Knowledge Agents*, as it serves as entry point to the VSL. Having a local directory allows fast search on types within the VSL.

The VSL is updated periodically via broadcast within the overlay. When the structure stored on an agent changes, it broadcasts the updates in a periodic update message. If a foreign agent misses an update, it will note this on the next update it receives based on a mismatch of the expected hash. It queries the agent where the *Knowledge Register* hash mismatches for an explicit update then. Using the broadcast mechanism minimises the network traffic from n^2 (bidirectional exchange between all agents) towards n. In practise, the quantity of messages is even lower, as not all agents have structure updates every round.

In the current implementation, each *Knowledge Agent* stores the information of services directly connected to it. As each information node is available only once, the VSL stays consistent. The node information is stored in the *Knowledge Store* (see Fig. 2).

B. Interface to Services

The operation of the VSL is transparent for service programmers. The interface to the VSL offers the methods *search*, *get*/*set*, *lock*/*unlock*, *subscribe*/*unsubscribe*, *addSubtree*/*removeSubtree*, and *registerClient*/*unregisterClient*.

In the current implementation, the *search* command allows searching for nodes of a given type. It is used to discover devices of a certain type, such as "/light/lamp", for instance. With the hierarchical addressing and multi-type nodes, the type is used to identify classes of devices (see III-C1). A device of type "/light/lamp" may contain a subnode "isOn" for instance. In this way, abstract devices are realised. Knowing that the state of each lamp can be queried using the address that is returned by the search and adding "/isOn" in DS2OS enables developers to write orchestration services that are independent of concrete devices. This overcomes a fundamental problem in smart space orchestration today: services can usually run on only one site, as they are built for a specific device context.

A search returns a list of VSL addresses that have the searched type and are accessible by the ID running the search. Addresses of non-accessible information nodes are not returned, as they cannot be accessed and as the structure of the space should not be revealed.

The *get* method is called with an *address* and a *readerID*. Services do not have to care about where information is stored. The KA retrieves it transparently. The *set* method sets the value of a node if the given ID has write permission.

The *lock* method locks the subtree originating at the given address if the *lockerID* write or read permission. After a specified time, the locker is notified that the lock will expire and can renew it or the lock will be unlocked.

Subscribe and *unsubscribe* work as described in II and require read rights on a node.

AddSubtree and *removeSubtree* are used to insert and remove new information nodes into the VSL. They require write permission on the parent node of the new subtree. *AddSubtree* is usually called using models of the *Model Repository* (see III-C1).

RegisterClient and *unregisterClient* are used to connect services to the VSL. When a service is authenticated a service node is created in the VSL if it is not present already. The service can store its state under this node.

C. Other Services of DS2OS

The VSL is the core of the Distributed Smart Space Orchestration System. It is designed to provide the extended tuple space. DS2OS contains additional functionality on top of the VSL to facilitate the development of applications.

Some additional concepts will be briefly introduced, as they make the use and the design of the VSL more clear. In particular, the *Smart Gateways* are relevant, as they reflect the part of the real world that can be measured by sensors and changed by actuators on the VSL and vice-versa.

1) Model Repository: The *Model Repository* is a global directory of definitions of complex objects in the VSL.

VSL nodes are hierarchically ordered in a tree address space (see II). This allows creating complex data types with multiple fields (subnodes). VSL nodes can

have multiple types. This allows inheritance. A type "/lights/dimmableLamp" can be derived from a type "/lights/lamp", for instance. Derived types have both their own type and their parent type. This enables services for controlling lamps to discover and control dimmable lamps when searching for "/lights/lamp".

Complex and derived types can be stored as templates (*Models*) in the so-called *Model Repository*. *Models* can contain not only structures, but also values (e.g. device-specific default values). *Models* are the abstraction of complex entities such as devices or services in the VSL. Services instantiate *Models* at runtime, leading to the creation of a new subtree inside the local VSL.

The *Model Repository* is global. This makes *Models* consistent in all spaces. Services are usually programmed against *Models*. *Models* are the base for portable services that can run site-comprehensively using the VSL *Models* as the interface to a smart space instance. As the *Model Repository* is global, service developers can look up the *Models* they want to use and write code conforming to them.

2) Runtime Environment: DS2OS contains a *Runtime Environment* (RTE) that manages services locally on a node. The RTE uses OSGI [5] to install, start, pause, and stop services on a node. It monitors services and sends information about the load on a device to the *Service Manager*.

3) Service Manager: In a DS2OS space, there is usually one *Service Manager* (SM) instance running. It places and migrates services autonomously. To do so, the SM collects resource information from the distributed RTEs.

The distribution of the VSL on all hosts and its strict decoupling of entities makes migration and replication of services possible in an easy way. The indirection over the types allows moving content to different addresses (for example, when the hostID changes as a node fails).

4) Services: The VSL and the *Model Repository* facilitate the creation of services. There are different kinds of services.

User Interface services offer interaction with users. They benefit from the type system, as they can provide generic inputs based on the basic data types of the *Model Repository*.

Control services orchestrate a space. With the support of the VSL, it becomes possible to realise complex workflows with simple logic (see I).

Reasoning Services take information out of the VSL, reason on it, and put new information back into the VSL. *Reasoning Services* are often used as external modules for services – especially *Smart Gateway* services. The use of the VSL as the only coupling mechanism makes it easy to couple any kind of information in a space, as it is accessed and represented in the same way.

Smart Gateway services couple sensors and actuators with the VSL. They reflect the part of the physical world that is interfaced by sensors and actuators to the VSL and vice-versa autonomously. A gateway is an ordinary service with the exception that some gateway services may need special physical interfaces (e.g. to LON or EIB). *Smart gateways* try to extract the physical reality out of the protocol messages exchanged with the devices. If they manage to do so, no loss is introduced via the VSL, as no additional limit is introduced to the existing interface to the real world.

In the *Models*, the autonomy of a *Smart Gateway* is reflected by distinguishing between desired and running states. The reflection of the desired state to reality and from there to the running state happens in the *Smart Gateway*. In order to be notified of value changes in the structure, the *Smart Gateway* uses a VSL subscription on the root of the desired subtree.

Smart Gateways are based on the Monitor-Analyse-Plan-Execute (MAPE, see Fig. 2) structuring for autonomy [6]. They can be configured according to the needs of the connected devices (see also [7]). The *Plan* module subscribes to the desired subtree. In case of a change (notification received), it tries to reflect the change to the physical world using the *Execute* module. On the way back, the *Protocol Adapter* sends information to the *Monitor* module. The *Analyser* processes the information and adds it to the VSL.

The presented version of the MAPE scheme fits for command line, webservice, REST, BACnet, and EIB. Using existing frameworks and libraries for the protocol adapter makes it easy to create a protocol adapter.

5) Application Store: The *Service Manager* (see III-C3) takes binaries of services and deploys them to local hosts. It is only a small step to connect the local *Service Manager* to the *Application Store*. The decoupling over the VSL makes site-comprehensive services possible.

IV. Related Work

There are many middleware approaches for smart spaces. Common to all approaches is the challenge of high-level instructions by humans on the one hand and concrete values on sensors and actuators on the other hand (see I).

The scientific community has developed ontologies as formalisms to map concrete events to increasingly abstract concepts [8], [9], [7]. The concept makes it easy to apply orchestration rules on abstract knowledge. The drawback of the ontological approach is that it is very complex to create ontologies and to apply them in a way that makes the mapping to and from concrete and abstract domains clear. Complex ontology-based approaches are seldom found in smart space environments outside research labs [10]

The opposite approach could be called "the babelfish approach". Communication messages of heterogeneous devices are translated at runtime (e.g. in order to match the format of the destination system). This allows synchronous communication without adding heavy abstraction. The translation happens autonomously, with gateways translating syntax and semantics [11], [12]. The approach of providing access to all services in a space without adding much abstraction is often found in customer installations [10].

The proposed VSL approach lies in the middle of both. It provides exactly the level of semantic abstraction that is needed for orchestration to work (see III-C4 *Smart Gateways*). The bidirectional mapping between the output of a device and its representation in the VSL remains.

The VSL is a good base for services reasoning with ontologies. They can use the VSL as interface to the world and provide their reasoning results to other services over it. At the same time, protocol translating gateways can be used very well as input or replacement of the Smart Gateways.

Microsoft started working on an operating system for the home in 2010 [13]. Their system follows the same goals our system does; however, the realisation is very different in that their system is centralised, while our system is decentralised. Microsoft's HomeOS is based on the idea of making remote devices appear as local to the machine the system is running. The abstraction of devices is function-based and synchronous. The VSL instead offers full spatial and temporal decoupling.

Microsoft's approach seems to be a good one for creating a business case for a company. The presented VSL and DS2OS approach will hopefully provide a good base for crowd-sourced development as it happens in the open source community.

V. Conclusion

The Virtual State Layer as a basic abstraction for orchestrating smart spaces was introduced. It extends the tuple space concept with hierarchical addresses, permanently-stored and versioned access-controlled tuples, and virtual nodes.

The representation of real world properties as state in the VSL is fit as sensor hardware samples the world and communicates the samples. Communication protocols as a means to communicate with sensors and actuators usually contain data that can be stored as VSL tuples. Therefore, inserting the VSL between orchestration software and hardware (see Fig. 1) does not usually limit the orchestration process.

Combined with the *Smart Gateways* (see III-C4), the VSL virtualizes the orchestratable part of a real space. Having to interact only with the VSL in order to orchestrate a space facilitates services significantly. As the VSL stores and brokers state tuples offered by any service, programmers do not have to worry about information storage and retrieval functionality. They can also use the VSL as a coupling mechanism between service to realise SOAs. The availability of the VSL allows mapping complex orchestration workflows to simple-to-implement concepts such as ECA rules (see III-C4). The combination of the VSL and the *Model Repository* (see III-C1) enables the creation of site-independent service.

Providing this base, we hope to see similar effects to what we observed in the app economy: crowd-sourced development of *Smart Gateways* (as special services) and orchestration software. With its vendor-comprehensive abstraction, the VSL could open up a sizeable playground and market for smart space orchestration.

References

[1] W. Prinz, "Perception and Action Planning," *European Journal of Cognitive Psychology*, vol. 9, no. 2, pp. 129–154, 1997.

[2] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, 1985.

[3] P. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 114–131.*, vol. 35, no. June 2003, pp. 114–131, 2003.

[4] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *the 5th international conference*. New York, New York, USA: ACM Press, 2009, p. 1.

[5] The OSGi Alliance, *OSGi Service Platform Core Specification*, ser. Release 4, Version 4.3. The OSGi Alliance.

[6] J. O. Kephart and D. M. C. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[7] N. Lasierra, A. Alesanco, J. Garcia, and D. O'Sullivan, "Data management in home scenarios using an autonomic ontology-based approach," in *2012 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2012, pp. 94–99.

[8] C. Bolchini, C. A. Curino, E. Quintarelli, and F. A. Schreiber, "A data-oriented survey of context models," *ACM SIGMOD*, 2007.

[9] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni, "A survey of context modelling and reasoning techniques," *Pervasive and Mobile Computing*, vol. 6, no. 2, pp. 161–180, Apr. 2010.

[10] A. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon, "Home automation in the wild: challenges and opportunities," in *Proceedings of the 2011 annual conference on Human factors in computing systems*. ACM, 2011, pp. 2115–2124.

[11] Y.-D. Bromberg, P. Grace, L. Reveillere, and G. S. Blair, "Bridging the interoperability gap: overcoming combined application and middleware heterogeneity," in *Middleware'11: Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag, Dec. 2011.

[12] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci, "The role of ontologies in emergent middleware: supporting interoperability in complex distributed systems," in *Middleware'11: Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag, Dec. 2011.

[13] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, "An operating system for the home," in *NSDI'12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, Apr. 2012.