An Architecture Description Language Based on Dynamic Description Logics

Zhuxiao Wang[1], Hui Peng[2], Jing Guo[3], Ying Zhang[1], Kehe Wu[1], Huan Xu[1], and Xiaofeng Wang[4]

[1] School of Control and Computer Engineering, State Key Laboratory of Alternate Electrical Power System with Renewable Energy Sources, North China Electric Power University, Beijing 102206, China
{wangzx,yingzhang,wkh,xuhuan}@ncepu.edu.cn
[2] Education Technology Center, Beijing International Studies University, Beijing 100024, China
penghui@bisu.edu.cn
[3] National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing 100029, China
guojing.research@gmail.com
[4] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
wangxiaofeng@ict.ac.cn

Abstract. ADML is an architectural description language based on Dynamic Description Logic for defining and simulating the behavior of system architecture. ADML is being developed as a new formal language and/or conceptual model for representing the architectures of concurrent and distributed systems, both hardware and software. ADML embraces dynamic change as a fundamental consideration, supports a broad class of adaptive changes at the architectural level, and offers a uniform way to represent and reason about both static and dynamic aspects of systems. Because the ADML is based on the Dynamic Description Logic DDL($\mathcal{SHON}$(D)), which can represent both dynamic semantics and static semantics under a unified logical framework, architectural ontology entailment for the ADML languages can be reduced to knowledge base satisfiability in DDL($\mathcal{SHON}$(D)), and dynamic description logic algorithms and implementations can be used to provide reasoning services for ADML. In this article, we present the syntax of ADML, explain its underlying semantics using the Dynamic Description Logic DDL($\mathcal{SHON}$(D)), and describe the core architecture description features of ADML.

Keywords: Architecture Description Languages, Knowledge Representation and Reasoning, Software Architecture, Dynamic Description Logics, Dynamic Adaptation.

1 Introduction

ADML is a promising Architecture Description Language (ADL) towards a full realization of the representing and reasoning about both static and dynamic aspects of

Z. Shi, D. Leake, and S. Vadera (Eds.): IIP 2012, IFIP AICT 385, pp. 157–166, 2012.

concurrent and distributed systems. Concurrent and distributed systems, both hardware and software, can be understood as a world that changes over time. Entities that act in the world (which can be anything from a monitor to some computer program) can affect how the world is perceived by themselves or other entities at some specific moment. At each point in time, the world is in one particular state that determines how the world is perceived by the entities acting therein. We need to consider some language (like the Architecture Dynamic Modeling Language, ADML) for describing the properties of the world in a state. By means of well-defined change operations named transition rules in ADML, transition rules can affect the world and modify its current state. Such transition rules denote state transitions in all possible states of the world.

In this paper we describe the main features of ADML, its rationale, and technical innovations. ADML is based on the idea of representing an architecture as a dynamic structure and supporting a broad class of adaptive changes at the architectural level. However, simultaneously changing components, connectors, and topology in a reliable manner requires distinctive mechanisms and architectural formalisms. Many architecture description languages[1-4] are dynamic to some limited degree but few embrace dynamic change as a fundamental consideration. ADML is being developed as a way of representing dynamic architectures by expressing the possible change operations in terms of the ADML constructors.

ADML can be viewed as syntactic variants of dynamic description logic. In particular, the formal semantics and reasoning in ADML use the DDL($\mathcal{SHON}$(D)) dynamic description logic, extensions of description logics (DLs) [5] with a dynamic dimension [6-9]. So the main reasoning problem in ADML can be reduced to knowledge base (KB) satisfiability in the DDL($\mathcal{SHON}$(D)). This is a significant result from both a theoretical and a practical perspective: it demonstrates that computing architectural ontology entailment in ADML has the same complexity as computing knowledge base satisfiability in DDL($\mathcal{SHON}$(D)), and that dynamic description logic algorithms and implementations can be used to provide reasoning services for ADML.

In the following sections, we firstly present an overview of the capabilities of ADML in Section 2. It covers the basic language features and includes a few small examples. Furthermore, we demonstrate the descriptions of transition rules can be formalized as actions in the DDL($\mathcal{SHON}$(D)). In Section 3, we summarize basic ADML syntax with an overview of ADML semantics. We show that the main reasoning problem in ADML can be reduced to knowledge base (KB) satisfiability in the DDL($\mathcal{SHON}$(D)) dynamic description logic. Finally, we summarize the paper in Section 4.

2 An Overview of ADML

ADML is intended as a new formal language and/or conceptual model for describing the architecture of a system. ADML is built on a core ontology of six types of entities for architectural representation: components, connectors, systems, ports, roles, and behaviors. These are illustrated in Figure 1 and Table 1. Of the six types, the most basic elements of architectural description are components, connectors, systems, and

behaviors. It's important to recognize that ADML is based on the idea of representing an architecture as a dynamic structure. In other words, ADML may also be used as a way of representing reconfigurable architectures by expressing the possible reconfigurations in terms of the ADML structures (like behaviors). For example, an architectural model might include behaviors that describe components that may be added at run-time and how to attach them to the current system.

2.1 ADML Design Element Types

As a simple illustrative example, Figure 1 shows the architectural model of a secure wireless remote-access infrastructure, which is represented as a graph of interacting components. Nodes in the graph are termed components, which represent the primary computational elements and data stores of the system. Typical examples of components include such things as terminals, gateways, filters, objects, blackboards, databases, and user interfaces. Arcs are termed connectors, and represent communication glue that captures the nature of an interaction between components. Examples of connectors include simple forms of interaction, such as data flow channel (e.g., a Pipe), a synchronous procedure call, and a particular protocol (like HTTP). Table 1 contains an ADML description of the architecture of Figure 1. In the software architecture illustrated in Table 1, the secure access gateway (SAG), and the wireless terminal are components. The component exposes its functionality through its ports, which represents a point of contact between the component and its environment. The wireless terminal component is declared to have a single send-request port, and the SAG has a single receive-request port. The connector includes the network connections between the wireless terminal and the SAG. A connector includes a set of interfaces in the form of roles, which may be seen as an interface to a communication channel. The rpc connector has two roles designated caller and callee. The topology of this system is defined by listing a set of attachments, each of which represents an interaction between a port and some role of a connector.

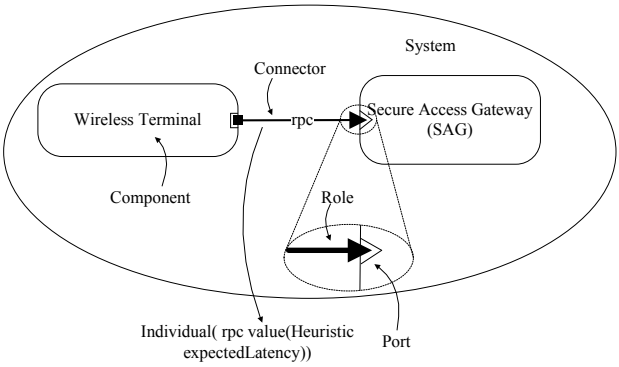

Fig. 1. Elements of an ADML Description

Table 1. The architectural model in ADML

Element Axioms	Property Axioms	Facts
Element(System complete Thing restriction (hasComp minCardinality (1))) ;	ObjectProperty(has Comp domain(unionOf (System Comp)) range(Comp));	Individual(sys type (System)); Individual(term type (Comp)); Individual(SAG type (Comp)); Individual(send-request type (Port)); Individual(receive-request type (Port));
Element(Comp complete Thing unionOf (restriction (hasPort minCardinality (1)) restriction (hasComp minCardinality (1))));	ObjectProperty(hasPort domain(Comp) range(Port));	Individual(caller type (Role)); Individual(callee type (Role)); Individual(sys value(hasComp term));
Element(Connector complete Thing restriction (hasRole minCardinality (2)));	ObjectProperty(hasRole domain(Connector) range(Role));	Individual(sys value(hasComp SAG)); Individual(sys value(hasConnector rpc));
	DatatypeProperty(Heuristic domain(Connector) range(Float));	Individual(term value(hasPort send-request)); Individual(SAG value(hasPort receive-request));
	ObjectProperty(attached domain(Port) range(Role));	Individual(rpc value(hasRole caller)); Individual(rpc value(hasRole callee)); Individual(rpc value(Heuristic expectedLatency)); Individual(send-request value(attached caller)); Individual(receive-request value(attached callee));

2.2 Components

A component provides an abstract definition of externally visible behavior—i.e., the behavior that is visible to, and may be observed by, a system containing the component. A component defines

1) the ports which may be used to represent what is traditionally thought of as an interface: a set of operations available on a component.

2) its states and state transitions.

The syntax structure of components is outlined in Figure 2. Many features are omitted from this overview.

A component declares sets of port constituents. Those port constituents are visible to the system. Other components of the system can be wired up (by connectors) to those port constituents. Thus communication between components is defined by the ports.

A component must provide the objects and functions named in the port constituents. Thus, for example, other components can be connected to call its provided

```
type_declaration ::=
  type identifier is component_expression ';'
component_expression ::=
  component
    {component_constituent}
  end [component]
component_constituent : :=
{ port_declaration}
| behavior
  [declaration_list]
 begin
  { state_transition_rule }

port_declaration : := Port name '= {'
  {property_declaration;}
  {representation_declaration;}
'}'

declaration_list : := ADML facts - see Section 3

state_transition_rule : :=
  Transition '(< ' trigger ', ' transition_body '>) ' '; ; '

trigger ::= pattern - see Section 2.3

pattern ::= ADML axioms- see Section 3

transition_body : := {state assignment}
  [ restricted_pattern ]
```

Fig. 2. Outline of component syntax

functions. Conversely, a component may call its requires functions and assume they are connected to call provided functions of other components. Connectors between required and provided functions define synchronous communication.

A component specifies the types of events it can observe and generate by declaring, respectively, in and out ports. Connectors in the system can call the in ports of a component, thus generating in events which the component can observe; conversely, the component can call its out ports, thereby generating events which the system can observe. Thus connectors define asynchronous communication between components.

A component optionally contains a behavior which consists of a set of objects, functions, and transition rules. The set of objects, functions is described as ADML facts (see Section 3). Facts declared in a component model state. Transition rules model how the component react to patterns of observed events by changing its states and generating events. The behavior of a component in a system is constrained to be consistent with the component's behavior.

A component observes in events from the system. It reacts by executing its transition rules and generating out events which are sent to other components. A component observes calls to its provides functions; the component must declare functions in its behavior that are executed in response to these calls. When a component calls its requires functions it depends upon the system to connect those calls to provides functions in other components.

2.3 Transition Rules

One of the key ingredients of the behaviors in a component is a set of transition rules that model how the component react to patterns of observed events by changing their states and generating events. A state transition rule has two parts, a trigger and a body. A trigger is an event pattern (a finite set of facts). A body is an optional set of state assignments followed by a restricted pattern which describes a finite set of facts. In our description frameworks for transition rules, functional descriptions are

essentially the state-based and use at least pre-state and post-state constraints to characterize intended executions of a transition rule. On the basis of the above consideration, we give some formal definitions related to transition rules:

Definition 1 (Atomic transition rules). An atomic transition rule is a tuple Transition(t)= Transition(<trigger, transition_body>)= Transition(<Pre, Effects>), where Pre is a finite set of facts specifying the preconditions for the execution of t; and Effects is a finite set of facts holding in the newly-reached world by the transition rule's execution. A function body is treated as if it was a transition rule that is triggered by a function call; function calls are treated as events.

A close observation on the state transition rule reveals some resemblance between transition rules and DDL($\mathcal{SHON}$(D)) actions. As mentioned in [9-12], the formulas in both Pre and Effects are conferred with well-defined semantics encoded in some TBox, which specifies the domain constraints in consideration.

The execution semantics of transition rules are as follows. If any of the preconditions of the transition rules are satisfied, the process of arbitrarily choosing one of the transition rules, executing its rule body is repeated until none of the preconditions are satisfied.

Executing a rule body consists of changing the state of the behavior part (by calling operations of facts declared there), generating new events defined by the instance of the restricted pattern, and adding the new events to the execution of the system.

Composite transition rules are constructed from atomic transition rules with the help of classic constructors in ADML. Both atomic and composite transition rules are transition rules:

Definition 2 (Transition rules). Transition rules are built up with the following rule:

t_i, t_j ::= Transition(t) | Transition(t_i test) | choiceOf(t_i t_j) | sequenceOf(t_i t_j) | Transition(t_i iteration), where t is an atomic transition rule; t_i, t_j denote transition rules.

We name sequenceOf(t_i t_j), choiceOf(t_i t_j), Transition(t_i iteration) and Transition(t_i test) as sequence, choice, iteration and test transition rules, respctively.

Remark: Note that test Transition(t_i test) is used to check the executability of the transition rules, which can be reduced to the satifiablity-checking of preconditions of the component transition rules. Test transition rules' execution effects no changes to the world.

ADML is a promising Architecture Description Language (ADL) towards a full realization of the representing and reasoning about both static knowledge and dynamic knowledge in concurrent and distributed systems. In addition to the features inherited from $\mathcal{SHON}$(D), i.e., expressive power in static knowledge representation and decidability in reasoning, DDL($\mathcal{SHON}$(D)) still employs actions to capture functionalities of transition rules. Hence it is intuitive to model transition rules by actions in DDL($\mathcal{SHON}$(D)). As demonstrated in this section, the functionalities of transition rules can be semantically transformed into actions in DDL($\mathcal{SHON}$(D)) by a proper domain ontology (TBox). As a result, all kinds of reasoning tasks concerning the functionalities of transition rules thus can be reduced to the reasoning about actions in DDL($\mathcal{SHON}$(D)).

3 ADML as the DDL($\mathcal{SHON}$(D)) Dynamic Description Logic

ADML is very close to the DDL($\mathcal{SHON}$(D)) Dynamic Description Logic which is itself an extension of the $\mathcal{SHON}$(D) Description Logic [5] (extended with a dynamic dimension[6,9]). ADML can form descriptions of components, connectors, and systems using some constructs. Given the limited space available, in this article I will not delve into the details of the ADML syntax. ADML axioms, facts, and transition rules are summarized in Table 2 below. In this table the first column gives the ADML syntax for the construction, while the second column gives the DDL($\mathcal{SHON}$(D)) Dynamic Description Logic syntax.

Because ADML includes datatypes, the semantics for ADML is very similar to that of Dynamic Description Logics that also incorporate datatypes, in particular DDL($\mathcal{SHON}$(D)).

The specific meaning given to ADML is shown in the third column of Table 2. A DDL(X) model is a tuple M = (W, T, Δ, I), where,

W is a set of states;

T : $N_A \rightarrow 2^{W \times W}$ is a function mapping action names into binary relations on W;

Δ is a non-empty domain;

I is a function which associates with each state $w \in W$ a description logic interpretation $I(w) = < \Delta, \cdot^{I(w)} >$, where the mapping $\cdot^{I(w)}$ assigns each concept to a subset of Δ, each role to a subset of $\Delta \times \Delta$, and each individual to an element of Δ.

What makes ADML an architecture description language for concurrent and distributed systems, is not only its semantics, which are quite standard for a dynamic description logic, but also the use of transition rules for changes at the architectural level, the use of datatypes for data values, and the ability to use that dynamic description logic algorithms and implementations to provide reasoning services for ADML.

Table 2. ADML axioms, facts, and transition rules

ADML Syntax	DDL Syntax	Semantics
Elements		
Element(A partial C_1 ... C_n)	$A \sqsubseteq C_1 \sqcap ... \sqcap C_n$	$A^I \subseteq C_1^I \cap ... \cap C_n^I$
Element(A complete C_1 ... C_n)	$A = C_1 \sqcap ... \sqcap C_n$	$A^I = C_1^I \cap ... \cap C_n^I$
SubElementOf (C_1 C_2)	$C_1 \sqsubseteq C_2$	$C_1^I \subseteq C_2^I$
EquivalentElements (C_1 ... C_n)	$C_1 = ... = C_n$	$C_1^I = ... = C_n^I$
DisjointElements (C_1 ... C_n)	$C_i \sqcap C_j = \bot$, $i \neq j$	$C_i^I \cap C_j^I = \emptyset$, $i \neq j$
Datatype(D)		$D^I \subseteq \Delta_D$
Datatype Properties (U)		
DatatypeProperty(U super(U_1) ... super(U_n))	$U \sqsubseteq U_i$	$U^I \subseteq U_i^I$
DatatypeProperty(U range(D_1) ... range(D_l))	$\top \sqsubseteq \forall U.D_i$	$U^I \subseteq \Delta \times D_i^I$
DatatypeProperty(U Functional)	$\top \sqsubseteq \leq \infty U$	U^I is functional

Table 2. *(continued)*

ADML Syntax	DDL Syntax	Semantics
SubPropertyOf(U_1 U_2)	$U_1 \sqsubseteq U_2$	${U_1}^I \subseteq {U_2}^I$
EquivalentProperties(U_1 ... U_n)	$U_1 = ... = U_n$	${U_1}^I = ... = {U_n}^I$
Object Properties		
ObjectProperty(R super(R_1) ... super(R_n))	$R \sqsubseteq R_i$	${R}^I \subseteq {R_i}^I$
ObjectProperty(R domain(C_1) ... domain(C_m))	$\geq 1\ R \sqsubseteq C_i$	${R}^I \subseteq {C_i}^I \times \Delta$
ObjectProperty(R range(C_1) ... range(C_l))	$\top \sqsubseteq \forall\ R.C_i$	${R}^I \subseteq \Delta \times {C_i}^I$
ObjectProperty(R Functional)	$\top \sqsubseteq \leq \infty R$	${R}^I$ is functional
ObjectProperty(R Transitive)	Tr(R)	${R}^I = ({R}^I)^{\Downarrow}$
SubPropertyOf(R_1 R_2)	$R_1 \sqsubseteq R_2$	${R_1}^I \subseteq {R_2}^I$
EquivalentProperties(R_1 ... R_n)	$R_1 = ... = R_n$	${R_1}^I = ... = {R_n}^I$
Annotation		
AnnotationProperty(S)		
Facts		
Individual(o type(C_1) ... type(C_n))	$\wr \in C_i$	${o}^I \in {C_i}^I$
Individual(o value(R_1 o_1) ... value(R_n o_n))	$<o, o_i> \in R_i$	$< {o}^I, {o_i}^I > \in {R_i}^I$
Individual(o value(U_1 v_1) ... value(U_n v_n))	$<o, v_i> \in U_i$	$< {o}^I, {v_i}^I > \in {U_i}^I$
SameIndividual(o_1... o_n)	$o_1 = ... = o_n$	${o_1}^I = ... = {o_n}^I$
DifferentIndividual(o_1... o_n)	$o_i \neq o_j$, $i \neq j$	${o_i}^I \neq {o_j}^I$, $i \neq j$
negationOf (φ)	$\neg\varphi$	$(M,w) \not\models \varphi$
disjunctionOf (φ φ' ...)	$\varphi \vee \varphi'$	$(M,w) \models \varphi$ or $(M,w) \models \psi$
diamondAssertion(π φ)	$<\pi>\varphi$	$\exists w' \in W.((w, w') \in T(\pi)$ and $(M,w') \models \varphi)$
Transition rules		
Transition(α)	α	$T(\alpha) = T(P, E) = \{ (w, w') \mid$ ① $(M,w) \models P$, ② $(M,w) \models \neg\varphi$ for every $\varphi \in E$, ③ $(M,w') \models E$, ④ ${C}^{I(w')} = ({C}^{I(w)} \cup \{u^I \mid C(u) \in E\}) \setminus \{u^I \mid \neg C(u) \in E\}$, and ⑤ ${R}^{I(w')} = ({R}^{I(w)} \cup \{(u^I, v^I) \mid R(u, v) \in E\}) \setminus \{(u^I, v^I) \mid \neg R(u, v) \in E\}$. $\}$
Transition (φ test)	$\varphi?$	$\{(w, w) \mid w \in W$ and $(M,w) \models \varphi\}$
choiceOf (π π' ...)	$\pi \cup \pi'$	$T(\pi) \cup T(\pi')$
sequenceOf (π π' ...)	$\pi\ ;\ \pi'$	$\{ (w, w') \mid \exists w''.(w, w'') \in T(\pi)$ and $(w'', w') \in T(\pi') \}$
Transition (π iteration)	π^*	reflexive and transitive closure of $T(\pi)$

4 Summary and Outlook

In this paper we presented ADML a new formal language and/or conceptual model for representing system architectures. We described the main features of ADML, its rationale, and technical innovations. By embracing transition rules into ADML, ADML combine the static knowledge provided by the system requirements with the dynamic descriptions of the computations provided by transition rules, and support the representing and reasoning about both static knowledge and dynamic knowledge in concurrent and distributed systems. ADML can be viewed as syntactic variants of dynamic description logic DDL($\mathcal{SHON}$(D)). The functionalities of the transition rules are abstracted by actions in DDL($\mathcal{SHON}$(D)), while the domain constraints, states, and the overall system objectives are encoded in TBoxes, ABoxes and DL-formulas, respectively. So the main reasoning problem in ADML can be reduced to knowledge base (KB) satisfiability in the DDL($\mathcal{SHON}$(D)) dynamic description logic. Afterwards, dynamic description logic algorithms and implementations can be used to provide reasoning services for ADML. ADML has evolved from several sources:1) Rapide[13] (a concurrent event-based simulation language for defining and simulating the behavior of system architectures.), 2) Acme[14] (a common representation for software architectures), 3) DDL[6][9] (for event patterns and formal constraints on concurrent behavior expressed in terms of description logics). While ADML is still too new to tell whether it will succeed as a community-wide tool for architectural development, we believe it is important to expose its language design and philosophy to the broader software engineering community at this stage for feedback and critical discussion.

In the future, we plan to investigate how we can leverage Distributed Dynamic Description Logics (D3Ls) to support a larger variety of heterogeneous systems with our approach. Another issue for future work is the design of "practical" algorithms for DDL($\mathcal{SHON}$(D)) reasoning.

Acknowledgements. This work is supported by the Fundamental Research Funds for the Central Universities (No.11QG13) and the National Science Foundation of China (No.71101048).

References

1. Dashofy, E.M., Van der Hoek, A., Taylor, R.N.: A comprehensive approach for the development of modular software architecture description languages. ACM Transactions on Software Engineering and Methodology 14(2), 199–245 (2005)
2. Azevedo, R., Rigo, S., Bartholomeu, M.: The ArchC architecture description language and tools. International Journal of Parallel Programming 33(5), 453–484 (2005)
3. Mishra, P., Dutt, N.: Architecture description languages for programmable embedded systems. IEE Proceedings-Computers and Digital Techniques 152(3), 285–297 (2005)
4. Pérez, J., Ali, N., Carsí, J.Á., Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 123–138. Springer, Heidelberg (2006)

5. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F.: The description logic handbook: theory, implementation, and applications. Cambridge University Press (2003)
6. Shi, Z., Dong, M., Jiang, Y., Zhang, H.: A logical foundation for the semantic web. Science in China, Ser. F 48(2), 161–178 (2005)
7. Artale, A., Franconi, E.: A temporal description logic for reasoning about actions and plans. J. Artif. Intell. Res. 9, 463–506 (1998)
8. Baader, F., Lutz, C., Milicic, M., Sattler, U., Wolter, F.: Integrating description logics and action formalisms: First results. In: Proc. Natl. Conf. Artif. Intell., vol. 2, pp. 572–577 (2005)
9. Chang, L., Shi, Z., Gu, T., Zhao, L.: A Family of Dynamic Description Logics for Representing and Reasoning About Action. J. Autom. Reasoning, 1–52 (2010)
10. Wang, Z., Yang, K., Shi, Z.: Failure Diagnosis of Internetware Systems Using Dynamic Description Logic. J. Softw. China 21, 248–260 (2010)
11. Wang, Z., Guo, J., Wu, K., He, H., Chen, F.: An architecture dynamic modeling language for self-healing systems. Procedia Engineering 29(3), 3909–3913 (2012)
12. Wang, Z., Zhang, D., Shi, Z.: Multi-agent based bioinformatics integration using distributed dynamic description logics. In: Int. Conf. Semant., Knowl., Grid., China, pp. 66–71 (2009)
13. Luckham, D.C., Vera, J.: An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering 21(9), 717–734 (1995)
14. Garlan, D., Monroe, R., Wile, D.: Acme: an architecture description interchange language, CASCON First Decade High Impact Papers. USA, pp. 159–173 (2010)