Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 12–31
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Moving architectural description from under the technology lamppost

Nenad Medvidovic [a,*], Eric M. Dashofy [b,1], Richard N. Taylor [b,2]

[a] Computer Science Department, Viterbi School of Engineering, University of Southern California, Los Angeles, CA 90089, USA
[b] Department of Informatics, Donald Bren School of Information and Computer Sciences, University of California, Irvine, CA 92697, USA

Available online 10 October 2006

Abstract

In 2000, we published an extensive study of existing software architecture description languages (ADLs), which has served as a useful reference to software architecture researchers and practitioners. Since then, circumstances have changed. The Unified Modeling Language (UML) has gained popularity and wide adoption, and many of the ADLs we studied have been pushed into obscurity. We argue that this progression can be attributed to early ADLs' nearly exclusive focus on technological aspects of architecture, ignoring application domain and business contexts within which software systems and development organizations exist. These three concerns – technology, domain, and business – constitute three "lampposts" needed to appropriately "illuminate" software architecture and architectural description.

Keywords: Software architecture; Software architecture description languages

1. Introduction

Software architecture emerged as a field of software engineering research in the early 1990s, after the publication of Perry and Wolf's seminal paper [33]. Quickly thereafter a number of architecture-based software development notations, methods, techniques, and tools were formulated. Of particular interest to the early software architecture researchers and (to a somewhat lesser extent) practitioners were the notations for modeling software architectures. These came to be known as architecture description languages, or ADLs.

Several ADLs appeared in the software engineering literature in relatively short succession: Acme [15], C2 [39], Darwin [25], MetaH [4], Rapide [24], UniCon [37], Weaves [16], Wright [1], and so on. Several other, already existing notations were also claimed to be ADLs, either by their developers or users: module interconnection languages (MILs) [34], StateCharts [17], CHAM [19], LILEAnna [41], UML 1.x [7], and so on. The early understanding of this subject was so immature that it was very difficult to argue for or against considering a given software modeling notation to be an ADL; languages often "became" ADLs simply because someone referred to them as such. However, one thing common across all these notations was the implication that they would significantly alter and improve the way software is produced.

Our study, which was conducted in the late 1990s and published in early 2000 [27], provided some much needed answers. It provided a technical basis for determining what an ADL is and is not. In particular, this allowed us to formulate a clear argument *against* the inclusion of several commonly used notations into the ADLs category. Our conclusions regarding some of these "non-ADLs", such as StateCharts or CHAM, largely agreed with the conventional wisdom of the time, while in the case of others, such as UML 1.x, they were surprising (even though subsequent developments – the architecture

* Corresponding author. Tel.: +1 213 740 5579; fax: +1 213 740 4927.
E-mail addresses: neno@usc.edu (N. Medvidovic), edashofy@ics.uci.edu (E.M. Dashofy), taylor@ics.uci.edu (R.N. Taylor).
[1] Tel.: +1 949 824 4101; fax: +1 949 824 1715.
[2] Tel.: +1 949 824 6429; fax: +1 949 824 1715.

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.08.006

modeling constructs added to UML 2.x – proved our conclusions sound).

These early "first-generation" ADLs came from different sources: commercial industry, government-funded aerospace companies, standards bodies, and academia. They emerged from very different areas of software development. For example, MetaH was targeted primarily at real-time systems (with a view toward the control systems domain), while Weaves modeled asynchronous data-flow architectures (geared to the needs of weather satellite-based systems). The early ADLs also emerged from different areas of computer science, outside software engineering. For example, Rapide's predecessors were used for *hardware* architecture modeling, while Darwin grew out of a distributed computing research project.

Yet, despite these differences, the first-generation ADLs all shared certain traits. They all modeled the structural and, with the exception of Acme, functional characteristics of software systems. They invariably took a single, limited perspective on software architecture. Some, such as Rapide, focused almost exclusively on event-based modeling at the expense of other system aspects; others, such as Wright, were specifically geared toward deadlock detection in concurrent architectures; still others, such as MetaH and UniCon, were mainly concerned with process scheduling. To support such objectives the early ADLs heavily focused on formalization of software architectural models, with an eye on their analysis.

The result of this narrow focus was a set of ADLs that were deficient in various areas that were critical to many stakeholders. Most of these ADLs focused exclusively on software. For example, only MetaH had explicit support for modeling software architecture as well as the hardware on which it runs. Few first-generation ADLs were accompanied by a strategy for implementing the described architecture. Moreover, characteristics of the architecture description could often be verified using analysis tools and methods, but there was no way to ensure that the implemented system conformed to the architecture. Finally, these ADLs were not extensible in any meaningful way – it was prohibitively expensive to add features to them to support any unmet needs – severely limiting their range of applicability.

1.1. A broader notion of software architecture

Since our 2000 study, the landscape of software architecture has continued to evolve. Two major changes are of interest here. First, the notion of software architecture has been expanded, allowing us to view ADLs in a new light. Second, notations and approaches for modeling software architecture have themselves continued to evolve, thereby providing us with information about directions in which the architecture modeling community is heading.

There is not today, nor has there ever been, a clear consensus on a definition of software architecture. Yet defining software architecture is critical to understanding what constitutes an architecture description language. Literally hundreds of definitions have been proffered; many have been cataloged by the Software Engineering Institute (SEI) and are available on the Web [10]. Our initial study struggled with this problem as well, identifying several alternative notions of what constituted architecture and what made up an ADL. Based on a broad survey of architecture description notations and approaches, we identified that ADLs capture aspects of software design centered around a system's *components*, *connectors*, and *configuration*. This framework is concordant with what is supported in most first-generation ADLs, which, as we noted above, primarily tend to capture architectural structure along with properties of that structure. It was this framework that provided us with a "litmus test" as to whether a modeling notation was or was not an ADL.

Since that time, however, other concerns have become increasingly prominent in the software engineering community, specifically those derived from domain-specific and business needs. Additional insight has come from notions of architecture beyond the software engineering community – systems engineers, for example, have a broad notion of what constitutes architecture. Conferences such as the Working IEEE/IFIP Conference on Software Architecture (WICSA) have brought together researchers and practitioners from the software architecture, systems engineering, and enterprise architecture communities. All this has revealed that, while structural concerns retain a place of primacy in software architecture modeling, they do not and should not define its scope. Instead, we propose a broader definition of software architecture:

Definition. A software system's architecture is the set of *principal design decisions* about the system.

Design decisions encompass every aspect of the system under development, including:

* design decisions related to system *structure* – for example, "there should be exactly three components in the system, the 'data store,' the 'business logic,' and the 'user interface' component;"
* design decisions related to *behavior* (also referred to as *functional*) – for example, "data processing, storage, and visualization will be handled separately;"
* design decisions related to *interaction* – for example, "communication among all system elements will occur only using event notifications;"
* design decisions related to the system's *non-functional properties* – for example, "the system's dependability will be ensured by replicated processing modules;"
* design decisions related to the system's *development* itself – for example, the process that will be used to develop and evolve the system; and

* design decisions related to the system's *business position* – for example, its relationship to other products, time-to-market, and so on.

An important term that appears in the above definition is "principal." It implies a degree of importance that grants a design decision "architectural status." It also implies that not all design decisions are architectural, that is, they do not necessarily impact a system's architecture. How one delimits "principal" will depend on what the system goals are. Ultimately, the system's *stakeholders* (including, but not restricted only to the architect) will decide which design decisions are important enough to include in the architecture, and which are not. For example, consider a design decision such as "the log viewer component will check for new log entries once every second." For systems where real-time log viewing is needed, the log refresh interval might be specified as part of the architecture. For other systems, this may simply be an implementation detail and be elided from the description of the system's architecture.

From this definition of architecture, we can also derive definitions for architecture models, description languages, and the act of modeling:

Definition. An architecture *model* is an artifact or document that captures some or all of the design decisions that make up a system's architecture. Architecture models are sometimes referred to as architecture descriptions.

Definition. An *architecture description language* is a notation in which architecture models can be expressed.

Definition. Architecture *modeling* is the effort to capture and document the design decisions that make up a system's architecture.

1.2. A new perspective of ADLs

This broader perspective changes the test for whether a notation can be considered an architecture description language. Instead of defining ADLs based on features (e.g., the ability to model high-level system structure), they are defined by *stakeholder concerns* – whether a notation can adequately capture design decisions deemed principal by the system's stakeholders. As we will explain below, these concerns are substantially broader than those captured by first-generation ADLs.

In some sense, the broader definition may seem like a step backward in that it is a relaxation of our original litmus test. In this new light, even some determinations we made in our original study about what is and is not an ADL may change. However, this raises the importance of discussing the *adequacy* of different notations for modeling software architecture. For example, under our new definition a notation that is not suitable for high-level structural modeling may now be classified as an ADL, but one that is clearly deficient in a critical respect.

This induces a new way to evaluate architecture modeling notations: not based on how they model basic structural elements like components, connectors, and interfaces, but rather how adequately they model concerns important to their target stakeholders.

Our criticisms of the early ADLs, and, to a somewhat lesser extent UML, stem from this issue. Those notations focused primarily on general concerns related to the technical aspects of designing and constructing software systems. However, we argue that additional concerns must be illuminated: those from application *domains*, as well as *business* needs. We posit that these three "lampposts" (technology, domain, and business) can help explain the reasons behind the limited impact of the first-generation ADLs and the shortcomings of UML. In fact, they provide a means for a more complete treatment of ADLs than was given in our original study [27]. We also posit that several more recent, "second-generation" ADLs, including UML 2.0, can be better understood and put in their proper context with the help of the three lampposts.

This paper, therefore, provides a different perspective of ADLs, both first- and second-generation, with an improved understanding of a system's development context and an ADL's role in it. In order to properly understand, and assess, an architecture modeling language, we believe that one needs to understand a number of issues that go beyond the usual system structure and behavioral concerns: the many overlapping and sometimes conflicting non-functional properties; the characteristics and needs of the application domain(s) at which the ADL is targeted; the system stakeholders; the organization's business goals (e.g., managing architectural assets to support product families); and so on. Capturing all such concerns with a single, narrowly focused notation (e.g., a first-generation ADL) is impossible. At the same time, as we will discuss, it is also impractical to try to do so with a "universal" notation, such as UML.

In light of this, we will argue in this paper that a truly effective ADL must strike a proper balance between a strict focus on recurring technical concerns and the extensibility needed to include the concerns mandated by different application domains and business contexts. Our principal objectives are to highlight and improve the current understanding of

1. the limitations of purely technical approaches to software architecture, as in the first-generation ADLs;
2. the justified attraction, but also limitations of "one-size-fits-all" approaches, as embodied in UML; and
3. the need for specialization of a modeling language based on the demands of a specific application, application family, or application domain.

Most importantly, returning to our metaphor, we will argue that, in order to realize their impact, ADLs have to step away from the technology lamppost and let in some light from the remaining two lampposts. This is not to say that there exists today an "ideal" or "perfect" ADL; in fact, we believe no such ADL can emerge because of

the diversity of concerns that can impact systems development. Rather, we argue that ADLs can be made more effective by taking into account concerns from all three lampposts, and that several recent ADLs are indeed moving in this direction.

The remainder of the paper is organized as follows: Section 2 fleshes out our vision of the three lampposts and how they can (and should) influence software architecture modeling. Section 3 briefly recaps the lessons taught by first-generation ADLs. Section 4 describes, in some detail, a set of "second-generation" architecture description languages that have evolved from first-generation ADLs and are more appropriately positioned under the three lampposts. Section 5 summarizes lessons we have learned since our 2000 study, and extrapolates some future directions for architecture modeling. Section 6 concludes the paper.

2. The three lampposts

The old story about the man who lost his keys provides some context for our perspective on the design of architecture description languages. One variant of the story states that one night a man dropped his keys in a parking lot just before getting to his car. A friend saw him searching for the keys on the ground under a lamppost, but quite some distance away from the car. When asked why he was looking for the keys so far away from where he dropped them, the man replied, "Because the light is much better over here." This story serves as a cautionary tale to all researchers: from time to time we fail to discover what is needed simply because we are looking for a solution merely in places where it is easiest to perform the research.

The notion of "what is needed" is quite variable of course. For researchers in an academic computer science program, "what is needed" may simply be some interesting technical innovation. For an engineer working in a specific application domain, "what is needed" might be something that provides exceptional power in that domain, regardless of whether it has any value in any other situation.

This characterizes how many architecture modeling languages have been developed: the concerns addressed by modeling languages have tended to reflect only the characteristics and specific interests of their creators. The creators of first-generation ADLs were largely software engineering researchers, and as such these ADLs model concerns that are of particular interest to various segments of the research community. For example, determining appropriate scheduling policies, deadlock-freedom, or the best separation of functionality into components and connectors are the focus of various first-generation ADLs. These concerns primarily arise from technological and engineering problems in constructing and maintaining large software systems, and are illuminated by the 'technology lamppost.'

We argue that an excessive or exclusive focus on concerns found under the technology lamppost is a critical failing of early architecture description languages: they do not provide "what is needed" to satisfy a robust software engineering ecology. As architecture modeling has evolved, languages that encompass concerns beyond the technology lamppost are those that have been the most influential, widely adopted, and, arguably, successful. We believe that this is because they address concerns important to a wider variety of critical stakeholders.

Developing a successful software system means satisfying a wide variety of stakeholders. Indeed, software engineers whose expertise and interests lie mostly under the technology lamppost are included. However, unless the system's target users are themselves software engineers, the concerns of the users will fall mostly under another lamppost – the domain lamppost. These stakeholders are interested mostly in how well the software models and addresses problems that they are encountering in their own application domain. Additionally, the production of a software product typically does not occur in isolation. Products are largely interrelated – sharing components, providing complementary functionality, being built by overlapping teams of developers, and filling a particular space in a market. These concerns fall under a third lamppost – the business lamppost.

These three lampposts – technology, domain, and business – can provide several new insights about architecture description languages:

* First, they provide a way to classify and evaluate architecture description languages, by asking to what extent a given ADL addresses concerns from each lamppost;
* Second, they provide a possible way to explain the relative success or failure of past ADLs;
* Third, they provide guidance for developers of new ADLs, as a reminder to include and balance support for concerns from all three lampposts.

We thus posit that these three lampposts – technology, domain, and business – provide the necessary broad perspective on architecture description languages and their role in supporting product development as a whole.

Fig. 1 shows these three types of concerns as intersecting areas; overlapping circles of light from the three lampposts, if you will. The technology circle is concerned with the specific technical bases for describing, developing, and reasoning about architectural models. It includes formalisms, analysis techniques, and supporting tools. The domain circle is concerned with specific application domain knowledge. It includes knowledge about the domain's nature and underlying science, typical approaches for solving problems in the domain, and standard elements of solutions to problems within that domain. The business circle is concerned with markets, organizations, the relationships of different products to one another, as well as the processes, people, finances, and organizations that all influence and are influenced by software systems. In the subsections below, we examine each of the circles of light, and discuss their areas of overlap.

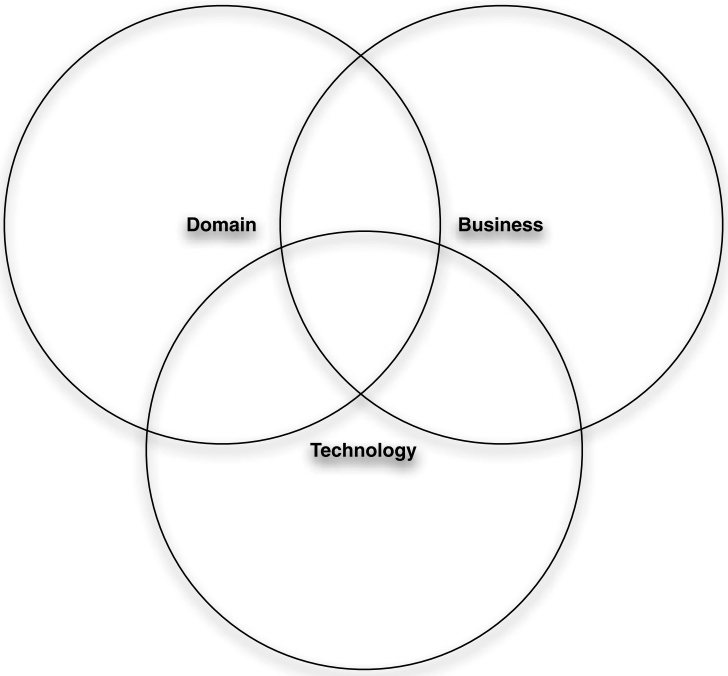

Fig. 1. The "three lampposts": domain of application, business concerns, and technology.

2.1. Technology

The technology lamppost illuminates concerns surrounding the recurring technical challenges of engineering software systems, and creating the means for representing and reasoning about their architectures. This includes the perspectives of classical computer science and software engineering, including work on the most theoretical side, such as abstract formal models, as well as work on the very practical, such as linguistic means for describing architectures. It primarily includes work that is unfettered with (or uninformed by!) concerns of a particular implementation domain or business context.

The focus on identifying the critical abstractions, or conceptual foundations of software architectures, has led, for example, to discussions about what components, connectors, and interfaces are, and how important they are in modeling systems. Linguistic concerns have shaped discussions regarding the relative merits of declarative or imperative ADLs. Interoperability among different ADLs has been the focus of another line of work.

Many of the discussions and choices regarding fundamental modeling techniques have been driven by concerns for the types of analyses that can be performed on the various models. For instance, the earliest work in ADLs seems to have been largely shaped by the need to capture the interfaces of components in such a way that mismatches between interfaces could be automatically identified. Later work in analysis has included data flow analysis, analysis for potential deadlocks, performance estimation, system composition, and others. Often, these ADLs leveraged pre-existing analyzable formalisms and applied their concepts to software architectures, since analysis techniques and tools were already available.

Much of the work done in building a conceptual foundation for the modeling and analysis of software architectures has been accompanied by work to create a technical infrastructure of tools and environments. Such infrastructure allows experimentation with the concepts and enables attempts to apply the results in practical settings. Of particular interest in this area are ADLs developed with the intent that models can be directly mapped into implemented systems. Models developed in ADLs such as SDL [40] and Executable UML [29] can be directly "compiled" into (partial) implementations, while those expressed in Weaves and C2 can be mapped to implementations using available architecture frameworks.

Most of the early work in the area of architectural description was conducted almost exclusively under this lamppost. A representative, though by no means only, example is the chemical abstract machine (CHAM) model [19]. This model has proven to be of some theoretical interest, but there is little evidence that it has any direct practical application, or that it was shaped by a specific practical need or context.

2.2. Domain

The domain lamppost illuminates concerns driven and informed by the knowledge of a specific application domain. In a sense, therefore, the "domain lamppost" is really a host of lampposts, one for each domain that someone cares to identify. The commonality, however, is in the approach: choices made for representing and reasoning

about systems in a particular domain are driven by knowledge of that domain, and that knowledge can be used to shape the representation and reasoning techniques employed in developing applications.

Modeling languages that address domain-specific concerns offer special support for solving problems common to a particular target domain. This distinguishes domain-specific concerns from concerns found under the technology lamppost – which address general, recurrent problems in software development independent of any particular domain. Domain lamppost concerns may also include characteristics of the domain itself. Because of this, the notations are optimized for creating models in that particular domain. Put another way, by restricting one's focus to just one domain, the smaller set of concerns that one has to worry about enables specialized, deep solutions to be created. In principle, this approach allows engineers to 'speak the language' of the target system's users. Compared to generic approaches, descriptions of applications can be created that are parsimonious and precise.

Unfortunately, much of the work in specifying and developing domain-specific applications has not been captured in rigorous notations. Frequently the domain knowledge is captured only in the minds of the engineers involved. This renders it difficult to see the exact ways domain knowledge shaped a particular approach. Moreover, if work in a domain does not result in something that can be identified as an ADL, then it becomes difficult to separate domain-focused *architecture-based* development from any type of technology that supports development specifically within that domain.

Several notable successes from first-generation ADLs included a domain-specific focus. One good example is Weaves [16], which was targeted at supporting development of satellite ground stations and has been employed by a number of development organizations. In particular, Weaves supported the creation of applications which processed, in stages, continuous telemetry data. Particular knowledge of the domain, and telemetry streams in particular, exploited in Weaves, included the arrival rate and type of data received, the independent "connection-less" character of the data, and the need for dynamic reconfiguration of the data processing. Another example is MetaH, which was targeted at aircraft and missile avionics and flight control. Specifically, MetaH supported the specification and analysis of real-time, fault-tolerant multi-processor systems. MetaH was applied by a number of organizations beyond its developers, including Boeing, the US Army, and the SEI, with significant cost savings realized from at least one effort at the US Army [14].

2.3. Business

The business lamppost illuminates concerns focused on capturing and exploiting knowledge of the business context of a given development effort. This includes a product strategy – e.g., how a product will differentiate itself in its target market, how multiple products are related to one another, how a product fits into its development organization's long-term vision, and so on. It also includes the development organization's processes for creating, managing, and evolving its products.

Costs, including financial concerns, also fall under the business lamppost. It is important to recognize that business concerns do not exist only for for-profit development organizations or for commercially sold software. Open-source and free software products also compete in the marketplace, are developed by organized groups, and are evolved and diversified into families of related products.

At first glance it may not seem as though the "business lamppost," with its context as described above, is related to or sheds any light on architecture description languages. To illustrate the importance, however, consider the issue of how an organization retains and exploits its core competencies in the face of developer turnover. One of an organization's key assets is knowledge of how it builds its products, i.e., what enables it to build those products in a superior manner to its competitors. If such knowledge is only retained in the heads of key people, then loss of one of those individuals could severely damage the company's ability to compete. If, on the other hand, the knowledge is recorded in a more generally accessible, manageable, and useful manner, it can be used in training new personnel and in assisting in the effective production of the company's products.

Some of the knowledge so recorded may include description of what stakeholder perspectives are valued, how input from those stakeholders is recorded and used, and how various and competing internal perspectives are used to shape products. Other knowledge may concern the qualities that the company values in its products, how those qualities are articulated both internally and externally, and how are they achieved, assessed, and monitored. Still other aspects include how the company knows what features or properties of its products are responsible for their sales, and how it knows which of the products' features are the critical ones. A final example concerns how a company values its product assets that are under development. That is, for products whose development cycle extends over many months, how is the value of the emerging asset accounted for; what artifacts are considered in making the valuation; and so on.

As we stated above, business concerns have been largely ignored by the ADL development community. Software engineering researchers have addressed business concerns, of course. Examples include the process modeling notations that emerged in the 1980s and 1990s and cost modeling frameworks such as COCOMO [5] which, to an extent, addresses business and technology concerns together. Still, even recent ADLs address business concerns less than technological or domain concerns.

Still, there are many business concerns that will likely never find their way into an ADL, simply because they are too distant from domain and technology concerns to

be of much value in an ADL. For example, corporate management structures, marketing plans, and organization-wide financial data will probably never be found in an ADL. However, concerns such as products' relationships to each other in product lines and cost data per component may well appear in an ADL.

2.4. Overlapping areas: shedding light from multiple lampposts

It should be apparent from the above descriptions that the three perspectives described are not mutually exclusive – several ways in which the concerns overlap are clear. Similarly it is clear that a particular ADL or related tool or technique may support objectives in more than one "circle of light."

Our perspective is that these overlaps, and technologies that support more than one concern, are important targets for developers and researchers. ADLs that address concerns from under only one lamppost are unlikely to be adopted beyond a very small number of enthusiasts. We have already seen this with, for example first-generation ADLs that addressed only general concerns from under the technology lamppost, or process modeling notations that addressed business concerns exclusive of technology or domain considerations. To be broadly adopted and to be effective in practice an ADL must satisfy the needs of diverse stakeholders, among many other factors (ease of use, return on investment, tool support, and so on). It must fit within a development organization, broadly construed. Thus an exclusive focus on a single concern (e.g., technology) is insufficient; multiple views, domain-specific concerns, and the business context must all be considered.[3]

Fig. 2 repeats the diagram of Fig. 1, but now with the different areas labeled. Our intent is to elaborate the meaning of the various areas in the diagram.

The Technology area that is exclusive of any overlap with domain knowledge or business context comprises generic concepts, description languages, tools, and infrastructure focused on recurring concerns in the development of software systems. All too frequently, research work in this area has produced solutions that are difficult to use and which have not been widely adopted – precisely because of a focus which is narrower than the full set of stakeholder perspectives necessary.

The Domain area that is exclusive of the other two perspectives comprises concerns regarding the underlying nature, principles, and science of a domain, as well as domain characteristics deemed "irrelevant" to either the business or technology perspectives.

The Business area that is exclusive of the other two concerns are those facets of business that are independent of the development organization's domain of expertise and independent of technology insofar as it relates to product development. This includes items such as financial accounting practices, human resources, and so on.

Clearly the intersections are the areas of interest. The intersection of Domain and Technology, for example, includes technological concerns that are specific to a particular domain. That is, where the technology-only area addresses recurring problems that occur while building software systems in general, the domain-plus-technology area addresses technical problems that occur while building software systems within a target domain. Application-family architectures fit within this sub-area, as do domain-specific modeling languages to capture those family architectures. The Weaves language mentioned above is an example formalism; other examples of this ilk include MetaH (where the domain is control systems) and ADAGE [3] (where the domain is avionics guidance, navigation, and control).

The intersection of Business and Technology links business concerns such as costs, product-to-product relationships, and processes to the technical construction of software systems, independent of any particular domain. Examples of tools that exist in this space include software project and process management approaches that relate process steps to specific software elements, configuration management systems that track the relationships of various software elements to one another, and architecture-centric cost modeling notations and tools. Such tools have a "one-size-fits-all" character to them, since they are domain-independent.

The intersection of Domain and Business includes the core competencies of an organization: that knowledge of a domain combined with business strategies and practices which enable the organization to succeed in that domain. To the extent that such knowledge is not supported by technology it remains somewhat ephemeral. This area is typified by classical systems engineering: a focus on modeling the domain, customer requirements, and processes to develop a solution, while deferring decisions about specific technological (e.g., implementation) details until late phases of development.

The "sweet spot" of the diagram is, of course, the region where all three lampposts overlap. We have labeled that region in Fig. 2 "product-line architectures" as most evocative of what this intersection includes and enables. Knowledge of an application domain combined with a business strategy for that domain and supported by technology enables the representation of and reasoning about a family of applications comprising a business product line. However, product lines are not the only approach that fits in this "sweet spot"; any approach that takes into account all three lampposts will fit there.

[3] As we have pointed out, there are many concerns under these lampposts that will not be appropriate for representation in an ADL, particularly concerns that exist far at the periphery and address only one of the domain, technology, or business concerns. Because of this, we believe that ADL designers should take all three lampposts into account, but they should not attempt to cover all possible concerns from all three lampposts.

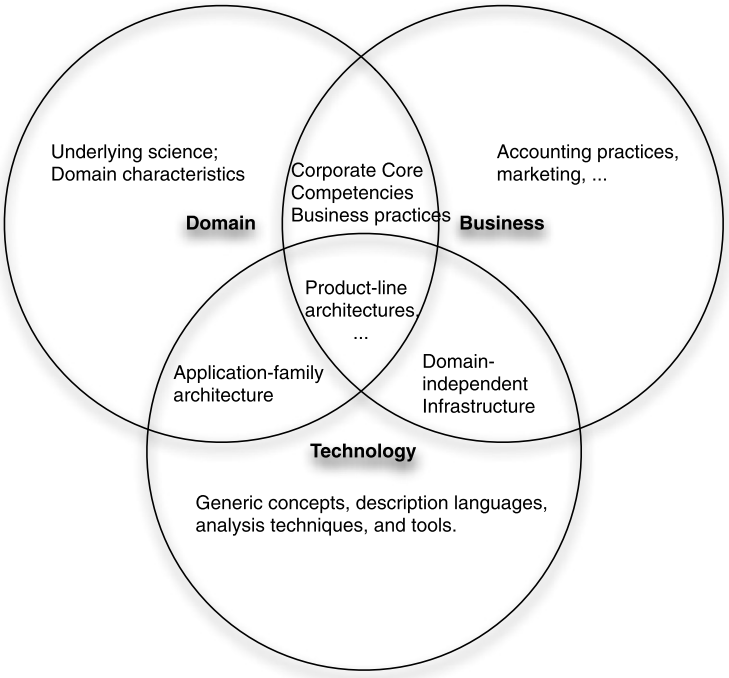

Fig. 2. The overlapping areas of the "three lampposts".

These three lampposts help illuminate the various types of research that have been undertaken in areas related to architecture description languages. However, the lampposts cast light which is, at best, imprecise and fuzzy at its boundaries. The discussion below proceeds to consider how ADLs have evolved, being driven by one or more concerns from these lampposts. The discussion is specific and more precise, however, being grounded in the specifics of several languages.

3. First-generation ADLs

As discussed in the Section 1, in our initial study of ADLs [27] we considered several classes of notations: "first-generation" ADLs, UML [7], formal modeling notations – some of which were targeted specifically at architectural concerns, such as CHAM [19] and LILEAnna [41], module interconnection languages [34], and even programming languages. We defined an ADL as a modeling notation that provides facilities for capturing a software system's *components* (i.e., computational elements), *connectors* (i.e., interaction elements), and *configurations* (i.e., overall structure). Additionally, we identified specific dimensions of components, connectors and configurations, as well as guidelines for evaluating a given notation as a potential ADL.

This definition allowed us to eliminate certain candidate notations relatively easily, and we eventually reduced the set to about 20 notations. Of those, 10 notations were included in the study as *bona fide* ADLs, as shown in Fig. 3.

Placed in the context of the three lampposts, most of these ADLs solely focused on technology: Acme [15], Aesop [30], Darwin [25], Rapide [23], SADL [35], UniCon [37], and Wright [1]. Aesop's explicit support for architectural styles [33] placed it closer than the others to the intersection of technology and application domain shown in Fig. 2 because a style, such as model-view-controller, may provide a canonical architectural solution for a given application domain, such as GUI-intensive systems.

In addition to a clear technological focus, three of the studied ADLs leveraged more extensively properties of an application domain:

* C2's ADL [26] grew out of an architectural style for graphical user interface-intensive applications [39], which was eventually broadened to a larger class of distributed, dynamic systems characterized by asynchronous event-based interaction.
* MetaH [4] focused on concurrency issues and real-time computation scheduling as found specifically within control systems.
* Weaves [16] was created to support large-volume, asynchronous data-flow architectures as found in satellite ground stations. Similar to C2, Weaves has shown potential for applicability to data-intensive systems beyond its original target domain.

Perhaps most strikingly, none of the first-generation ADLs focused on business concerns. This may very well have been their critical shortcoming. On the other hand, as will be further elaborated in the next section, UML

ADL	ACME	Aesop	C2	Darwin	MetaH
Focus	Architectural interchange, predominately at the structural level	Specification of architectures in specific styles	Architectures of highly-distributed, evolvable, and dynamic systems	Architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings	Architectures in the guidance, navigation, and control (GN&C) domain

ADL	Rapide	SADL	UniCon	Weaves	Wright
Focus	Modeling and simulation of the dynamic behavior described by an architecture	Formal refinement of architectures across levels of detail	Glue code generation for interconnecting existing components using common interaction protocols	Data-flow architectures characterized by a high volume of data and real-time requirements on its processing	Modeling and analysis (specifically, deadlock analysis) of the dynamic behavior of concurrent systems

Fig. 3. The scope and applicability of first-generation ADLs (adopted from our original study).

has attempted to provide a broad coverage of modeling issues spanning the area illuminated by both the technology and business lampposts. This may help to explain its wide adoption in industry.

3.1. The rise of UML

UML, the Unified Modeling Language, has achieved more mainstream support than any other notation for modeling software-intensive systems since the use of flowcharts. As its name implies, UML was derived through the unification of multiple influential modeling approaches: the Booch method [6], Rumbaugh's Object Modeling Technique (OMT) [36], Jacobson's Object-Oriented Software Engineering (OOSE) [20] method, Harel's statecharts [17], and various other sources.

Whether UML is an ADL, and how suitable it is for that purpose, has been the subject of study and debate. Under our broader definition of software architecture given above, it is clear that even the earlier versions of UML can be used as architecture modeling notations.

UML emerged around the same time as the first-generation ADLs and, despite its shortcomings when it came to modeling critical architectural concerns [27,28], it was rapidly and widely adopted. UML has continued to evolve. Since 1997 alone, UML has undergone four major revisions: UML versions 1.0, 1.1, 1.3, and 2.0.[4] Furthermore, the "impending" release of UML 2.0, with at least some of its enhancements targeted at improved software architecture modeling, was announced and awaited for several years. Then, very soon after this major revision of the language appeared, the main UML standards body began preparing another version (UML 2.1). Unlike the often highly specialized first-generation ADLs, UML is a huge and constantly growing composite notation, comprising 13 different, loosely connected individual notations ("diagrams" in UML parlance).

Despite its popularity, UML is not a panacea: it has shortcomings that make it less than ideal for architecture modeling in many respects. For example, UML has ambiguous semantics – a UML diagram can often be interpreted in different ways – making it less than ideal for system architectures where precision is critical (e.g., safety-critical systems). UML has unlimited extensibility in principle, but with virtually no control over it. For example, it allows one to introduce separate UML *profiles* to address different application domains and modeling needs, but it is also possible to introduce multiple profiles to address the very same concerns. In either case, unless the semantics behind those profiles are formalized using something akin to UML's Object Constraint Language (OCL) [42] – which is seldom done in practice – standard UML tools will be completely agnostic as to the intended meaning behind the profiles [28].

UML 2.0, as a second-generation ADL, will be discussed below.

4. Second-generation ADLs

The first-generation ADLs predominantly remained under the "technology lamppost" – addressing interesting technical problems, but largely ignoring domain or business concerns. The contributions and lessons of this first generation of ADLs were not lost, however: even though they did not achieve significant adoption, they inspired a second generation of ADLs. These "second generation" ADLs tend to inherit lessons from earlier ADLs, as well

[4] We became intimately familiar with this issue when we conducted an early study of UML's suitability for architectural description: by the time that work was published, we were forced to update our study three separate times as new UML versions kept appearing.

as to include more domain and business concerns. In this section, we will discuss four representatives of this second generation of architecture description languages: UML 2.0, AADL, Koala, and xADL 2.0.

4.1. UML 2.0

The latest version of UML, UML 2.0 [8], is a syntactically rich language comprised of 13 different viewpoints, which are called "diagrams" in UML parlance. Stakeholders play a large part in how UML is employed in a project. They may use as many (or as few) of the diagrams as needed. Multiple instances of each viewpoint can be used to capture increasing detail about a system. UML is a "Swiss Army Knife" of notations – it provides stakeholders with a collection of useful notations (i.e., UML diagrams) for accomplishing different modeling goals. UML is not specialized for modeling any particular domain, although its diagrams reflect a bias toward modeling systems constructed in an object-oriented (OO) way.

Early versions of UML (1.0 and 1.1) had only limited abilities to specify traditional architectural concepts from "under the technology lamppost" such as components, connectors, and deployments [28]. This has been rectified to some extent in UML 2.0. In particular, the component diagram has been almost completely overhauled to support the notion of components as loci of computation rather than just artifacts, as well as explicit specification of interfaces (both provided and required) and ports. Additionally, the new composite structure diagram allows hierarchical modeling, which lets stakeholders better express the relationships between different models and architectural concepts. As we have noted, however, as UML has changed so has the understanding of what constitutes 'software architecture.'

From the perspective of concern-driven modeling, the analogy of UML to a Swiss Army Knife remains useful. Each UML diagram can be seen as a tool for modeling a particular concern. However, the number of tool-diagrams is finite. UML does not have diagrams for modeling every possible architectural concern. For example, there is no explicit support in UML for capturing product variants, or the evolution of a system's architecture over time. Some diagrams can be employed for modeling multiple concerns by interpreting the same symbols in different ways, or by specializing the diagrams using UML's extension mechanisms. In general, UML diagrams and symbols can be interpreted in different ways by different stakeholders. This intentional ambiguity is how UML maintains its generic, domain-independent nature. Specialization mechanisms allow users to define new attributes (called stereotypes and tagged values) and constraints that can be applied to existing elements. A collection of these additional attributes and constraints is known as a UML profile. Profiles are used to specialize UML to reduce ambiguity and better capture domain- and project-specific concerns.

Profiles cannot define new diagram types, however – they can only decorate and specialize existing diagram types and their elements. Sometimes, this is not enough. An example is SysML [38], a notation developed by a consortium of large systems development organizations as an extension of UML. SysML uses the built-in UML extension mechanisms (primarily stereotypes) to specialize existing UML constructs for specific purposes. For example, the standard UML Package element is stereotyped with a ≪DependencySet≫ stereotype to represent a special kind of package containing dependencies. However, where simply specializing existing UML diagrams and elements is not enough, SysML extends UML itself to include entirely new viewpoints and elements. For example, the SysML Requirements diagram is a new view for the specific purpose of capturing system requirements in a more detailed manner than UML's use case diagrams allow.

4.1.1. UML example

To illustrate how UML models a system, we present some UML models of a basic three-tier Web application consisting of three components: a Web browser that serves as the user interface, a business logic component that processes data from the Web browser, and a database component that is responsible for persistent data storage. (In a real system, each of these coarse-grained components would likely be replaced with several finer-grained components; the use of coarse-grained components here is to simplify our example). The structure of the application might be modeled using the UML component diagram shown in Fig. 4.

This diagram depicts the components in the application and their dependencies. It leverages several new features of UML 2.0: components are now specializations of UML classes, and component instances are modeled like objects. Additionally, this diagram uses explicit ports, as well as provided and required interfaces to describe the dependency relationships among the components. This is one view of the system's architecture. Another view might capture its behavior. The behavior

Fig. 4. UML component diagram of a simple Web application.

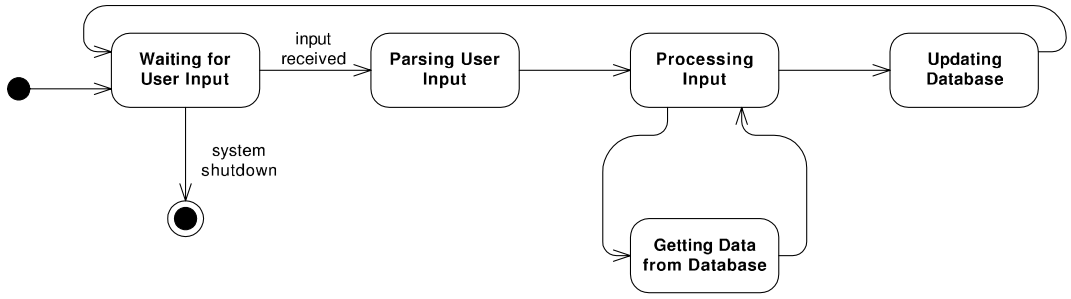

Fig. 5. UML statechart diagram of a simple Web application.

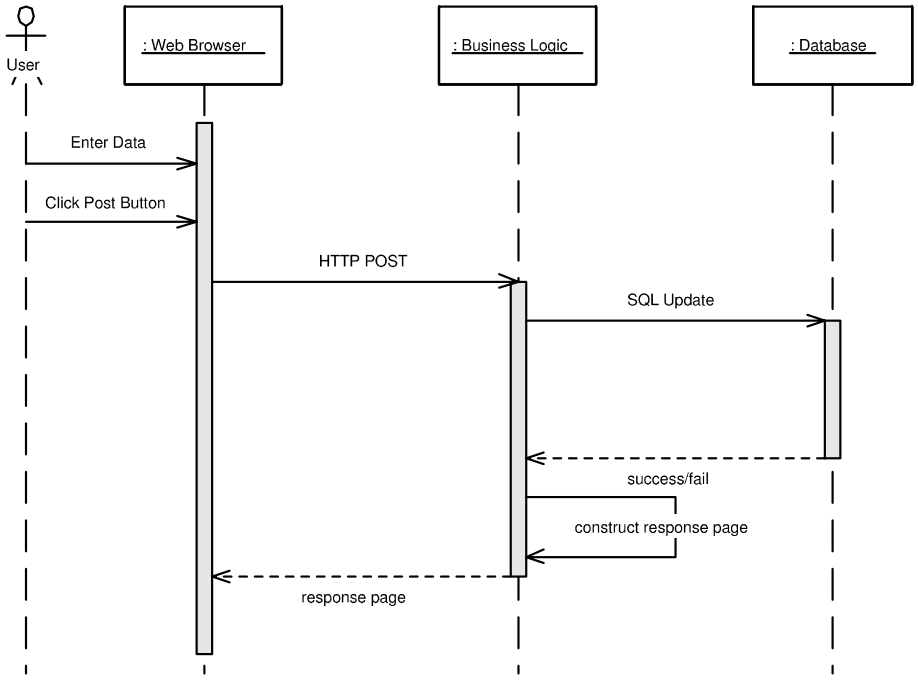

Fig. 6. UML sequence diagram for a simple Web application.

of the system might be expressed using a UML statechart diagram, as shown in Fig. 5.

This diagram shows the behavior of the application whose structure is depicted in Fig. 4. Note that this diagram shows the overall behavior of the application, independent of its components. A diagram that combines aspects of both the system's structure and behavior might be a sequence diagram like that shown in Fig. 6.

This diagram shows one particular interaction among the components in the system – the browser sends a POST request to the business logic component, which updates the database, and returns a results page to the browser, which displays the page.

These diagrams are complementary – each purports to depict the same system from a different perspective. UML does not have a set of consistency rules that can be used to determine whether this is the case, however – stakeholders are responsible for defining what it means for a set of UML diagrams to be consistent and for ultimately making that determination.[5] Additionally, each diagram is ambiguous in different ways. Although the various symbols have basic meanings in UML (e.g., the dashed, open-headed arrow generally represents some sort of dependency, while the rounded rectangles in the statechart represent various system states) their specific meanings in the context of our Web application are not defined within the diagrams. This can be partially rectified through the use of the extension mechanisms mentioned above. For example, a UML profile might define stereotypes to add more detail,

[5] Of course, there have been extensive efforts to explore the issue of UML consistency, including an entire workshop series ("Consistency problems in UML-based Software Development") dedicated to the topic. What has emerged from this line of research has been a wide variety of alternative ways of defining and checking certain types of consistency within and across UML diagrams.

elaborating the meaning of these stereotypes using natural language in a separate document.

4.1.2. UML under the lampposts

From the perspective of our three-lamppost model of ADL interpretation, UML is heavily centered in the business and technology areas. UML has significant ability to model systems from a technical perspective – diagrams such as class and statechart diagrams allow users to express the technical inner-workings of a system in great detail (if desired). However, it should be noted that UML is not a notation rooted in formality, and as such mathematical verification of technical system properties is not generally possible. UML does take into account business needs in a much stronger way than first-generation ADLs. Diagrams such as use case and interaction overview diagrams capture more stakeholder and process-oriented aspects of a system than a first-generation ADL would. Support for domain-specific modeling is mostly accomplished through the use of UML profiles, although profiles are not a panacea: they cannot eliminate ambiguity, nor can they be used to create new kinds of diagrams – they can only specialize existing diagram types.

4.2. AADL

The Architecture Analysis and Design Language (AADL, formerly the Avionics Architecture Description Language) [2] is an ADL for specifying system architectures. While its historical name indicates that its initial purpose was for modeling avionics systems, the notation itself is not specifically bound to that domain – instead, it contains useful constructs and capabilities for modeling a wide variety of embedded and real-time systems such as automotive and medical systems. It is an outgrowth of the first-generation ADL MetaH developed by Honeywell [4], and is now developed collaboratively by a group of industrial and academic organizations. AADL is a Society of Automotive Engineers (SAE) standard, and as such is guided by a larger, more open group than its predecessor MetaH.

AADL can describe the structure of a system as an assembly of components. It has special provisions for describing both hardware and software elements, and the allocation of software components to hardware. It can describe interfaces to those components for both the flow of control and data. It can also capture non-functional aspects of components (such as timing, safety, and reliability attributes). Syntactically, AADL is primarily a textual language, although a graphical visualization and a UML profile for it are under development. The syntax of the language is defined using BNF production rules.

The basic structural element in AADL is the component. AADL components are defined in two parts: a component type and a component implementation. A component type defines the interfaces to a component – how it will interact with the outside world. A component implementation is an instance of a particular component type. There may be many instances of the same component type. The component implementation defines the component's interior – its internal structure and construction. One additional element that affects components is a component's category. AADL defines a number of categories (or kinds) of components; these can be hardware (e.g., memory, device, processor, and bus), software (e.g., data, subprogram, thread, thread group, and process), or composite (e.g., system). The category of a component prescribes what kinds of properties can be specified about a component or component type. For example, a thread may have a period and a deadline, whereas memory may have a read time, a write time, and a word size.

AADL is supported by an increasing base of tools, including a set of open-source plug-ins for the Eclipse software development environment that provide editing support and import/export capabilities through the extensible markup language (XML) [9]. An additional set of plug-ins is available for analyzing various aspects of AADL specifications – for example, whether all the elements are connected appropriately, whether resource usage by the various components exceeds available resources, and whether end-to-end flow latencies exceed available time parameters.

4.2.1. AADL example

To examine how AADL models systems, we present a partial AADL model of a sense-compute-control system. These systems are typical of what might be modeled in AADL. Fig. 7 shows this model of a temperature sensor driver, running on a physical processor connected to a local 33 MHz 32-bit PCI bus.

The first thing to note about this specification is the level of detail at which the architecture is described. A component (sensor_type.temperature) runs on a physical processor (the_sensor_processor), which runs a process (sensor_process_type.one_thread), which in turn contains a single thread of control (sensor_thread), all of which can receive control instructions through an in port (control) and output temperature data through an out port (sensed) over a PCI bus (local_bus_type.pci). Each of these different modeling levels is connected through composition, attachment of ports, and so on. This level of detail emphasizes the importance of tools, such as graphical editors, for modeling this information in a more easily understandable fashion.

The second thing to note is that several of the elements are annotated with specific properties that describe their operation in more detail. For example, the PCI bus transmits 4 bytes (32 bits) of information every 30 ns, and the sensor process runs and samples the temperature every 20 ms. It is these details, tailored for real-time concerns, that make AADL's analysis tool-set possible.

4.2.2. AADL under the lampposts

AADL is heavily steeped in both the technology and domain areas of concern. From a technology perspective

AADL allows detailed, automatically analyzable specifications akin to those that can be created in first-generation ADLs (and, in fact, similar to those created in its predecessor MetaH). It is a high-value, but high-cost notation. The kinds of automated analyses possible with AADL models are powerful, but models of even simple systems are verbose and complex at this level of detail, as the example above suggests. From a domain perspective, AADL is optimized for modeling systems in its target domain – namely embedded, real-time, hardware/software systems. The kinds of constructs and properties that are available are tailored for this purpose. This limits what kinds of viewpoints and concerns can be captured in AADL, but it also helps to focus the language. The increasing realization that no single ADL will suffice for all modeling needs advocates solutions, such as AADL, which have deep support for the set of needs in a particular domain.

```
data sensor_control_data
end sensor_control_data;
data sensor_data
end sensor_data;
bus local_bus_type
end local_bus_type;
bus implementation local_bus_type.pci
properties
  Transmission_Time => 30 ns;
  Allowed_Message_Size => 4 b;
end local_bus_type.pci;
system sensor_type
features
  network : requires bus access
        local_bus_type.pci;
  sensed  : out event data port sensor_data;
  control : in event data port sensor_control_data;
end sensor_type;
system implementation sensor_type.temperature
subcomponents
  the_sensor_processor :
        processor sensor_processor_type;
  the_sensor_process : process
        sensor_process_type.one_thread;
connections
  bus access network -> the_sensor_processor.network;
  event data port sensed ->
        the_sensor_process.sensed;
  event data port control ->
        the_sensor_process.control;
properties
  Actual_Processor_Binding => reference
        the_sensor_processor applies to
        the_sensor_process;
end sensor_type.temperature;
processor sensor_processor_type
features
  network : requires bus access local_bus_type.pci;
end sensor_processor_type;
process sensor_process_type
features
  sensed  : out event data port sensor_data;
  control : in event data port sensor_control_data;
end sensor_process_type;
thread sensor_thread_type
features
  sensed  : out event data port sensor_data;
  control : in event data port sensor_control_data;
properties
  Dispatch_Protocol => periodic;
end sensor_thread_type;
process implementation sensor_process_type.one_thread
subcomponents
  sensor_thread : thread sensor_thread_type;
connections
  event data port sensed -> sensor_thread.sensed;
  event data port control -> sensor_thread.control;
properties
  Dispatch_Protocol => Periodic;
  Period => 20 ms;
end sensor_process_type.one_thread;
```

Fig. 7. Partial model of a sense-compute-control system in AADL.

Fig. 7 (*continued*)

The kinds of analyses that AADL makes possible are driven by business goals. For instance, making quantitative determinations early about an embedded, real-time system (even at high cost) is important because such systems are often safety-critical and expensive to redeploy if an error is found. Nonetheless the language itself does not directly capture business decisions or concerns.

4.3. Koala

Consumer electronics is a dynamic and highly competitive domain of product development. For decades, devices such as televisions and cable descramblers were relatively simple devices with a few, well-defined capabilities. Over the years, these devices have become more and more complex, largely due to enhancements in their embedded software. The latest incarnations of these devices include features such as graphical, menu-driven configuration, on-screen programming guides, video-on-demand, and digital video recording and playback. In a global marketplace, each of these devices must be deployed in multiple regions around the world, and specifically configured for the languages and broadcast standards used in those regions.

The increasing feature counts of consumer electronic devices are accompanied by fierce competition among organizations, and it is just as important to keep costs down as it is to deploy the widest range of features. From a software perspective, keeping costs down can be done in two primary ways: limiting the cost of software development and limiting the resources used by the developed software and thus the costs of hardware needed to support it. Additionally, manufacturers often "multi-source" certain parts. That is, they obtain and use similar parts – chips, boards, tuners, and so on – from multiple vendors, buying from vendors who can offer the part at the right time or the lowest price, and providing a measure of insurance against the failure of one particular part vendor to deliver. If the parts are not completely interchangeable, software can be used to mask the differences.

Product line architectures provide an attractive way to deal with the diversity of devices and configurations found in the consumer electronics domain. Product line architectures allow a single model to express the architecture of multiple systems simultaneously, through the explicit modeling of variation points. Each variation point captures a number of possible alternatives. Products in the product-line are selected by choosing from the alternatives at each variation point.

Philips Electronics has developed an approach called Koala [32] to help them specify and manage their consumer electronics products. Koala is primarily an architecture description language derived from the Darwin ADL [25]. Koala also contains aspects of an architectural style, however, since it prescribes specific patterns and semantics that are applied to the constructs described in the Koala ADL. Koala, like Darwin, is effectively a structural notation: it retains Darwin's concepts of components, interfaces (both provided and required), hierarchical compositions (components with their own internal structures) and links to connect the interfaces. In addition to these basic constructs, Koala has special constructs for supporting product-line variability. Koala is also tightly bound to implementations of embedded components: certain aspects of Koala, such as the method by which it connects required and provided interfaces in code, are specifically designed with implementation strategies in mind, such as static binding through C macros.

Koala's main innovations over Darwin include:

IDL-based interface types: An interface type in Koala is a named set of function signatures, similar to those found in C. For example, the interface to a TV tuner in Koala might be declared like this:

```
interface ITuner{
   void setFrequency(int freqInMhz);
   int getFrequency();
}
```

The ITuner interface type may be provided or required by any number of Koala components. When a provided and a required interface are connected, the provided interface type must provide at least the functions required by the required interface type.

Diversity interfaces: One of the philosophies of Koala is that configuration parameters for a component should not be stored in the component; instead, configuration parameters, including selection of alternatives, should be accessed by the component from an external source when needed. This allows the application to be configured centrally, from a single component or set of components whose purpose is to provide configuration data for the application. "Diversity interfaces" are special required interfaces that are attached to components and are used by each component to get configuration parameters.

Switches: A switch is a new architectural construct that represents a variation point. It allows a required interface to be connected to multiple different provided interfaces. When the variation point is resolved, only one of the connections will actually be present. Which provided–required interface pair is connected depends on a configuration condition. A switch is connected to a diversity interface to get its configuration parameters, just as a component would. Depending on the values returned through the diversity interface, the switch will route calls to one of the required interfaces connected to it. If this means that there will be disconnected components, that is, components that will never be invoked, then Koala will not instantiate these components to save resources.

Optional interfaces: Several components may provide similar, but not identical, services. For example, a basic TV tuner component may have only the ability to change frequencies, but an advanced TV tuner may be able to search for valid frequencies as well. It is possible for callers to a TV tuner to include an optional interface, and query whether this interface is actually connected or not. If it is connected, the caller can make calls on the optional interface; if not, the caller should behave/degrade gracefully.

4.3.1. Koala example

It is easier to understand these constructs and their use with a simple example. Like Darwin, Koala allows simple, non-product-line structural specification. In Fig. 8, we show a very simple part of an architectural model for a television set, a consumer electronics device for which Koala would typically be used.

This architectural model shows two components, an NTSC tuner driver component that receives a television signal on a selected channel and decodes it for display, and a channel changer component that instructs the tuner to change the channel upon a user's request. They communicate through an interface type ITuner, which might be specified as follows:

```
void set_channel(int channel_num);
```

This might be (part of) the software architecture of a single television. Because the tuner driver decodes NTSC signals, this television would be marketed in parts of the world where NTSC signals are used – the United States and Japan, for example. Now, consider the possibility that the architect wants to create a product-line of televisions that might be marketed globally, in areas where television standards are different. Here, the channel changer component might be reused, but the tuner component would have to be changed. The above architecture can be diversified into a product-line architecture using a diversity interface and a switch, as shown in Fig. 9 (adapted from [32]).

Here, a switch is used to select between an NTSC tuner driver (suitable for markets like the US and Japan) and a PAL tuner driver (suitable for markets like Europe). A diversity interface on the channel changer component, combined with a software module M is used to choose which driver component to invoke. The selection is made by a configuration component (not depicted in the figure) connected to the diversity interface. Through these mechanisms, Koala gives architects the power to specify and implement product lines of embedded systems.

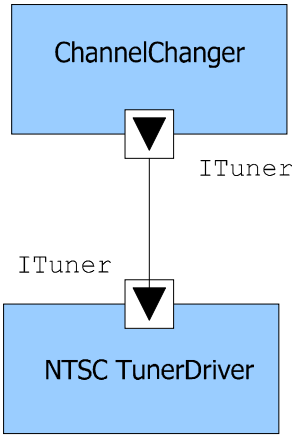

Fig. 8. Simple television architecture for a single product.

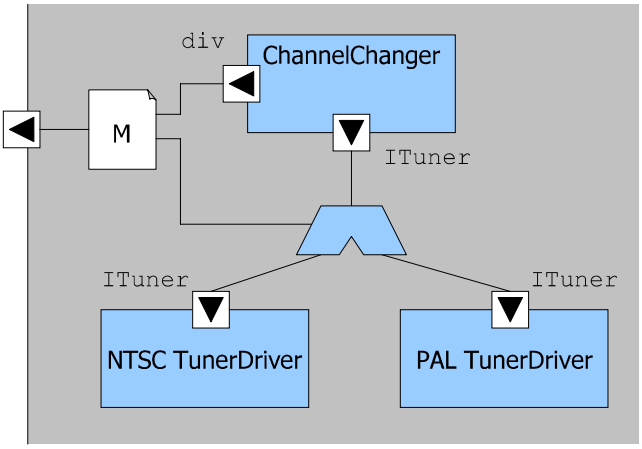

Fig. 9. Product line of two products in Koala.

4.3.2. Koala under the lampposts

Koala draws from all three areas of concern: technology, domain, and business. The explicit configurations and direct mappings to implementations firmly root Koala descriptions in technology. The features provided by Koala are optimized for the modeling of systems in a particular domain – embedded systems in the consumer electronics space. This domain is somewhat different from AADL's, because it lacks the real-time and safety-critical aspects. Although developed for the consumer electronics domain, nothing in Koala prevents its use in other embedded, component-based product lines.

The product-line aspects of Koala are driven primarily by business concerns. Koala allows certain business decisions – the relationship between products – to be specified directly in the language. Other business concerns – for example, reducing time-to-market and reducing costs through increasing reusability – influenced the particular selection of capabilities of the Koala ADL, but are not necessarily expressed directly in Koala models.

4.4. xADL 2.0

Our definition of software architecture characterizes it as a set of principal design decisions about a system. As we stated earlier, whether a particular design decision is principal or not is a function of the system's stakeholders and their needs. Looking beyond the technology lamppost, and taking into account both domain- and business-specific concerns, the diversity of stakeholder needs expands greatly. This makes the development of a single notation that can adequately cover this variety of stakeholder needs unlikely. UML's approach to this problem, as we have discussed, is to provide its users with a large set of often-ambiguous symbols that can be given additional meaning by users through profiles. However, this approach falls short when modeling needs arise that cannot be easily mapped to an existing diagram type, or

when users want to create highly optimized domain-specific notations. Ideally, it would be possible to create optimized notations tailored to individual project needs, but without developing the notations anew for each project.

xADL 2.0 [11] attempts to address these issues by providing a platform upon which modeling features can be defined modularly and reused across projects. New features can be created and added to the language as first-class entities. xADL inherits lessons from many different ADLs: first generation ADLs (namely, C2 [26] and xADL 1.0 [21]), as well as some second-generation concepts such as Koala's product lines. Its major contribution, however, is its support for language extensions. In a way, xADL can be seen as an ADL factory: users can use it to rapidly develop architecture description languages tailored to their domain needs and business goals.

xADL can best be described as a modular ADL. That is, modeling features such as the ability to model a component or a version tree for an artifact are grouped into modules. The xADL language itself is a composition of modules, and xADL users can extend the language by defining and adding in additional modules.

xADL is an XML-based language. Each xADL module is implemented by one XML schema [13]. XML schemas use a data type definition format similar to data structures in an object-oriented programming language. Like classes in an object-oriented programming language, XML schema datatypes can be extended through inheritance. Derived datatypes can be declared in separate schemas. Through this mechanism, datatypes declared in one schema can be extended in a different schema. This means that features defined in one module can be extended in other modules using XML schema inheritance.

Each xADL schema adds a set of features to the language. The constructs in each schema may be new top-level constructs or they may be extensions of constructs in other schemas. Modularizing the feature set in this way has several advantages. First, it allows for incremental adoption – users can use as few or as many features as makes sense for their domain. Second, it allows for divergent extension – users can extend the language in novel, even contradictory ways – to tailor the language for their own purposes. Third, it allows for feature reuse – because feature sets are defined in XML schema modules, schemas can be shared among projects that need common features without each group having to develop their own, probably incompatible, representations for common concepts.

Here, the interplay between syntax and semantics becomes important. XML schemas are largely concerned only with the syntax of a language (or part of a language, when used modularly as in xADL). XML schema does not provide facilities, beyond ad hoc documentation comments, to explain what individual elements mean and how they should be interpreted. As we have seen with UML, a single syntax can be interpreted in myriad ways to suit different purposes. For example, a syntactic element called a 'package' could be interpreted to mean a conceptual grouping of elements, a Java package, or a unit of deployment.

The semantics – the meaning – of xADL's modules and their elements are contained in several different places. At minimum, they can be carried in the minds of stakeholders and passed to new stakeholders by word-of-mouth. A better option is to document the semantics of a given feature in project documentation or in comments in the xADL schemas themselves. An even stronger option is to encode the semantics in tools and visualizations that stakeholders use: tools define how stakeholders interact with a model and can be used to provide additional context and information about the meaning of a feature. Finally, the semantics can be encoded in analysis tools associated with the feature, where errors in interpretation can be identified by automated tools.

As pointed out above, xADL purposefully allows divergent extension. This means that users can add new xADL features that conflict with existing xADL features. Syntactic conflicts will be detected automatically by tools like XML schema validators. Semantic conflicts – for example, two modules expressing the same concept in conflicting ways – cannot be automatically detected. It is up to the architect and other stakeholders to choose or define a set of compatible feature modules, and identify interactions between features.

From time to time, new schemas are added to xADL as they are developed, both by its creators and by outside contributors. Current xADL schemas include features for modeling basic architectural structure (components and connectors) at both design-time and run-time, mappings from architectural elements to their implementations in source code and object code, and product-line features similar to those found in Koala.

Because xADL can be extended with unforeseen constructs and structures in nearly arbitrary ways, it induces challenges that do not exist in languages with stable syntax and semantics, including most of the other ADLs we have discussed thus far. Specifically, parsers, editors, analyzers, and other tools must be developed to cope with a notation whose syntax may change from project to project.

xADL addresses these challenges with an associated set of tools, each of which has specific support for dealing with new schemas. This support may come in the form of automatic adaptation to new schemas, in the case of syntax-directed tools, or guidance and APIs that allow developers to plug in their own support for new schemas in a straightforward manner. These are discussed below.

The xADL Data Binding Library: The xADL data binding library is a library of Java classes that correspond to XML elements and attributes defined in xADL schemas. This library provides an interface through which tools can parse, read, modify, and serialize (write to disk) xADL documents. To support xADL's extensibility, the library

itself is modularized like the xADL language: each xADL schema is mapped to a package of Java classes. Adding new packages extends the library to support new schemas.

Apigen: The data binding library would be of limited use if it had to be manually rewritten each time schemas were added to xADL. Apigen [12] is a data binding library generator: given a set of XML schemas, it can generate a complete new data binding library with support for those new schemas. These new data binding libraries can be used in place of existing ones without affecting applications (except to provide support for the new schemas).

ArchEdit: ArchEdit is a GUI-based, syntax-directed editor for xADL documents. Users navigate the xADL document using a standard tree widget that mirrors the hierarchical structure of the xADL XML document. The editing options available to the user – what sorts of elements can be added and so on – are determined automatically based on the set of xADL schemas in use. When new schemas are added, ArchEdit adapts automatically to support them.

In addition to these tools for supporting extensibility, xADL 2.0 is supported by a collection of tools in the ArchStudio 3 [18] environment, which provide graphical editing and visualization support, automated analysis, product-line specification and selection, and so on. To the extent possible, these tools are also modularized to support xADL extensibility. For example, ArchStudio's graphical editor and analysis framework both allow new additions via plug-ins, such that new visualizations, editing behaviors, and analysis techniques for a new schema can be plugged into the environment easily.

4.4.1. xADL 2.0 example

Returning to our UML example of a three-tier Web application, the structure of the same xADL application might be depicted (in xADL's graphical visualization) as in Fig. 10.

For space reasons, we leave out a complete description of the system in the XML format. However, the Database component might be specified as in Fig. 11.

This very basic specification does not say much about the database component, however. By writing a new xADL schema that extends the definition of a component to better describe databases, the description might look like Fig. 12.

The new schema would extend the definition of a component to add a ⟨datasource⟩ element; the contents of this element would also be defined in the schema. Other properties of this component, and other components, are modeled in xADL 2.0 similarly.

```
<component id="dbComp">
  <description>Database</description>
  <interface id="sql-in">
    <description>SQL</description>
    <direction>in</direction>
  </interface>
</component>
```

Fig. 11. XML specification of a component modeled in xADL 2.0 (some housekeeping data elided).

```
<component id="dbComp">
  <description>Database</description>
  <interface id="sql-in">
    <description>SQL</description>
    <direction>in</direction>
  </interface>
  <datasource>
    <vendor>Oracle Corp.</vendor>
    <location>db.example.com:1234/db1</location>
    <username>webUser</username>
    <password>secret</password>
  </datasource>
</component>
```

Fig. 12. Extended description of the database component.

4.4.2. xADL 2.0 under the lampposts

xADL 2.0 addresses aspects of technology, business, and domain directly. Technology-centric xADL features include the ability to model architectural structure, types, and instances. Business-related schemas include xADL's product-line schemas, which allow xADL to track the evolution of products over time and the relationships of products to one another. xADL has a few domain-specific schemas, such as schemas for modeling the behavior of asynchronous message-based systems; outside users have added their own domain-specific schemas.

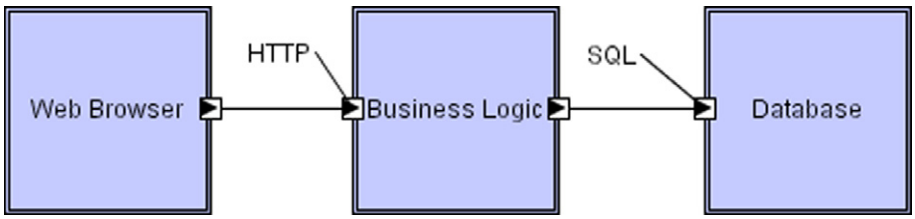

Fig. 10. Graphical visualization of a Web application modeled in xADL 2.0.

xADL's primary contribution, however, is that it addresses all three lampposts at the meta-level. That is, by providing users the ability to rapidly develop their own language extensions and tools, xADL users can customize the language for their own technology, domain, and business goals. In this way, it serves as a kind of factory for second-generation ADLs. It is this facet that renders xADL unique in comparison to all the other notations discussed in this paper.

5. Discussion

Several insights have emerged from our retrospective examination of the evolution of ADLs.

Growing numbers of concerns are being considered part of a system's architecture. First-generation ADLs reflect a relatively narrow view of what constitutes a system's architecture. Many of them focus on modeling structural views of a system, perhaps annotating the structural view with additional properties to capture one or two more concerns, such as behavior or implementation mappings. Second-generation ADLs take a decidedly more holistic approach, capturing a wider variety of concerns that might be of interest to stakeholders. In this sense, architecture modeling is becoming less feature-centric and more stakeholder-centric.

More mature and successful ADLs incorporate concerns rooted in technology, domain, and business needs. First-generation ADLs were largely developed to address technological and theoretical concerns – deadlock freedom, for example. While these capabilities are indeed powerful, they are often not the most critical properties of interest to stakeholders. Domain and business needs, both strong foci of traditional systems engineering practices, are now shaping software architecture description languages.

Multiple views are a necessity. As the scope of architectural concerns grows, unified architectural models that present all information about a system's architecture at once become impractically large. To be cognitively manageable, they must be partitioned into multiple views that show only a subset of concerns at once. The use of multiple views introduces difficulties, such as consistency management among the views, but these difficulties must be overcome to do multi-concern modeling.

No single set of modeling features is sufficient for every project. Even the richest composite notations, such as UML 2.0, will still be inadequate to satisfy every project's modeling needs. In architecture modeling, one size will never fit all. This is not to say that general-purpose notations are not worthwhile. In fact, they have certain attractive advantages, many of which derive from economies of scale. A general-purpose notation can attract more users, and therefore will likely be better validated, have more tool support from vendors, and have increased utility as a communication medium among stakeholders. However, such a notation can never be as expressive or highly optimized as a domain-specific notation.

Extensibility is a key property of modeling notations. This follows from the earlier insights above: if general-purpose notations are useful but insufficient, and architectural concerns vary across domains and projects, a natural and effective solution is the use of extensible modeling notations. Extensible notations provide a basic, general purpose foundation for architectural modeling along with mechanisms that allow stakeholders to specialize the notation for their particular technology, domain, and business needs.

Tools are as important as notations. The main power of a notation comes not through its syntax or even its semantics, but the tools that can be used to operate on the notation. All of the second-generation ADLs we surveyed are supported by a variety of software tools and environments, for editing, visualization, analysis, creating extensions, and so on. Arguably, good tool support from major vendors was a driving force behind the widespread adoption of UML. Conversely, lack of good tool support can doom an otherwise excellent ADL to obscurity – something that we would argue was at least in part the case with several first-generation ADLs.

5.1. The relationship between methodologies and ADLs

In this paper, we have looked at ADLs in terms of how well the languages themselves support modeling concerns from under the domain, business, and technology lampposts. The activity of developing and evolving a successful architecture, however, does not stand or fall on modeling notations alone. Instead, a wide variety of factors has to be considered, including how those notations are used in developing an architecture. A well-constructed ADL combined with a poor process will often result in a poorly defined architecture. For an ADL like UML, which does not enforce precise semantics or any particular method of use, processes and methodologies become critically important to developing high-quality architectures.

ADLs themselves often imply one or more methods for developing models in that ADL. For example, an iterative process of modeling, language extension, and tool extension is favorable for xADL, while methods like top-down and bottom-up design might be favorable for AADL. These, however, are "micro-processes" that fit within larger processes that govern software development throughout the lifecycle. ADLs can be integrated into these broader processes as well; perhaps the best examples we have seen are development processes that have been created with UML in mind. For example, many phases of the Rational Unified Process (RUP) [22] leverage UML models as inputs and outputs, and many efforts to implement Model-Driven Architecture (MDA) [31] processes use UML as a primary modeling notation.

At this point, synergy between the development of software development processes and architecture description languages is just beginning to occur. As ADLs expand beyond the boundaries of the technology lamppost, we

can expect that architecture-centric processes, some optimized for particular ADLs, will more substantially emerge in the future.

6. Conclusions and future trends

Our initial classification and comparison of ADLs [27] has been a useful reference point to researchers and practitioners, and has stood the test of time in many ways. All the same, it was a very technologically oriented study that largely ignored the other two lampposts. There are two clear reasons for this. The first is that ADLs themselves (and, it could be argued, even software architecture) were not sufficiently well understood at the time. The second reason behind the limited scope of our original study is that no first-generation ADLs extensively supported architectural concerns beyond the technology lamppost.

This paper has presented a new perspective on architecture and architecture description languages, which spans the three lampposts. We believe that this perspective provides a basis for a much more thorough treatment of ADLs. However, we are not so naïve as to think that this particular study will "close the book" on our collective understanding of software architectures and ADLs. In fact, having looked at both the first- and the second-generation ADLs through the prism of the three lampposts, a natural question to ask ourselves is: will there be a "third generation" of ADLs?

Certainly, even the best current architecture description notations leave a lot to be desired, so notations will undoubtedly continue to be developed and evolved. At the least, we should expect to see continued advances in the fundamentals of architecture modeling; it is unlikely that, for example, an ideal formal semantics will emerge that provides the basis for all architecture analysis activities in the future. Researchers will undoubtedly continue to investigate the best ways to detect deadlock, capture product-lines, and make judgments about software qualities, all based on architecture models.

If there is to be a major leap forward resulting in a third generation of ADLs, it will likely emerge from an even deeper confluence between high-level domain and business concerns of systems engineering and lower-level technological concerns derived from software engineering. For example, few ADLs today take into account concerns such as human organizations, costs, risks, and processes, while consideration of these factors is a key aspect of systems engineering. It seems likely that future architecture modeling approaches will begin to incorporate these concerns.

An additional area in which ADLs will likely see improvement is in their expansion to other lifecycle activities. Architectural modeling is still very much design-centric. Second-generation ADLs generally have some support for tracing architectural design to other lifecycle activities – SysML's requirements view, or xADL 2.0's implementation mappings, for example. As architecture modeling notations mature, we should expect to see even stronger traceability to activities such as requirements, implementation, testing, maintenance, evolution, and so on. Such developments are needed to fully realize the central role that we feel architecture should play in software development.

Acknowledgements

This material is based upon work sponsored in part by the National Science Foundation under Grant Nos. CCR-9985441, ITR-0312780, CCF-0430066, and CNS-0438996. The content of the information does not necessarily reflect the position or the policy of the Government or any sponsor and no official endorsement should be inferred. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors also wish to express their appreciation to the numerous helpful comments and suggestions made by the anonymous reviewers.

References

[1] R. Allen. A Formal Approach to Software Architecture. PhD thesis. Carnegie Mellon University, 1997, p. 248. Available from: <http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-144.pdf>.

[2] R. Allen, S. Vestal, B. Lewis, D. Cornhill, Using an architecture description language for quantitative analysis of real-time systems, in: Proceedings of the Third International Workshop on Software and Performance, ACM Press, Rome, Italy, 2002, pp. 203–210. Available from: <http://portal.acm.org/citation.cfm?doid=584369.584399>.

[3] D. Batory, L. Coglianese, S. Shafer, W. Tracz, The ADAGE Avionics Reference Architecture, in: Proceedings of the AIAA Computing in Aerospace-10 Conference, March, 1995.

[4] P. Binns, M. Englehart, M. Jackson, S. Vestal, Domain-specific software architectures for guidance, navigation and control, International Journal of Software Engineering and Knowledge Engineering 6 (2) (1996) 201–227.

[5] B. Boehm, C. Abts, W. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, B. Steece, Software Cost Estimation with COCOMO II, Prentice Hall, New Jersey, 2000.

[6] G. Booch, Object-Oriented Analysis and Design with Applications, Benjamin-Cummings, 1993.

[7] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide. Object Technology Series. Addison Wesley Professional, Reading, Massachusetts, 1998.

[8] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, second ed., Addison-Wesley Object Technology Series, Addison-Wesley Professional Reading, Massachusetts, 2005.

[9] T. Bray, J. Paoli, C.M. Sperberg-McQueen, Extensible Markup Language (XML): Part I. Syntax. World Wide Web Consortium, Recommendation Report, February, 1998. Available from: <http://www.w3.org/TR/1998/REC-xml>.

[10] Carnegie Mellon University. How Do You Define Software Architecture? <http://www.sei.cmu.edu/architecture/definitions.html>, Software Engineering Institute, Webpage, 2005.

[11] E. Dashofy, A.v.d. Hoek, R.N. Taylor, A comprehensive approach for the development of XML-based software architecture description languages, Transactions on Software Engineering Methodology (TOSEM) 14 (2) (2005) 199–245.

[12] E.M. Dashofy. Issues in Generating Data Bindings for an XML Schema-Based Language, in: Proceedings of the Workshop on XML

Technologies in Software Engineering (XSE 2001), Toronto, Canada, May 15, 2001.
[13] D.C. Fallside. XML Schema Part 0: Primer. World Wide Web Consortium, W3C Recommendation Report, May 2, 2001. Available from: <http://www.w3.org/TR/xmlschema-0/>.
[14] P.H. Feiler, B. Lewis, S. Vestal, The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering, in: Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems, Washington, D.C., May, 2003.
[15] D. Garlan, R.T. Monroe, D. Wile, ACME: An Architecture Description Interchange Language, in: Proceedings of the CASCON '97, IBM Center for Advanced Studies, Toronto, Ontario, Canada, November, 1997, pp. 169–183. Available from: <http://www-2.cs.cmu.edu/afs/cs/project/able/ftp/acme-cascon97/acme-cascon97.pdf>.
[16] M.M. Gorlick, R.R. Razouk, Using Weaves for Software Construction and Analysis, in: Proceedings of the 13th International Conference on Software Engineering. May, 1991, pp. 23–34.
[17] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.
[18] Institute for Software Research. ArchStudio, An Architecture-based Development Environment. Available from: <http://www.isr.uci.edu/projects/archstudio/>, University of California, Irvine.
[19] P. Inverardi, A.L. Wolf, Formal specification and analysis of software architectures using the chemical abstract machine model, IEEE Transactions on Software Engineering 21 (4) (1995) 373–386. Available from: <http://citeseer.nj.nec.com/inverardi95formal.html>.
[20] I. Jacobson. Object-Oriented Software Engineering: A Use Case Driven Approach, first ed., Addison-Wesley Professional, 1992, 552 p.
[21] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, R.N. Taylor, xADL: Enabling Architecture-Centric Tool Integration with XML, in: Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34), Software mini-track, Maui, Hawaii, January 3–6, 2001.
[22] P. Kruchten. The Rational Unified Process: An Introduction, third ed., 2003, 320 p.
[23] D.C. Luckham, J. Vera, An event-based architecture definition language, IEEE Transactions on Software Engineering 21 (9) (1995) 717–734.
[24] D.C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events, in: Proceedings of the DIMACS Partial Order Methods Workshop IV, Princeton University, July, 1996.
[25] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying Distributed Software Architectures, in: Proceedings of the 5th European Software Engineering Conference (ESEC 95), 989, Springer, Berlin, 1995, pp. 137–153.
[26] N. Medvidovic, P. Oreizy, J.E. Robbins, R.N. Taylor, Using Object-Oriented Typing to Support Architectural Design in the C2 Style, in: Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering. ACM SIGSOFT, San Francisco, CA, October, 1996, pp. 24–32. Available from: <http://www.isr.uci.edu/architecture/papers/ADL-FSE96.pdf>.
[27] N. Medvidovic, R.N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93, Reprinted in Rational Developer Network: Seminal Papers on Software Architecture. Rational Software Corporation, available from: <http://www.rational.net/>, 2001.
[28] N. Medvidovic, D.S. Rosenblum, D. Redmiles, J. Robbins, Modeling Software Architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology 11 (1) (2002) 2–57.
[29] S.J. Mellor, M.J. Balcer, Executable UML: A Foundation for Model Driven Architecture, first ed., Addison-Wesley Professional, 2002.
[30] R. Melton. The Aesop System: A Tutorial. The ABLE Project, School of Computer Science, Carnegie Mellon University, HTML. Available from: <http://www-2.cs.cmu.edu/afs/cs/project/able/www/aesop/html/tutorial/aesop-demo.html>.
[31] J. Mukerji, J. Miller (Eds.), MDA Guide Version 1.0.1. Object Management Group, 2003.
[32] R.v. Ommering, F.v.d. Linden, J. Kramer, J. Magee, The Koala Component Model for Consumer Electronics Software, IEEE Computer 33 (3) (2000) 78–85.
[33] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52. Available from: <http://citeseer.nj.nec.com/perry92foundation.html>.
[34] R. Prieto-Diaz, J.M. Neighbors, Module interconnection languages, Journal of Systems and Software 6 (4) (1986) 307–334. Available from: <http://www.cs.jmu.edu/users/prietorx/RubenPubs/publications/MILpaperPC.doc>.
[35] M. Moriconi, R.A. Riemenschneider, Introduction to SADL 1.0, A Language for Specifying Software Architecture Hierarchies. Report SRI-CSL-97-01, 1997. <http://www.sdl.sri.com/papers/csl-97-01>.
[36] J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, W. Premerlani, Object-Oriented Modeling and Design, first ed., Prentice Hall, 1990.
[37] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, Abstractions for software architecture and tools to support them, IEEE Transactions on Software Engineering 21 (4) (1995) 314–335.
[38] SysML Partners. Systems Modeling Language (SysML) Specification version 0.9. Report, January 10, 2005, p. 270. Available from: <http://www.sysml.org/artifacts/spec/SysML-v0.9-PDF-050110R1.pdf>.
[39] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, D.L. Dubrow, A component- and message-based architectural style for GUI software, IEEE Transactions on Software Engineering 22 (6) (1996) 390–406.
[40] Telecommunication Standardization Sector of ITU. Specification and Description Language (SDL). Report ITU Standard Z.100, 2002. Available from: <http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf>.
[41] W. Tracz. LILEANNA: A Parameterized Programming Language, in: Proceedings of the Second International Workshop on Software Reuse, 1993, pp. 66–78.
[42] J.B. Warmer, A.G. Kleppe, The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 1998.