This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier’s archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright
A type-centric framework for specifying heterogeneous, large-scale, component-oriented, architectures

Georg Jung a,∗, John Hatcliff b

a Universität Potsdam, Institut für Informatik, 14482, Potsdam, Germany
b Kansas State University, Computing and Information Sciences, United States

ABSTRACT

Maintaining integrity and consistency, and effecting conformance in architectures of large-scale systems require specification and enforcement of many different forms of structural constraints. While type systems have proved effective for enforcing structural constraints in programs and data structures, most architectural modeling frameworks include only weak notions of typing or rely on first order logic constraint languages that have steep learning curves associated with them and that become unwieldy when scaling to large systems.

We present the CADENA Architecture Language with Meta-modeling (CALM) — that uses multi-level type systems to specify and enforce a variety of architectural constraints relevant to the development of large-scale component-based systems. CADENA is a robust and extensible tool that has been used to specify a number of industrial strength component models and applied in multiple industrial research projects on model-driven development and software product lines.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Developing and maintaining long-lived, large, distributed, information and computation systems involves a number of challenges. Overall system functionality must be carefully decomposed and arranged into a modular architecture with precisely negotiated interfaces and a clear, hierarchical, organization. Requirements from multiple stake-holders have to be systematically reconciled while incrementally adding concerns to the system architecture. These requirements compete in various dimensions: They come from separate domains of expertise (e.g., hardware interfacing, network, application logic), different levels of abstraction (e.g., supervision, team management, implementation), individual stages of development (e.g., integration of legacy-code, new implementation, α- or β-stages), and so on. Additionally, the process of reconciliation continually grows more complex throughout the system evolution. Finally, architecture models have to be useful, accurate, and robust. Usefulness requires for example that the architecture adapts to refinement, globally as well as in detail, and that architectural elements have a concise representation. Aspects of accuracy include that the architectural elements faithfully reflect capabilities of the system’s execution environment, and that they cater to the abstractions used by various
domain experts. Further, to be robust the architectural abstractions have to support overall system integrity throughout the specification and refinement effort.

The architectural integrity and internal consistency of long-lived, large-scale, projects face many threats, starting with initial design and continuing throughout the system life-cycle. Industrial experience reports indicate a serious need for tools and processes that (a) enable concise, rigorous specification of architectural constraints and (b) provide mechanical checking of conformance to architecture constraints and ensure consistency between various architecture aspects [1, pp. 477–478]. This is even more the case in the context of a product-line approach, where the degree of cost savings is directly tied to the ability to constrain and impose discipline on architecture elements to increase their potential for reuse over multiple related projects.

At the programming language level, type systems have proven to be a very effective paradigm for enforcing constraints on interaction of system units (e.g., class/method types must be compatible with their use), for ensuring that data structures conform to certain structural invariants (e.g., tree shaped, list shaped), and for characterizing requirements for converting data between different formats. Previous work on architectural definition languages (ADL) and meta-modeling frameworks (frameworks for creating domain-specific modeling languages and environments) has made significant strides toward supporting higher-level architecture development tasks involving specification of architecture units (e.g., components and subsystems), composition of those units, and interactions between units. Nevertheless, many existing ADLs use weak type systems and incorporate only limited forms of type checking. Some existing frameworks that have been designed for architecture exchange [2,3] defer type checking to other tools or provide external constraint languages [4,5] based on first order logic that, while powerful, are sometimes difficult for engineers to understand, require verbose definitions to capture simple forms of type checking, and become unwieldy and hard to manage as systems scale. Finally, existing ADLs often fail to support several important capabilities needed for large-scale system development including the ability to

(a) specify domain or platform specific languages for building open-ended collections of component and interface types,
(b) incorporate multiple component models within a single system, as it is often needed when multiple systems are integrated to form a “system of systems”, or for describing multiple levels of abstraction within a system,
(c) specify relationships between architectural layers in multi-layered systems, and
(d) flexibly combine and extend architectures as system development unfolds.

In this article, we introduce the CADENA Architecture Language with Meta-modeling (CALM), a type-centric framework for rigorous meta-modeling and architecture definition of component-oriented systems. CALM enables rapid specification and scalable checking of many common forms of architectural constraints that occur in the context of large-scale system development.

The specific contributions of our work are as follows.

• We describe how CALM can faithfully capture industrial component models, component middleware platform capabilities, and domain-specific component modeling languages in rigorous, mechanically leverageable, meta-models. These meta-models can be incrementally refined to describe the evolving of architectural platforms, to capture architectures on various levels of detail, or to encompass new or more precise abstractions. Variations of the meta-models can, for example, be arranged in inheritance hierarchies or flexibly combined into models of complex hybrid-platforms.

• We summarize the foundations of CALM’s multi-tiered type system and explain

(a) how the type system enables architects to concisely specify important notions of architecture consistency, structural constraints, and topological (connectivity) requirements in an intuitive manner; and

(b) how associated type checking enforces compliance of system architecture elements to type-based constraints, domain-specific modeling languages, and platform descriptions captured via typed meta-models in an efficient, scalable, way.

Within this effort we outline CALM’s novel concepts of abstracting and typing whole networks of components as higher-level architectural elements with respect to their completeness/incompleteness (i.e., connectivity requirements/potential) to provide a rigorous approach to incremental refactoring, transitions in granularity, and subsystems as implementations of architectural elements.

• We illustrate how the CALM meta-models can be used to address a number of challenges in large-scale architecture development. These challenges include modeling of heterogeneous systems that contain various component platform models on different layers within the same system, using type-based coercions to model capabilities for integrating different frameworks, and capturing complex system layering and subsystem abstractions in which elements from one domain are embedded inside of elements from a different one. Among those, dual to the component abstraction, we introduce the connectors as service abstractions which – beyond middleware capabilities – can capture architectural conventions and strategies.

CALM concepts are implemented in an IBM Eclipse-based framework called CADENA – a robust and extensible environment for modeling and development of component-based systems which is freely available for download [6]. CADENA bridges the gap between meta-models and concrete architectures and enables automatic propagation of design changes throughout the model, providing a selfadapting development environment.

While the generality and expressiveness of CALM has been demonstrated by using it to capture a number of realistic component models (Section 7), in this paper we illustrate some of the principles of CALM using a system phrased in terms
of a hybrid component model which integrates three architectural styles, including a style for nesC — a component model
and associated infrastructure that has been widely used for building wireless sensor networks [7].

The current version of C aden a has been partially funded and used by Lockheed Martin Advanced Technology Laboratory
(ATL). Their aim was to evaluate the effectiveness of advanced architecture tools as part of their internally funded Software
Technology Initiative. This initiative seeks to develop innovative technologies for tackling challenges in large-scale system
design and integration.

The rest of the paper is organized as follows. Section 2 summarizes the principles of CALM, Section 3 introduces an
example to illustrate our approach, Section 4 describes architecture modeling in CALM, Section 5 presents higher-level
architecture manipulations, Section 6 provides an overview over related work, Section 7 briefly evaluates our approach, and
Section 8 offers some conclusions and outlines future work.

2. Principles of CALM

Following previous work on ADLs (see, e.g., [8,9] for an overview), CALM’s modeling primitives are based on the four
fundamental categories of entities that define every component-based system: components — loci of computation, interfaces
— loci of interaction, connectors — loci of communication, and configurations in which instances of components, interfaces,
and connectors are allocated and connected together to form what has been termed a component assembly or system
scenario. Existing ADLs tend to organize the definition and use of these elements using two modeling tiers: in the upper
tier, developers define component, interface, and connector types, and in the lower tier instances of the declared types are
allocated and connected to form component assemblies. Both the upper and lower tiers in existing frameworks (especially
those designed for architectural exchange [2,3]) tend to be very unconstrained since they seek to allow a variety of different
component structures to be embedded in them.

In order to more effectively specify and enforce structural constraints, CALM restructures the two-tiered approach to
provide three modeling tiers named style, module, and scenario — where each tier constrains and guides activities in the
tier below (Fig. 1). The style tier is a meta-modeling tier that allows architects to define ADLs constrained to a particular
component model or application domain. In particular, CALM styles specify a collection of type schemas that give rise to
languages of types for building component, interface, and connector types in the module tier below. This approach allows
architects to precisely capture the capabilities and type systems available in existing component middleware frameworks
like CCM, EJB, etc., and to define domain-specific component modeling frameworks. Using the language defined by the style
tier, the module and scenario tier provide the two stages of traditional ADLs, i.e., defining the elements of the architecture
(module) and instantiating and combining them into assemblies (scenario). Within these two lower levels, the distinction
from previous work is that checking inherent in the C aden a implementation guarantees that types declared at the module
level conform to an associated style and that instances at the scenario level conform to types. Moreover, the type framework
simplifies development by providing palettes and modeling commands tailored to the associated style and module types.
While related facilities have been provided in other meta-modeling tools such as GME [10], C aden a provides a variety
of additional, richer, mechanisms for guaranteeing conformance to meta-model and type definitions. Finally, providing a
separate style meta-modeling tier enables a collection of novel capabilities that go well beyond simply defining languages
of types. Because CALM styles are manipulable artifacts, architects can operate on and define relationships between styles
to capture structuring principles relevant for architectural modeling of large-scale systems as illustrated in Section 5.
Altogether, the different activities suggested by CALM’s three-tiered modeling concept align with the three specific developer roles as distinguished for example in [1] (chp. 6, p. 316ff). A platform designer or product-line architect lays out an infrastructure (consisting of, e.g., execution environment, communication services, persistence layer, etc.). The concrete implementation of this infrastructure with its standardized services forms a base layer for large, distributed, applications, which is called the component middleware. The middleware is the main shaping aspect of an architectural style, as it implies requirements and options for business-logic components that rely on its infrastructure. Component developers implement business-logic components adhering to the infrastructure requirements and making use of infrastructure services. Finally, system integrators develop applications by combining available components on the infrastructure platform. Fig. 1 illustrates the relations between the development activities with respect to the CALM modeling tiers.

CALM’s three-tiered modeling approach is inspired by type theory [11] in which type systems are organized into three levels – values/instances conform to types, and types conform to kinds – and CALM adopts the kinds/types/instances terminology for naming modeling elements in each of its three layers. Fig. 2 summarizes the notions of type checking and structural constraint enforcement that are enabled by our approach. We will describe these capabilities in detail in the remainder of the paper, referred to as Capability (a) through (i) (cap. (a)–(i)).

### 3. Example of a multi-layer architecture

Fig. 3 presents an example that we use throughout the article to illustrate various concepts of CALM. A sensor bank consisting of some number of sensors is connected through a local acquisition network to a controller which in turn is linked to a monitor base-station (Fig. 3(a)). On a lower level of abstraction, the link between the sensor bank and the monitor is established through a radio network link (Fig. 3(b)). The radio link contains a timer component which is implemented in terms of a nested component assembly (Fig. 3(c)). Both the Radio Network Link and Timer are built using the nesC sensor network infrastructure [7,12] and the standard nesC iconography is used in the diagrams of nesC components. Finally, the transmission of the data over a wireless hardware radio link (HWRadioLink) is implemented using a collection of power-controlled, dynamic frequency, phase-key or FM-modulated, hardware components (Fig. 3(d)).

Although simple, this example includes several characteristics which we believe are intrinsic to architectures of realistic, large-scale systems. We summarize these characteristics and explain how modeling and development of such systems are constrained and guided using the primary typing capabilities of CALM listed in Fig. 2.

- The system includes subsystems built using one or more existing component frameworks (nesC in this case). Therefore, modeling for these subsystems needs to be constrained to ensure that capabilities of component framework middleware infrastructure are accurately reflected in architecture specifications (cap. (d)). For example, proper capture of nesC capabilities guides developers in building types and component instances that conform to nesC (cap. (a) and (c)). Proper capture also ensures accurate mappings between model/specs and code, which means it helps in ensuring that tool infrastructure teams can implement plug-ins that auto-generate code skeletons and deployment “glue code”, and import existing nesC libraries into the modeling framework.
- The system includes multiple architecture styles within the same system, as often required in constructing “systems of systems” that incorporate new, legacy, and off-the-shelf systems built using multiple component frameworks. The architecture styles employed in the example are
(a) the high-level planning style (Fig. 3(a)),

<table>
<thead>
<tr>
<th>Typing Principle</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Instances conform to component types</td>
<td>a component type serves as a template from which a set of component instances can be generated; changes in a component type propagate to all instances</td>
</tr>
<tr>
<td>(b) Typed interfaces on ports</td>
<td>establishes basic notion of protocol/contract on component interaction points</td>
</tr>
<tr>
<td>(c) Type-correct port/interface/role connections</td>
<td>guarantees type compatibility between components</td>
</tr>
<tr>
<td>(d) Types conform to kinds (styles)</td>
<td>guarantees component types and instances conform to vocabulary specified by the style; enables precise specification of component models used in underlying component middleware frameworks and guarantees features of models match capabilities of underlying middleware; enables precise specification of domain-specific component modeling languages</td>
</tr>
<tr>
<td>(e) Architecture style inheritance</td>
<td>enables incremental construction/refinement of architectural styles and platform descriptions</td>
</tr>
<tr>
<td>(f) Incremental typing of component networks</td>
<td>captures if a network of components is completely connected, and if not, summarizes the potential for connection corresponding to all unconnected ports; enables a type-checked compositional approach to component assembly and assembly nesting</td>
</tr>
<tr>
<td>(g) Orthogonal nesting of component networks in components and connectors</td>
<td>enables component assemblies to be abstracted either as components or connectors; introduces the possibility of a service abstraction for reusable assemblies and strategies</td>
</tr>
<tr>
<td>(h) Typed-based multi-style nesting constraints</td>
<td>guarantees proper layering of multiple architectural styles within the same system; prohibits developers from violating layering constraints</td>
</tr>
<tr>
<td>(i) Inter-style coercions</td>
<td>captures as formal abstractions the necessary conversions between different platforms and component models in “system of systems” construction</td>
</tr>
</tbody>
</table>
Architectures specifications need to capture coercions that represent data conversions and marshalling/unmarshalling that often must be implemented to communicate between different component frameworks (cap. (i)).

- The architecture is layered, and developers must adhere to these layering constraints (e.g., the implementation of each planning layer component must be expressed in nesc and each nesC component associated with the hardware category must be described using the physical layer style) to avoid the architecture degrading over time (cap. (h)).
- Encapsulation is used as an abstraction mechanism for both components and connectors that may involve a change in architecture styles to represent an abstraction boundary. For example, the HWClock timeout-generator is encapsulated inside of the Timer component of the RadioNetworkLink, and the RadioNetworkLink is encapsulated (with a style change) inside of the Controller/Monitor link of the planning assembly (cap. (g)).
- While nesC components initially all have the same shape, in the example it is reasonable to distinguish them according to their possible contents. For example the LinkControl or the SendQueue components are software implementations, while the Timer represents a sub-scenario, and the HWRadioLink abstracts an assembly on the physical network layer. We use refinement to enhance the existing nesC style to reflect these differences (cap. (e)).

4. Typed modeling in CALM/Cadena

4.1. Environment creation through meta-modeling

As pointed out in Section 2, the modeling/development effort in CALM is organized in three tiers that correspond to the three commonly distinguished development roles, namely the product-line architect (style tier), the component developer (module tier), and the system integrator (scenario tier). The fundamental principle of this tiered structure is that the
definitions and declarations on the higher tier serve to create a typed modeling language that can then be employed on the lower tier. Concretely, the terms declared on the style tier denote keywords and entity-kinds that can be applied to defining types on the module tier and to instantiate automatically typed entities on the scenario tier. Likewise, the types defined on the module tier are available for instantiation on the scenario tier.

The language created on the style tier is meant to provide primitives that faithfully reflect capabilities of a component platform. Such capabilities include for example communication and message channels, execution environment, and platform services. The primitives that abstract these capabilities in CALM are the entity-kinds: A set of declared names that denote either interfaces, connectors, or components. Following the concept of providing a language, these declared kind-names are then available as kinds, but also as keywords to create types. For example, if the execution environment, platform services, and connectivity possibilities of a specific middleware allow for a certain component kind called “bean” (as, e.g., in EJB), then the component kind bean should be defined with the respective properties in a CALM EJB style. Using this style, a component developer can then declare bean-types in his module that correspond to available beans and reflect their specific connectivity constraints, for example a bean-type access-bean. On the scenario level then, a specific instance of access-bean can be created, for example DB-access. Then DB-access is an instance of the bean-type access-bean, and it is of the sort component. Thus, the CALM style tier creates a platform specific modeling language. In doing so CALM can employ the customary terminology of the respective platform.

The creation of the set of kinds that abstract a specific platform is central to the CALM modeling concept. A new kind is made from two entities:

- A name that designates the kind, and
- a shape, given through a so-called meta-kind.

The meta-kind is a CALM primitive that can be outfitted with a set of properties desired for the kind. It can be incrementally defined, altered, and mixed and merged with other meta-kinds. Once the properties of a meta-kind faithfully abstract the capabilities of the desired kind, the kind can be fixed or exported as a named shape. Note that therefore the creation of a kind in CALM is a two-stage process, which becomes useful if a platform allows for multiple kinds that share a substantial set of properties but differ in minor aspects. This two-stage process also simplifies some technical details should the need arise to alter the CALM meta-model (see Section 4.2).

For an intuition of the function of the style tier to the lower tiers, consider the triangle of reference or semiotic triangle by Ogden and Richards [13]. It connects a symbol (e.g., a kind-name, declared on the style tier) to a referent (e.g., a set of types of that kind, declared on the module tier), using a reference or definition, that describes the desired properties of the kind and connects it to the types. Fig. 4 shows an adapted semiotic triangle, there the position of the symbol (term) and reference (definition) are swapped (grey arrow) to emphasize the relation between the symbol and its referent (term and object). Note that, if the object of such a relation is itself a term (e.g., a name declared for a lower tier), the triangles can be stacked, in principle allowing for infinitely many tiers of meta-modeling. Nevertheless, since CALM explicitly targets component-oriented architectures, the given three modeling tiers combined with the elementary concepts of interfaces, components, and connectors (present in CALM as keywords/primitives), prove sufficient for effective modeling and development.

The Cadena tool-suite supports the task of creating middleware-platform-specific modeling languages through incremental consistency checking based on typing. Whenever a new kind is introduced in a style, it becomes available as a primitive to the lower tiers, whenever a type is defined in a module, it can be instantiated in a scenario. Similarly, alterations to or deletion of existing kinds/types is propagated automatically through the tiers. Therefore, together with the tool support, a CALM style not only defines a modeling language, but also adapts the integrated modeling environment of Cadena.

1 With the tool Cadena, all modeling/development is graphical or form-based, nevertheless for easy illustration, we use a simple language that expresses the CALM concepts in the absence of the tool.
4.2. The CALM meta-model

Fig. 5 displays the principal entities of CALM and their interrelations in an informal meta-model. As mentioned above, there are three central, independent, concepts: The component meta-kind, the interface meta-kind, and the connector meta-kind. Each of these entities can have any number of attributes. Attributes in CALM only exist as contents of a meta-kind, kind, type, or instance (they have to be declared on the meta-kind level). They come with their own, extensible, type system and can be used to model a host of structural aspects, as well as functional data or meta-information (attributes within an interface meta-kind, e.g., might include operation or method signatures; attributes within a component might include version information, authorship, technical data, etc.). Next to attributes, a component meta-kind can have any number of port options as contents, while a connector meta-kind can have any number of role options. These options constrain the form of ports/roles allowed on components and connectors. Each port option and each role option references exactly one interface meta-kind. This association (as opposed to the meta-kind-attribute and meta-kind-port/role option relation) is not containment, but reference (i.e., the interface meta-kinds exist outside of the port/role options). Further, port options and role options each feature a multiplicity that indicates the number of ports/roles a component/connector can have due to this option. Likewise, the multiplexity constrains the possible fan-in/fan-out of ports within this option.

Each of the three basic meta-kinds can be used to define kinds, and subsequently types and instances through the CALM modeling tiers, always with the object of enriching the precision of their interrelations. Fig. 6 illustrates the development of these interrelations from the meta-kind to the instance level on the example of the relations between component and interface. A component meta-kind contains some number of port options, each associated with one interface meta-kind. The corresponding port option on a component kind derived from the meta-kind is therefore associated with an interface kind which derives from the respective interface meta-kind. When declaring types within the component kind, concrete ports are declared according to the multiplicity constraints of the port options. Each such port is associated with an interface type.
which lives within the interface kind of the respective port option. Finally, concrete component instances are created from the types, which are again through their ports associated with interface instances taken from the respective types. The actual number of interface instances depends on the port parity and multiplexity.

4.3. Example outline

We now illustrate how CALM’s three tiers are employed to define a modeling language for the nesC component framework used in the example of Section 3 and how that modeling language is subsequently applied to define types and instances of nesC components.

Fig. 7 visualizes the genesis of model elements through the three tiers of CALM, as it will be built throughout the following sections (Sections 4.4–4.6). Starting with the meta-kinds and kinds on the style tier, entities are created through the types on the module tier down to concrete values or instances on the scenario tier.

4.4. The style tier

Fig. 8 shows a possible CALM style specifying the nesC component model in a syntax specifically designed for easy presentation on paper, in absence of the CADENA tool-suite (the CADENA tool-suite relies completely on graphic and form-based input instead, as shown, e.g., in Fig. 9). To capture the types that are to be available to developers programming in nesC, lines 2–19 define a few of the nesC platform types (since it is for example purposes only, the list is not comprehensive). These types can be used for attributes throughout the modeling tiers. Lines 21–38 define kinds that represent languages (schemas) of types that can be used to build the nesC interface, component, and connector model entities. Kinds are defined in a two-stage process. An architect first uses CALM meta-kinds (using meta-interface, metacomponent, and metacommector) to construct the basic building blocks for the modeling entities. Meta-kinds can be extended through inheritance to facilitate reuse. Once the construction for a particular class of entities is finished, the architect declares an associated kind from a meta-kind, which exposes the type language for use by developers at the module level.

As an example for the construction of a kind within a CALM style note the definition/declaration of the interface meta-kind mNesCInterface (defined on l. 21–22) and the interface kind nesCInterface (declared on l. 23). The (simplified) nesC ADL which is modeled by the style in Fig. 8 features one kind of interface, which in turn consists of an arbitrary number of events (i.e., asynchronous messages from the provider of an interface to the user), and commands (i.e., messages from the user of an interface to the provider). Lines 21–22 thus define the interface meta-kind mNesCInterface containing a list of nesC-operations. Line 22 illustrates the use of CALM’s attributes to model the operations (events and commands) of the interface. As described in Section 4.2, attributes can be associated with each of the CALM architectural entities to capture the platform types of the targeted middleware or execution environment as well as meta-data types such as organizational data, deployment information, or physical units (e.g., SI-units) not covered by the platform types. In the example, the attribute operations (l. 22) captures the ability to define a list of operation signatures built using the nesC platform types declared earlier (l. 14–19). CALM attributes can be labeled with binding times indicating that the attribute should be bound to a value either at the style level, module level, or the scenario level. In this case, the attribute operations is defined to be module level (keyword MODULE, l. 22), i.e., it describes a property of a type within this kind, in contrast to style-level

---

2 The original, non-simplified, nesC style has two more kinds of interfaces besides the “bundle”: bare command- and bare event-interfaces. They can be omitted without loss of expressiveness.
attributes which define properties of the whole kind, or scenario-level attributes which define specifics of an instance. Finally, line 23 declares the actual modeling entity, the interface kind, by attaching the name `nesCInterface` to the previously defined meta-kind `mNesCInterface`. This kind allows developers at the module level to build interface types that must conform to the structure of meta-interface specification `mNesCInterface`. It is possible to declare multiple, distinct, kinds from a single meta-kind. Since the declaration of a named kind from a meta-kind makes that kind available to the module level, it is called exposing a kind. The exposed kinds of a style make up theconstype language of models within that style.

Analogous to the interface kinds exposed from interface meta-kinds, component kinds can be declared in CALM from component meta-kinds. The NesC ADL provides one kind of component, called `module` (not to be confused with the CALM module tier). A `nesC` module can provide or use an arbitrary number of interfaces as its ports. As described in Section 4.2, the possibility of building component types with certain categories of ports is modeled through defining `port options` in the component (meta-) kind (l. 26–27). Like the declared (exposed) kinds of architectural elements define a language for building types, the port options within a component (meta-) kind define a language for declaring ports on a component type.

A port option starts with a parity, either `provides` or `uses`, to indicate whether or not the interface represents a service that the component provides or a service that the component needs to connect to (i.e., a context dependency). After the parity, an integer interval, the `multiplicity`, constrains how many ports within the respective option any component of that kind must have. The first port option of the component meta-kind `mNesCModule` which models the “module” of the NesC ADL (l. 26) defines a minimum of zero ports and no maximum ([0..*]). This means, that any component type of a kind derived from `mNesCModule` is allowed to feature an arbitrary number of ports that comply to this port option. The third position in the port option is the module-level keyword, which – in accord with the notion of creating a language – is used by developers at the module tier to indicate the particular category of ports being declared. The standard `nesC` keywords defined in the port options in lines 26 and 27 (`provides` and `uses`) coincide with the CALM parities (`provides` and `uses`), but are conceptually unrelated. Thus, the first port option (l. 26) specifies the use of the keyword “provides” to declare ports according to this port option on a component type derived from this meta-kind. Position four specifies the meta-kind from which interfaces associated with this port can be drawn, in this case `mNesCInterface`. The last position of the port option gives the `multiplexity`, which constrains the number of connections that can be made to ports within this option, i.e., the range of minimum and maximum fan-out. Fig. 10 shows the distinction between multiplicity (Fig. 10(a), min/max number of ports) and multiplexity (Fig. 10(b), min/max possible fan-out per port). In summary, a port option looks
as follows\textsuperscript{3}:

\[
\langle \text{parity} \rangle \langle \text{multiplicity} \rangle \langle \text{module-level keyword} \rangle : \langle \text{interface meta-kind} \rangle \langle \text{multiplexity} \rangle
\]

When exporting the nesCModule component kind from the mNesCModule meta-kind (l. 28–30), the architect must specify a particular interface kind for each interface meta-kind declared in the port options of the associated component meta-kind. In this case, the nesCInterface kind is specified for both port options. Thus, due to this definition/declaration of the component kind nesCModule, the module tier of CALM, now allows to declare nesCModule-types containing arbitrary many “provides” ports (first port option) and arbitrary many “uses” ports (second port option) each featuring a nesCInterface-type as interface and allowing arbitrary fan-out.

The one service featured in nesC is a one-to-one communication service called wire which connects interfaces of identical type. Analogous to the port options on component (meta-) kinds, connector (meta-) kinds have role options which specify the ability to define connection points associated with particular kinds of interfaces. Lines 33–34 declare role options for the mNesCWire connector meta-kind. In contrast to the port options seen earlier, the multiplicity and multiplexity of both role options are set to [1], shorthand for the interval [1..1], meaning that connectors conforming to this meta-kind can

\textsuperscript{3} Note that the concrete syntax used in this paper is not authoritative (and hence not overly elaborated), as Cadena, the implementation of the CALM concepts, relies on form-based, graphical, and wizard guided input/output.
only have one connection point for each of its two roles (i.e., the connector is binary) and that the fan-out value for each connection point is constrained to be one. CALM allows different compatibility requirements to be stated for component interfaces that communicate through a connector. In addition, type variables can be introduced to achieve a notion of polymorphism. For example, to export the \texttt{nesCWire} from the meta-kind \texttt{mNesCWire}, a type variable \texttt{a} for types within the kind \texttt{nesCInterface} is declared (l. 36). Type variables such as \texttt{a} enable CALM to express constraints about relations between types such as equality (=), and sub- or super-typing (\texttt{\geq, \leq}). In this simple case, in which the wire can only connect interfaces of what would be equal type in the CALM model, the variable \texttt{a} is used twice, denoting equality of the types. Therefore, the role \texttt{provider\_side} and the role \texttt{user\_side} can both be associated with the same type of \texttt{nesCInterface}.

The style defines now three kinds,

- the interface kind \texttt{nesCInterface} with the attribute \texttt{operations} modeling interfaces of the NesC ADL,
- the component kind \texttt{nesCModule} with two port options modeling modules of the NesC ADL which can contain any number of provided ports (first port option) and any number of used ports (second port option), each of them featuring \texttt{nesCInterface}-types, and
- the connector kind \texttt{nesCWire} with two role options, each featuring the same \texttt{nesCInterface}-type, modeling connections of the NesC ADL.

It serves to define a language where \texttt{nesCInterface}, \texttt{nesCModule}, \texttt{nesCWire}, \texttt{uses}, etc., are keywords that can be used to specify and declare types of NesC architectural elements.

4.5. The module tier

Having defined type schemas (via CALM (meta-) kinds) for NesC types, developers work at the module tier to build up libraries of types, and \textsc{cadena} tool support guarantees that these types conform to schemas declared at the style tier.

Fig. 11 shows excerpts from a CALM module, declaring types in the \texttt{nesC} style from Fig. 8. A CALM module declares types for interface kinds (e.g., l. 40–43) and for component kinds (e.g., l. 48–70). Connector types are not declared on the module tier; the style-level typing constraints enable connector types to be created on-the-fly when instantiated on the scenario tier. Again, this strategy emphasizes the interpretation of connectors as service entities and as part of the infrastructure.

Note for example the \texttt{nesCModule}-type \texttt{LinkControl} (l. 48–55). The type is declared with the kind-name \texttt{nesCModule} from the style, ports on the type are declared with the names of the port options they comply to, namely \texttt{provides} and \texttt{uses}, featuring \texttt{nesCInterface}-types previously declared (l. 40–46). The module-level attribute \texttt{operations} of the \texttt{nesCInterface}-kind is valuated to declare \texttt{nesCInterface}-types (l. 2–9 and following, 40–43 and following).
Fig. 12 shows the module editor in CADENA, with a module within the nesC style defined by Fig. 9. Analogous to the CALM textual form, the CADENA module tier declares types of component and interface kinds. To support the module developer, the editor automatically adapts to the architectural style. The highlighted NesComponent type Checkpoint is displayed in the Outline area (bottom-left), with the provided interfaces on the left and the used interfaces on the right. As an example for the context-sensitive (i.e., style adapted) editing, the context menu for adding a port is open in the screenshot. Note that the menu lists the port option names of the nesC style to add ports. Also, the component and interface icons are custom defined for the given style. Similar style adapted editing exists for the scenario tier in CADENA.

Declaring the types of an architecture means declaring the building blocks or the functional units. While nesC itself does not distinguish clearly between type and instance, CALM models always contain the type layer and each type can be instantiated multiple times.

4.6. The scenario tier

Fig. 13 shows the assembly of the nesC hardware clock wrapper outlined in Fig. 3(c) (Section 3). In Line 2 the nesCModule-type TimerControl is instantiated to obtain the value control. Naturally, the provided interfaces of that type, i.e., StdControl and Timer are instantiated with the component. A nesCWire with unnamed type is instantiated in lines 5–6 and connects the two nesCModule instances with the appropriate ports.

5. Style manipulation and combination

With the architectural style being a separate, manipulable, part of the modeling framework, CALM allows various operations on styles which go beyond architectural exchange and directly address problems of architecture refinement.
and combination of domain-specific layers in multiple dimensions. Fig. 14 gives a conceptual visualization of three different ways to modify and interrelate styles, related to Capability (e) and (f) (Fig. 14(a), Fig. 14(b)) and Capability (g) through (i) (Fig. 14(c)). This section overview how emphasizing type-based meta-modeling enables architects to easily combine and refine styles using multiple inheritance. Working at the style tier, architects also (a) define relationships between styles, e.g., defining precisely where coercions are available to connect interfaces from different styles and (b) specify layering/abstraction constraints. Larger examples and associated formalization can be found in [14].

5.1. Building hybrid styles

Fig. 14(a) illustrates how hybrid styles (e.g., representing different platforms cooperating on the same abstraction level) can be formed through CALM style manipulations: a union of two styles is formed (via multiple inheritance), and then “bridging elements” (i.e., component or connector kinds whose ports/roles associate interface kinds drawn from both styles) are introduced. This capability enables system architects to formally capture the practice of integrating components from different sources with related yet dissimilar context requirements (e.g., Bonobo and CORBA components within the Linux Desktop), and allows for a smooth transition of existing scenarios into hybrid environments.

5.2. Refining styles

CALM style manipulations allow architects to create modeling environments that control the evolution or refinement of an architecture from a general purpose platform independent description (as might be used in earlier stages of development) to a style which contains more specifics about the underlying platform. For example, Fig. 14(b) illustrates a situation where the architect specifies a style A in which developers initially work. The style is constrained to be platform independent by the absence of any kinds that describe the capabilities of the specific platform. This prevents rogue developers from “running ahead” of the development plan by adding additional details that might threaten the genericity or portability of the system description. Once further details of a target platform are identified, the architect refines the style and leads the migration of existing models into the refined style in three steps.

- First, he creates a sub-style A’ which adds new, more specific kinds that capture, for example, particular communication services available on the target platform as new connector kinds.
- Second, he directs developers to change generic capabilities of style A to specific implementation options exposed in style A’. Developers carry out these tasks in style A’ which contains modules and scenarios which are not yet completely migrated (the style supports both the original generic elements as well as newly added platform specific elements).
- Third, as the migration phase nears completion, the architect creates a new sub-style A’’ derived from A’. In A’’ he removes the original platform independent kinds (through CALM’s kind elision), so that only the refined, platform dependent kinds remain (i.e., the generic elements from A which were still present in A’ are no longer supported in A’’).
It can often be quite difficult to tell via manual inspection if all generic modeling elements from style A have been replaced by the more specific elements added to obtain style A’ (this is especially true in large-scale development). However, instead of manual inspection, the confirmation of a completed migration is obtained by type checking the migrated scenarios in style A’ which no longer supports the generic elements. This same three-step process can be repeated multiple times, forming a succession of validated development “check points” moving from platform independent to increasingly platform specific architectures in a controlled sequence of style refinements.

Fig. 15 refines the original nesC style by introducing three new kinds, and at the same time eliding the nesCModule kind. This new style will be used to connect nesC to other styles by distinguishing the components with respect to their possible implementation contents. The new component kinds nesCSoftModule (for software-implemented components), nesCHWModule (for hardware wrappers), and nesCNWModule (for network infrastructure wrappers) are not distinguished by the nesC definition, nevertheless the architect can further tailor a component model like nesC to a particular development context. For example, these kinds are useful to establish layering and nesting constraints such that only modules and scenarios from the hardware style can be nested in components that are built from the nesCHWModule kind (Section 5.3.2). In [15] the possibilities for model abstraction, specialization, migration, and hybrid construction given through style manipulation are discussed in more detail.

5.3. Implementation–abstraction relations on styles

5.3.1. Typing assemblies

While various notions of typing for components, interfaces, and connectors at modeling layers analogous to CALM’s module and scenario tiers have been considered in earlier works, we are not aware of existing approaches for typing assemblies of allocated components as units where the assembly type serves as a summary of how the assembly can interact with its context. CALM includes a notion of typed assembly that captures the overall connection potential in terms of may/must modalities (cap. (f)). In CALM, a port/role is called open, if its multiplexity allows further connections (may connect), otherwise it is called closed. A port/role is called complete, if its multiplexity does not require any further connection, otherwise it is called incomplete (must connect). Intuitively, completeness requirements of ports/roles constrain the minimum of further necessary connections in a scenario, while openness indicates the maximum of further possible connections. CALM calls the set of possible types which fall into these minimum/maximum constraints the type spectrum of the scenario. In other words, type checking of the assembly not only tells the developer about the compatibility of component/connector connections, it also indicates whether additional connections are possible/required. Those pending connections can be presented in a tool task list along with their corresponding interface types, so that developers can easily identify remaining development steps. For each such type, CADENA presents a summary of type-correct connection opportunities (automatically filtering out incompatible types), and this serves to rapidly focus the developer’s attention on appropriate connections.

As an example, consider the scenario timer_assembly in Fig. 13, Section 4. It features two component instances control of type TimerControl, and hwClock of type Clock. Associated with these components, the scenario contains three provided interfaces, control.init of type StdControl, control.timer of type Timer, and hwClock.clock of type Clock, and one used interface, control.clock of type Clock, i.e., the assembly has four ports. Also, it contains one unnamed instance of a nesCWire-kind connector, through which it features two roles (Fig. 16(a), note that nesC displays single connectors sometimes as “bundles” of lines).

By their multiplexity, all roles in the scenario are closed and complete, because their minimum and maximum fan-out of one connection is used. The ports on the other hand are complete but open (i.e., their multiplexity invariably is [0...*], they can connect with an arbitrary number of further connectors). Thus, as a maximum, this assembly can be typed as a component with all four aforementioned ports exposed, while as a minimum, no ports or roles have to be exposed. If within this spectrum, the two ports control.init and control.timer (Fig. 16(b)) are chosen to be exposed, the resulting type of the assembly is equal to the declared nesCModule type Timer (Fig. 11, Section 4).

Alternatively, the type spectrum can be changed by adding two nesCWire-kind connectors attached to the previously exposed ports control.init and control.timer with the respective user_side role left open (Fig. 16(c)). As a result,
the two unconnected roles become minimum members of the type spectrum, and the assembly can now be typed as a connector by exposing only the two open roles, and hiding the complete ports. Unary connectors such as this one have turned out to be an elegant means to abstract services provided by library functions in nesC such as the timer.

In summary, this notion of typing provides a sound approach to specifying when an assembly can be abstracted as a component or a connector (cap. (g)). If all the roles in an assembly are complete (i.e., all connection responsibilities of connectors are fulfilled), then it can be typed as a component with an interface corresponding to the open ports of the assembly. If the ports are complete, the assembly can be typed as a connector, featuring only the open roles. Further, not all ports of a component-typed assembly (or roles of a connector typed assembly) have to be visible in the abstraction; only incomplete ones have to be mentioned.

Abstracting an assembly into a component or connector is accomplished using CALM’s wrapping facility. For example, the wrapping into a component as described above (Fig. 16(b)) is accomplished through

```java
implementation HWTimer :
    wrap timer_assembly into sensornet.Timer {
        expose Init = control.Init;
        expose timer = control.timer;
    }
```

Cadena records such wraps in implementation tables, so that when instantiating a component the architect can choose a fitting implementation, which might include simulation stubs if a component is to be placed in a testing environment. Fig. 17 shows a screenshot of Cadena with the Timer-assembly associated with the Timer-component in RadioNetworkLink. While
other frameworks also include the notion of nesting (almost always limited to just nesting in components), the novelties here are that (a) built-in typing guarantees the well-formedness of the nesting, and that (b) our distinction of open/closed roles and ports enables the principle to be applied orthogonally to nesting in components and connectors. In addition, our assembly types provide a foundation for typing component architectural patterns in which assemblies are parameterized via assembly-typed placeholders/variables into which other assemblies of compatible type can be substituted.

5.3.2. Style coercions and nested styles

A key feature of CALM is its ability to specify when different styles can be used in the same system model at different levels of abstraction to allow capturing such relations of Fig. 14(c) (cap. (h)–(i)). Fig. 18 shows two revisions of a conceptual style as used for the highest-level assembly of this paper’s example (Fig. 3(a), Section 3). As stated, many elements of this architecture are implemented in nesC at deeper levels of nesting, i.e., the global style abstracts nesC, and in turn nesC implements global. Specifically, the connector representing the communication link between the Controller and the Main Monitor is realized by a non-trivial nesC assembly (Fig. 19) in which an instance of the RadioNetworkLink (Fig. 3(b)) is used at each end of the connector.

CALM enables the architect to capture implementation/abstraction relationships such as the one described here with specifications on each modeling tier: First, the style level defines a coercion between the global_b style and nesC:

```
coelection nesC_to_global :
from nesC build global_b.Plug {
  provides [1..*] available : nesCInterface
  uses [1..*] required : nesCInterface ;
}
```

A coercion specifies that interfaces from an abstract overview style (global_b) can be expressed in terms of a set of interfaces from the implementing style, in this case an arbitrary number of elements from nesCInterface are specified as a valid implementation of an element of the Plug interface kind. A coercion always describes provided interfaces, i.e., a provided interface of the kind Plug can internally provide interfaces of kind nesCInterface declared with the available keyword introduced in line 3, and use nesCInterface-kind interfaces introduced with required. For a Plug port or role with parity uses the internal parities are reversed.

On the type level (module tier), type conversions can be declared with the vocabulary defined by the coercion:

```
nesC_to_global sensornet_to_bank :
abstract sensornet into sensorbank.Send {
  available send : Send,
  available control : StdControl ;
}
```

Fig. 18. Global style (base & refined).

Fig. 19. Network assembly as a connector.
This conversion packs one \textit{sensornet.Send} interface type and one \textit{sensornet.StdControl} interface type into the interface type \textit{sensorbank.Send}. This enables the architect to wrap the network assembly into a connector of the global style analogous to the single-style wraps explained in Section 5.3.1 (cap. (g) and (i)).

5.3.3. Multi-dimensional layering

Fig. 20 interprets the implementation/abstraction relation between CALM styles as positive layering constraints, in the sense that a style \textit{A} can access the functionality of a style \textit{B} iff \textit{A}’s architectural elements can provide a shell for assemblies of \textit{B} through wrapping, which is enabled through respective coercions/conversions of interface kinds/types. In this specific case, architectural elements of the \textit{global_b} style (Fig. 20(a)) are implemented in the \textit{nesC} style which represents the layer below the global assembly. The \textit{nesC} style in turn has elements implemented by the \textit{radioComm} style (Fig. 20(b) and (c)), and other elements implemented in a nested assembly in \textit{nesC} (Fig. 20(d)).

These conceptual layers, given through specific styles, correspond to the suggested lower (media-) layers of the ISO/OSI stack [16]. The absence of coercions between the styles of the network layer and the physical layer prevents direct access in circumvention of the data-link layer. This use of component encapsulation provides a correct-by-construction approach to layered architectures, as well as the ability to introduce layers in multiple directions (i.e., specific to individual components).

5.3.4. The implementation table

Hiding functionality inside monolithic architectural entities per se only adds a quantitative improvement to the concepts of architectural development by reducing the number of elements which are visible at one time and thereby clarifying complex system assemblies. To add a qualitative improvement, nesting has to blend with the black-box notion of the component-oriented paradigm, which means that the implementations associated with the architectural elements have to be easily exchangeable. Therefore, CALM separates the topology from the specification of the components’ and connectors’ internals. This separation is expressed in the distinction between the terms \textit{assembly} and \textit{scenario}. The term assembly is used for the topology, whereas the scenario is described as an assembly with additional (meta-) data. There can be multiple scenarios for every assembly, given that multiple implementations are possible for each architectural element. The actual implementations within a scenario, both “atomic” implementations or sub-assemblies, are recorded in the \textit{implementation table}.

To support incremental supply of the meta-data, CALM distinguishes different stages of completeness of a scenario. A scenario is said to describe a \textit{system}, iff all multiplicity constraints of the underlying assembly are met (i.e., no connections have to be added), otherwise it is said to be a \textit{subsystem}. Further, a scenario is said to be \textit{complete}, iff every shell of the underlying assembly is assigned an implementation. If instead some elements of the assembly are only represented by a shell with no further information, the scenario is called \textit{incomplete}. The completeness of a scenario only considers one level, which means that some implementations might be done in terms of sub-scenarios which in turn are still incomplete. Therefore, scenarios are distinguished as being either \textit{abstract}, which means that they might be still incomplete at some level of nesting, or \textit{deployable}, which is the equivalent of recursive completeness.

This suggests two strategies of assigning sub-scenarios (as opposed to atomic implementations) as implementations to architectural elements (Fig. 21). The first strategy is to record \textit{assemblies} as implementations. Having chosen an assembly for a given element, an implementation sheet has to be chosen for that assembly in a recursive step. This method can intuitively be thought of as a \textit{top-down} approach (or \textit{branching, call-by-name, lazy}) since first the global assembly is outfitted with implementation data, and then recursively the nested assemblies are handled. Fig. 21(a) illustrates this method, following the nested specification of the grey elements. A deployable (sub-) system in this strategy has to feature a tree of implementation sheets.

The second strategy is to record \textit{scenarios} in the implementation table. This approach can be thought of as \textit{bottom-up} specification (or \textit{linear, call-by-value, eager}), since when choosing an implementation for an element of a scenario, all recursive implementations are already given. Instead of a tree of implementation sheets, the single implementation sheet on the top level describes the deployable (sub-) scenario (Fig. 21(b)).
Note that, if incomplete scenarios (or incomplete implementation tables) are allowed in either of the two strategies, they become very similar in their practical aspects (i.e., the bottom-up strategy would no longer mean that sub-scenarios are completely specified when they are included into a table). CALM favors a bottom-up strategy with possibly incomplete scenarios because it allows the software architect to concisely specify (incomplete) multi-level topologies which manifest the concept of reference architectures.

5.4. Abstracting strategies and conventions

The ability to easily tailor the modeling language and development environment towards very specific platform requirements by introducing, eliding, and/or manipulating architectural elements (including connectors) and freely attaching implementations (constrained by the type system) enables CALM and CADENA to offer a novel way of incorporating coding strategies into the architectural style.

To illustrate, consider the PRiSM component model for real-time aviation systems developed by the Boeing company within their Bold Stroke project, modeled in earlier versions of CADENA (e.g., [17]). PRiSM features a common publish-subscribe event middleware with additional synchronous remote method-call capability on a real-time platform. Fig. 22(a) shows a simple example of a two-way steering system. One strategy used in this system is called control-push–data-pull: whenever new data is generated by a sensing device (e.g., a GPS), the device publishes an asynchronous event announcing the data. Upon reception of the event notification, subscribing components use the synchronous connection to read the data from the sensing component. This strategy is common in a real-time setting, since it ensures that the receiving component never blocks while waiting for new data to be available. Yet it means that many virtual connections actually involve two concrete connectors, the event-push and the data-pull connector. The example in Fig. 22(a) contains four such double connections, one (between GPS and airframe components) is highlighted.

The CALM model of PRiSM introduces a new connector into the system (the push–pull connector) which can be implemented by such a pair. The benefit lies not only in the reduction of lines in the graphic model, but mainly in making “crossed” control-data connections (i.e., where event source and data source are not the same) impossible. Since the implementation of such a push–pull connector can be expressed as an assembly of existing middleware services, the new, more abstract, model still faithfully captures the capabilities of the middleware platform. In Fig. 22(b), the highlighted single connection between GPS and airframe corresponds to the two highlighted connections of Fig. 22(a).5

5 Note that CADENA does automatic layout. Since the added push–pull connector allows for fewer connections (in the example scenario, a total of four paired connections are replaced by single connections), the placements in Fig. 22(b) differ significantly from those in Fig. 22(a).
6. Related work

6.1. Architectural typing

In [18], Medvidovic, Rosenblum, and Taylor present a type theory for architectures based on notions from programming languages (specifically object-oriented languages). A focus of their work is the development of a subtyping relationship for concrete architecture elements (in this work they do not discuss any sort of meta-modeling/style tier) based on the concepts of name compatibility, interface conformance, behavioral equality, and implementation conformance, together with the derived notions behavioral conformance and strictly monotone subtyping. As architectural elements, the work considers components, connectors, and configurations (i.e., assembly definitions), in contrast to CALM which considers interfaces to be separate, type-able, entities, and develops a distinct typing methodology for assemblies. As a result of the work, Medvidovic, Rosenblum, and Taylor propose the use of multiple (sub-)typing relationships, noting the complexity of an architectural setting in comparison to programming languages. While the work presents a sophisticated and useful solution for the subtyping problem, unlike CALM it does not use typing and in particular not kinding to enforce platform requirements or to incorporate domain-specific knowledge.

6.2. Architectural styles

Abowd, Allen, and Garlan [19] proposed the notion of architectural styles to capture the environment vocabulary of a software configuration by providing component and connector types, structural constraints, and (optionally) a semantic
model [20]. Di Nitto and Rosenblum investigate ADL suitability for modeling component systems, noting the need for support of architectural styles and style refinement [21]. Of ADLs evaluated they found only Acme/Armani [3,4] satisfactorily supporting style refinement for modeling middleware compliant software through Acme family extensions. Nevertheless, an Acme family is simply an enumeration of types (at the level of a CALM module) that form the “palette” from which instances can be drawn to represent a particular style of architecture. There is no higher-level typing mechanism such as Capability (d) in Acme to enforce that the types of an Acme family conform to particular constraints on structure or that new types added to the enumeration are aligned with capabilities of a particular execution environment. For example, the Acme user manual notes that “Typically, a family also embodies a set of rules that specify design rules that constrain how designs can be pieced together and declare certain ‘well-formedness’ rules. However, the Acme type model is actually quite weak, which places a burden on someone defining the family to include either language descriptions about these assumptions, or to specify the constraints in some form that can be interpreted by a tool (e.g., Armani)” [22]. Thus, when style constraints are to be enforced, they must be specified and checked by a mechanism external to the type system — with the suggested approach being to use Armani’s first order logic (FOL) constraint language. While Acme does support Capability (a), even basic capabilities like (b) and (c) must be specified in FOL.

CALM goes beyond the notion of families by introducing a separate meta-modeling tier which captures the architectural style in a mechanically leverageable way, thereby defining precisely what can appear within a style and what cannot. While this type-based CALM meta-modeling tier does not provide the same expressive power as first order logic, it is much easier to use, more scalable, and it directly captures most common component system capabilities and structural constraints. We argue that the complexity of first order constraint languages limits the accessibility to developers, making constraints more difficult to specify, maintain, and evolve, while typing on the other hand, being a familiar concept to engineers, seamlessly integrates into development processes and scales easily. We believe that first order constraint languages are necessary, but they should only be applied after simpler, more directly integrated, notions of typing are applied. Also, CALM emphasizes the distinction between the meta-modeling tier and the typing and instantiation levels to provide an environment for manipulation, combination, evolution, and cooperation of styles (cap. (e)–(i)). These capabilities are not supported in Acme and supporting them in any constraint framework based on FOL seems difficult.

Like Acme, xADL 2.0 has been designed as architecture exchange language, but seeks greater robustness and flexibility by using a framework of XML schemas [2]. xADL provides basic notions of component and interface types (cap. (b)) but type checking is deferred to other tools. Using xADL XSD schemas [23], all type definitions must reside within the element xArchTypes, and instances of these types model a run-time system under element xArchInstance. However, xADL 2.0 provides no way to strictly separate a platform definition (style) from libraries of design-time types. Using xADL 2.0 it is possible to define a platform vocabulary through a set of types collected under element xArchTypes, then provide a schema extension, extending elements representing platform kinds to arrive at a library of types, but this only yields a two-tiered capability similar to that of Acme again, with any conformance checking deferred to other tools. While there is value in a tool engineering approach that enables separate tools to provide constraint enforcement, we wish to pursue a research agenda that enables exploitation of the benefits and synergy that result from directly integrating a variety of forms of typing into the modeling framework itself.

6.3. Meta-modeling

The Generic Modeling Environment (GME) is a powerful framework supporting graphic definition of domain-specific modeling languages and the capability to generate domain-specific graphic modeling environments [10,24]. GME allows users to define a meta-model paradigm using an extended UML class diagram notation with constraints written in OCL. Based on a meta-model paradigm, GME generates a domain-specific modeling environment with entities defined in the paradigm available to graphically construct domain-specific models. GME paradigm definitions like CoSMIC [25] may be written to capture platform component, connector, and interface templates, but some basic notions of type checking (cap. (c)) must be accomplished through OCL constraints. GME offers some support for composing paradigms from elements defined in existing paradigm definitions [26] (cap. (e)) but provides no direct support for Capability (f)–(i).

6.4. Analysis on architectures

In [27], Edwards et al. propose an extensible architecture analysis framework, in which the architecture (i.e., the topology of architectural elements) is clearly distinguished from a separate model interpretation infrastructure which utilizes flexible extensions to the architecture to perform all kinds of highly specialized analysis. They argue that the separation of the abstract component technology (ACT) from the model interpretation framework (MIF) combines the benefits of general purpose modeling languages and domain-specific modeling languages as it, next to other benefits, allows the user to run various low-level analyses on an otherwise general and reusable model. The approach is implemented in the XTEAM tool-chain, which derives its ACT part from a GME meta-modeling paradigm (Section 6.3). With their emphasis on flexibly incorporating analyses, their attribute extension mechanism is more thoroughly worked out than its analogue in CADENA, where the emphasis is stronger on the ACT side and on the topological constraints than on the MIF part. Both tools could have an interesting synergy.
The Architecture Analysis & Design Language (AADL) [28–30], a modeling framework based on MetaH [31] which is mainly used in aviation industry, follows a more direct but less flexible approach towards incorporating analyses into architecture design. AADL does not include an architectural meta-layer, but incorporates extensions through so-called annexes. The analysis capabilities are introduced through specialized modeling elements with fixed semantics which each abstract a concrete set of resources. The approach is very powerful with respect to the variety of analyses which can be performed, nevertheless the need to add specialized modeling elements through annexes makes it hard to perform new analyses on existing models. AADL offers a very basic component typing capability (cap. (a)), and component types are further grouped with respect to the modeling element they belong to (comparable to the kinds). These groupings entail predefined nesting constraints, but in contrast to CALM (cap. (h)) these constraints cannot be changed or refined.

6.5. Industrial tools

In current industrial practice, initial design is hampered by the inadequacies of existing commercial tools that are almost exclusively UML-based. These tools focus on lower-level class architectures and provide few mechanisms for establishing higher-level architectural subsystem and layering constraints that can guide system architects. Notions of components and interfaces introduced into UML 2.0 are an initial step in addressing these concerns, but tools like Rhapsody and Rational Modeler only provide limited aspects of even the most basic typing capabilities (cap. (a)–(c)). Moreover, the generality of UML does not easily allow the architecture vocabulary to be tailored to the concerns of expert designers from different domains working at different layers within the system (cap. (d)–(h)) nor are mechanisms provided for formalizing data and interface conversions that mitigate the inconsistencies and ambiguities that arise as experts from different domains seek simultaneously to express their concerns within the architecture (cap. (i)).

6.6. Other frameworks

In summary, previous work has provided significant insights and innovations that have inspired our work, and space constraints do not allow a detailed comparison to each of these (for a survey of ADLs, see [8,9]). For example, [32] emphasized notions from object-oriented (OO) type systems to describe interesting concepts of component type refinement in the C2 ADL. While C2 supports a particular class of architectures (layered message passing systems) and confines type descriptions to a modeling tier analogous to CALM’s module tier, CALM emphasizes use of typing including OO concepts such as inheritance in meta-modeling facilities (at the CALM style tier) that allow one to describe any number of component model styles, style refinement, combinations of styles, nesting relationships between styles.

We seek to complement and add value to previous work by emphasizing a variety of forms of typing to enforce structural constraints. There are other important notions that are orthogonal and can be combined with our approach. Behavioral descriptions [33] and dynamic reconfiguration mechanisms [34] can be added to support specifications of interaction protocols between components. Notations for specifying variability points and variations for software product lines (e.g., as in xADL) can easily be incorporated. In fact, we believe that the strong typing capabilities of CALM are very important for supporting a product-line approach in which variants are plugged into a reference architecture at component and subsystem variability points: CALM’s typing at these points serve as a contract on the variation point that potential variants must satisfy to accurately conform to product-line architecture.

The Fractal component model [35–37] can be considered a general component framework rather than a single model since it offers the possibility to capture various different component models through so-called Fractal personalities (comparable to CALM styles). Unlike CALM, where interfaces can be defined with arbitrary properties constrained only by the individually tailored style, Fractal a priori distinguishes three sorts of interfaces, namely server, client, and control interfaces. The model is therefore intrinsically directed and explicitly supports dynamic (re-) configuration of component internals (similar to, e.g., the approach of the Service Component Architecture of the Open Service Oriented Architecture project [38]). Fractal does not consider connectors as primitives and instead connects interfaces directly through bindings. While Fractal supports nesting and sharing of components, we did not find the possibility to introduce novel abstractions such as the CALM services, that go beyond the elementary component-oriented paradigm. The main strength of Fractal lies in its substantial capabilities for reliable dynamic reconfigurations of components as well as configurations that allow introducing a wide range of high-availability technologies from on-line service-adaption and automatic load-balancing to self-healing. Fractal enjoys wide support and comes in several implementations that have been used in telecommunication and other industries and are actively developed.

6.7. Previous work on CadenA

The previous version of CadenA [17] was directly tied to CCM and did not include the meta-modeling and architecture structuring capabilities presented in here. A recent article [15] gives a high-level business-oriented summary of the capabilities of the CadenA tool and its use in product-line development. In contrast, the present paper provides technical details for CALM’s type system (cap. (a)–(e)) and introduces a number of additional capabilities ((f)–(i)).
7. Evaluation

We have evaluated the generality of our CALM typed modeling concepts by using the CALM style mechanism to define CADENA modeling environments for multiple “industrial strength” component models including Enterprise Java Beans [EJB] [39], the CORBA Component Model (CCM) [40], Boeing’s PRiSM component model, and nesC [12,7] — demonstrating the ability to handle component frameworks ranging from enterprise level to real-time (avionics) and embedded systems. Using CADENA’s plug-in facility to enhance the basic functionality of the resulting CADENA CCM modeling framework, we have created an end-to-end model-based development environment for the Java-based OpenCCM implementation (included with the CADENA distribution) that includes code generation facilities for component implementation skeletons and CCM’s configuration and deployment infrastructure.

In a similar fashion, we are building an end-to-end development environment for sensor network product lines with nesC as the underlying infrastructure using the collection of styles presented in the preceding sections. The current implementation includes nesC code import/export facilities that are able to process the entire component library provided with the nesC distribution. As an indicator of the number of artifacts, processing the primary section system of the library that is used in almost every application build yields 129 interface types, 97 component types, and 49 scenarios. Overall the library contains approximately 275 interface types and 456 component types. The models for these libraries are included in the current CADENA release.

Researchers at the Kansas State University’s Sensor Network Center of Excellence are using this framework to develop product lines for multiple application domains including large livestock herd health monitoring, ground water run-off sensing, and radiation detection and response systems — these efforts are providing significant opportunities for us to evaluate the modeling facilities presented here. CALM’s ability to support multiple linked styles within a single modeling framework is playing a significant role in the design of the framework. For example, we are able to provide several higher-level modeling languages above the nesC style that aid scientists who are experts in the application domains (but not in the area of sensor networks) in carrying out the initial design of a sensor system.

Further, CADENA has been used for teaching software architecture concepts at Potsdam University.

8. Conclusions

In this article, we have argued for an increased use of typing to specify and enforce structural constraints in modeling of component-based systems. A type-based approach provides a number of benefits and useful capabilities, it reduces the need to rely on more complicated forms of constraint checking, and it complements other forms of structural and behavioral constraint specification.

For the details of complete framework, we refer the reader to the CADENA website [6]. Website materials illustrate how we have validated the concepts presented here by giving a complete formalization of the typing system and by building a robust tool framework that has been used (a) to specify several widely used component models, and (b) to implement a complete development environment for a Java-based version of CCM. Further, for a complete formalization of CALM’s complex type system please refer to [14]. CADENA is being used by Lockheed Martin engineers to specify representative aspects of architectures for satellite mission control systems.6

8.1. Future work

Currently, CALM employs three tiers of (meta-) modeling out of practical and theoretical considerations. The meta-object facility by the Object Management Group [41] for example argues for four layers of modeling, the model level itself (M0) and three meta-levels (M1–M3). Nevertheless, a strictly layered approach, independently of the number of layers, proves clumsy and somewhat contingent. Advances in knowledge-representation (e.g., [42]) suggest that it is unnecessary to use layers for structuring the modeling entities. Instead, the mention–use relationship (see Section 4.1) is in general a complete partial order (see, e.g., [43]). Modeling entities can therefore simultaneously exist on different meta-levels depending on the specific ascending chain. We hope to arrive at a more consistent meta-modeling theory by discarding strict layering bounds.

In a more short-term effort, we project to include functional aspects into the abstractions of CALM to be able to capture component and system behaviors. Such aspects were covered in the previous version of CADENA, using a simple state-transition language that mostly captured mode-changes of components during execution. Instead of simple state-transitions a concept called lightweight process coordination (LPC) [44] appears more promising: Behaviors are specified in a hierarchical way through networks of services (i.e., elementary, location-agnostic, functional blocks) much akin to a component architecture itself. Also, LPC is completely platform and implementation independent, and therefore can be used inside CALM’s abstractions without loss of generality. Further, the granularity of the abstraction can be chosen freely with LPC, which is also a main aspect of CALM’s current structural abstractions.

6 While non-disclosure agreements prevent us from reporting on the details of this work, the CADENA website contains PowerPoint slides from a two-hour demo given at Lockheed Martin STL in August 2006.