STRUCTURED INTERRELATIONS OF COMPONENT ARCHITECTURES

by

GEORG JUNG

Dipl.-Inform., University of Saarbrücken, Germany, 2001

AN ABSTRACT OF A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences

College of Engineering

Kansas State University

Manhattan, Kansas

2007

ABSTRACT

Software architectures—abstract interrelation models which decompose complex artefacts into modular functional units and specify the connections and relationships among them—have become an important factor in the development and maintenance of large scale, heterogeneous, information and computation systems. In system development, software architecture design has become a main starting point, and throughout the life-cycle of a system, conformance to the architecture is important to guarantee a system's integrity and consistency.

For an effective use of software architectures in initial development and ongoing maintenance, the interrelation models themselves have to be clear, consistent, well-structured, and—in case substantial functionality has to be added, reduced, or changed at any stage of the life-cycle—flexible and manipulable. Further, enforcing the conformance of a software artefact to its architecture is a non-trivial task. Implementation units need to be identifiable and their association to the abstract constructs of the architecture has to be maintained. Finally, since software architectures can be employed at many different levels of abstraction, with some architectures describing systems that span over multiple different computing platforms, associations have to be flexible and abstractions have to be general enough to capture all parts and precise enough to be useful.

An efficient and widely used way to employ software architecture in practice are *middleware-based component architectures*. System development within this methodology relies on the presence of a service layer called middleware which usually resides between operating system (possibly spanning over multiple operating systems on various platforms) and the application described by the architecture. The uniform set of logistic services provided by a middleware allows that communication and context requirements of the functional units, called components, can be expressed in terms of those services and therefore more shortly and concisely than without such a layer. Also, component development in the middleware context can focus on high-level functionality since the low-level logistics is provided by the middleware.

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 instead. Nevertheless, a consequent, adherent, use of typing can seamlessly enforce a wide range of constraints crucial for the structural integrity of architectures and the computation systems specified by them without the steep learning curve associated with first-order logic. Also, type systems scale better than first-order logic both in use and understandability/legibility as well as in computational complexity.

This thesis describes component-oriented architecture modeling with CADENA and introduces the CADENA Architecture Language with Meta-modeling (CALM). CALM uses multi-level type systems to specify complex interaction models and enforce a variety of structural properties and consistency constraints relevant for the development of large-scale component-based systems. Further, CALM generalizes the notion of middleware-based architectures and uniformly captures and maintains complex interrelated architectures integrated on multiple, differing, middleware platforms. CADENA is a robust and extensible tool based on the concepts and notions of CALM 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.

STRUCTURED INTERRELATIONS OF COMPONENT ARCHITECTURES

by

GEORG JUNG

[ˈgeɔʁk ˈjʊŋ]

Dipl.-Inform., University of Saarbrücken, Germany, 2001

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences

College of Engineering

Kansas State University

Manhattan, Kansas

2007

Approved by:

Dr. John Hatcliff

ABSTRACT

Software architectures—abstract interrelation models which decompose complex artefacts into modular functional units and specify the connections and relationships among them—have become an important factor in the development and maintenance of large scale, heterogeneous, information and computation systems. In system development, software architecture design has become a main starting point, and throughout the life-cycle of a system, conformance to the architecture is important to guarantee a system's integrity and consistency.

For an effective use of software architectures in initial development and ongoing maintenance, the interrelation models themselves have to be clear, consistent, well-structured, and—in case substantial functionality has to be added, reduced, or changed at any stage of the life-cycle—flexible and manipulable. Further, enforcing the conformance of a software artefact to its architecture is a non-trivial task. Implementation units need to be identifiable and their association to the abstract constructs of the architecture has to be maintained. Finally, since software architectures can be employed at many different levels of abstraction, with some architectures describing systems that span over multiple different computing platforms, associations have to be flexible and abstractions have to be general enough to capture all parts and precise enough to be useful.

An efficient and widely used way to employ software architecture in practice are *middleware-based component architectures*. System development within this methodology relies on the presence of a service layer called middleware which usually resides between operating system (possibly spanning over multiple operating systems on various platforms) and the application described by the architecture. The uniform set of logistic services provided by a middleware allows that communication and context requirements of the functional units, called components, can be expressed in terms of those services and therefore more shortly and concisely than without such a layer. Also, component development in the middleware context can focus on high-level functionality since the low-level logistics is provided by the middleware.

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 instead. Nevertheless, a consequent, adherent, use of typing can seamlessly enforce a wide range of constraints crucial for the structural integrity of architectures and the computation systems specified by them without the steep learning curve associated with first-order logic. Also, type systems scale better than first-order logic both in use and understandability/legibility as well as in computational complexity.

This thesis describes component-oriented architecture modeling with CADENA and introduces the CADENA Architecture Language with Meta-modeling (CALM). CALM uses multi-level type systems to specify complex interaction models and enforce a variety of structural properties and consistency constraints relevant for the development of large-scale component-based systems. Further, CALM generalizes the notion of middleware-based architectures and uniformly captures and maintains complex interrelated architectures integrated on multiple, differing, middleware platforms. CADENA is a robust and extensible tool based on the concepts and notions of CALM 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.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODE EXAMPLES

ACKNOWLEDGEMENTS

For the work presented here I owe thanks to various friends, colleagues, and family for support, contributions, discussion, and criticism which made it possible to to develop the concepts presented here to their current level of maturity.

I wish to thank my advisor John Hatcliff for his professional and personal guidance, for continuous encouragement and appreciation of my work, and the fruitful cooperation on the many projects we took on. John not only made sure that I always had interesting problems to work on (last minute and deep into the nights), but also that I had food when I came out of the hospital after an appendicitis. Also, Dave Schmidt and Alley Stoughton both became sort of "side-advisors" to me, as each sat through lengthy discussions and presentations of complicated calculi, always providing important insight into the notions of typing or the art of formalization. Adam Childs, Jesse Greenwald, and Matt Hoosier all made crucial contributions to the concepts presented here, the cooperation with them was indispensable for the quality of the work particularly because we never started with the same opinion. More recently, Todd Wallentine joined into the development, providing the practical perspective I sometimes miss, always in a witty way. Luckily, I can call each of them a friend.

For serving on my committee next to John, Dave, and Alley, I would like to thank Matt Dwyer, Joe Kiniry, Robby, and Gurdip Singh. Matt Dwyer's cooperation with our group opened up interesting new directions, he always shows interest in our progress and often takes the two-and-a-half hour trip down from Lincoln, his new place of work. Joe provides the invaluable outside perspective, I hope to have more of a chance for cooperation with Joe in the future. Robby definitely provides an inside perspective as he often drops by to discuss research, or sometimes just to discuss. Cooperation Dr. Singh was continuous and often "parallel" in an interesting way, as his group would look at distributed systems from a networking perspective and thereby provide the complementary and orthogonal view to the architectural perspective. Finally, I want to thank Dr. Medhat Morcos, who on short notice took the assignment of committee chair from the KSU graduate school. The fact that Medhat was always quickly reachable and actively helped to speed up the process made a quick scheduling of my defense possible.

I would also like to thank the remaining members (and former members) of the SAnToS laboratory, especially Venkatesh Prashant Ranganath, who was always ready for controversial discussions, and Xianghua Deng, who recently shared the office and the agony of thesis-writing with me. Further, I would not want to miss the time with Radu Iosif and Dino Distefano, personally and for the insight into model checking I gained from them. Finally, many friends from here and from around the world made living in Manhattan bearable.

For being able to finish this work I would like to thank my family, especially my parents who taught me to go through the world without ever getting lost and to live with ease 8.000 kilometers from home. To Kerstin I owe for staying with me through long years and long distance and through her and my PhD efforts.

CHAPTER 1

INTRODUCTION

"*Divide each difficulty into as many parts as is feasible and necessary to resolve it*"

— René Descartes. Le Discours de la Méthode.

1.1 MOTIVATION

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 stakeholders competing in various dimensions such as 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) *etc.*, have to be systematically reconciled while incrementally adding concerns to the system architecture, a process which continually grows more and more complex throughout the system evolution. Architecture models have to be accurate and robust, *i. e.*, their elements have to faithfully reflect capabilities of the system's execution environment, cater to the abstractions used by various domain experts, and adapt to architecture refinement, globally as well as in detail, while maintaining overall integrity.

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 [11, 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 re-use 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 conforming to certain structural invariants (*e. g.*, tree shaped, list shaped), and for characterizing requirements for converting data between different formats. While 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, 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 [26, 13] defer type checking to other tools or provide external constraint languages [64, 51] 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, 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 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.

This work introduces 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 interrelation constraints that occur in the context of large-scale system development. In detail, this thesis contains the following contributions.

- An intuitive, example-based, description of the nature of the information and computation systems targeted by CALM, together with a semi-formal overview of the concepts and design rationale of CALM and a summary of CALM's multi-tiered (meta-) modeling approach;
- An in-depth, formal presentation of the core concepts of CALM, together with a specification of CALM's language and type-based interrelation semantics, and a rigorous, mechanically leverageable architecture meta-model that can specify industrial component models, component middleware platform capabilities, and domain-specific component modeling languages;
- An example-driven illustration of how CALM can capture widely used middleware-based component-oriented architecture frameworks and concrete architectures within them, with an illustration on how these architectural styles can be enriched with new, useful, abstractions; and
- A detailed presentation of how CALM can be used to interrelate heterogeneous, complex, architecture models from different styles on different levels of abstraction and different layers of functionality and to form traceable integration of planning-level abstractions and implementation-level artefacts, generalizing the concept of middleware platforms.

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 that is freely available for download [7].

The generality and expressiveness of CALM has been demonstrated by using it to capture the definition of a number of realistic component models including Enterprise Java Beans (EJB) [45], the CORBA Component Model (CCM) [54], Boeing's PRiSM component model, and the nesC sensor network component model [28, 29] (see also *chp.* 5). Further, the suitability of CALM/CADENA to integrate into industrial product-development process standards (see also [11]) has been discussed in previous works [8]. An earlier version of CADENA was used by Boeing engineers to develop the mission-control software (built using Boeing's PRiSM framework) for two SCAN Eagle unmanned aircraft flown in the DARPA PCES Capstone Demo at White Sands Missile Range [12]. Also, complete end-to-end model-based development environments for the OpenCCM Java-based CCM implementation and for the nesC component model for highly distributed, embedded, systems have been realized and are available in the CADENA distribution.

The current version of CADENA/CALM has been partially funded and used by Lockheed Martin Advanced Technology Laboratory (ATL) to evaluate the effectiveness of advanced architecture tools as part of their internally funded Software Technology Initiative that seeks to develop innovative technologies for tackling challenges in large-scale system design and integration.

1.2 THE CALM/CADENA PROJECT

1.2.1 BACKGROUND

The CADENA project forked from an approach to apply advanced model checking techniques to industrial-size aviation control systems. From the architecture of the aviation software enhanced with behavioral information, a model would be extracted and fed into the model checker. The model checker would then verify simple temporal properties against the model that were identified to be crucial based on the experience of the industry's developers, yet extremely hard to verify without tool support. Finally, a refinement relation between the architectural model and the actual code would guarantee that the verified properties persist.

To make the model checking feasible for large scale software systems, models of the recurring parts of the software model (for example the communication infrastructure) would be expressed in terms of highly optimized extensions of the model checker. With the modular model checking framework Bogor [5], which is also developed at Kansas State University, the approach could rely on a model checking engine which allows to easily introduce the complex extensions. A particular success was the development of the quasi-cyclic checking which exploited the fact that in periodic real-time contexts it is often possible to restrict the state-space to be explored by the model checker to a single hyper-period [17]. Additional value was added by introducing mode-dependent (*i. e.*, state-dependent) slicing and dependence analysis [32].

CADENA 1 was developed in this context to integrate the various analysis and checking methodologies into a single tool which allows to specify a software architecture with additional behavioral information which can be mechanically leveraged [15, 31, 9]. To be useful for a wide variety of systems, the open standard of the CORBA Component Model (CCM) [54] was chosen as a basis for describing the architectures. Nevertheless, it quickly became apparent that a fixed architecture description language cannot feasibly capture even a small variety of real-life systems. As a patchwork solution in CADENA 1 the discrepancy between standard CCM and the actual industrial system was bridged with a "project" file specifying amendments to CCM properties.

To address the problem of capturing various kinds of system architectures in a uniform, tractable, way, the Cadena Architecture-description Language with Meta-Modeling (CALM) was developed to be the conceptual basis of CADENA 2, a completely re-worked architecture specification framework. Adding a meta-modeling layer on top of the existing two step process of first defining elements and then assembling them into systems provided the notion of the architectural style. The architectural style as a manipulable modeling tier turned out to be an extremely helpful concept to enable many important transformations and interrelations of architectural models and to make them accessible to refinement, comparison, and interaction.

1.2.2 AIMS AND SCOPE

CALM and CADENA strive to capture a wide variety of component-oriented, middleware-based, software architectures. Component-oriented means that the targeted architectures are built in a modular way around well defined, identifiable, units called components which abstract the system's functionality. Middleware-based means that some specific, standardized, (and possibly complex) infrastructure is assumed to exist for each architecture that glues the components together and enables their functionality and cooperation. An architecture captures the topology and interrelations of the modeled system, which means it specifies which component uses which infrastructure service, which components interact with each other, *etc.*. On the other hand, an architecture is not primarily meant to capture behavioral or procedural aspects of a component oriented system.

Consequently, CALM has no operational semantics in the traditional sense (*i. e.*, defining run-time execution behavior, state transitions, or data manipulation). Rather, the semantics of a CALM artefact are a precise network of interrelations of entities with declared names, gradually built by interpreting the CALM syntax. Any operational semantics has to be added to CALM in form of model interpreters. Model interpreters can work on the interrelations or on attributes of the elements within CALM, they can be abstract behavioral descriptions added on top of CALM and executed within CADENA's plugin system, or they can be concrete implementations outside the scope of CALM associated to CALM architectural elements. From the perspective of CALM, such model interpreters are simply additional, more complex attributes to architectural elements.

1.3 CONTRIBUTIONS

The development of CADENA and CALM rested on the shoulders of many people. The initial design team included also Adam Childs, Jesse Grenwald, and Matt Hoosier, who all co-authored most of the CALM-related publications.

Adam Childs contributed many core ideas to the initial design based on his extensive studies on existing component frameworks, but later left the team to work in Chicago. Jesse Greenwald did the main software engineering on CADENA and produced substantial amounts of the code. He also proposed the use of the

Eclipse Modeling Framework (EMF) [19] (see *sec.* 4.5). The three-tier design of CALM and an initial stab towards style refinement was a group achievement, from there I created the CALM formalization, syntax, and formal semantics (deviating in many aspects from the initial design which was too focused on parallels to object oriented programing). Jesse Greenwald, Matt Hoosier, and Adam Childs developed and maintained CADENA, my own contributions to the tool were mostly design, concepts, and feature layout. Currently, CADENA is maintained by Todd Wallentine.

Some examples in this work are taken from a publication co-authored by my advisor Dr. John Hatcliff [35]. Together with [8], [35] is the main foundation of the ideas presented here, nevertheless the thesis goes into far more detail, expands the concepts, and wherever parallels are found, the text is completely re-written.

Work I performed during my studies (after completing the Diplom degree at Saarbrücken University) which is not directly relevant to the concepts presented here includes original work on modeling middleware infrastructure in PROMELA for the model checker dSPIN [14] (see also *sec.* 1.2.1). During the work with the CORBA Component Model, I developed a correlation framework for CCM which allows to optimize network traffic and enables a smarter typing for components [36]. The paper will appear in an extended version also in the International Journal on Software Tools for Technology Transfer (STTT). Unfortunately, since CCM does not allow to easily introduce additional abstractions into the platform definition, the correlation model was only implemented inside CADENA 1 and in the KSU Event Channel implementation developed at Kansas State University by the Networking Research group under Dr. Singh, but did by far not receive the same recognition in practice as it did in publication. One major benefit of the CALM specification framework is that additional abstractions which are not part of an original platform but which allow clearer architecture or strong performance optimization can now be integrated easily.

1.4 ORGANIZATION OF THIS THESIS

This work is organized as follows. Chapter 2 introduces three simple examples of component-oriented architectures which informally outline (but not fully cover) the range of systems which CALM targets. The examples are interconnected in the sense that they each represent a view onto the same system on different levels of detail and abstraction. They reappear separately or together and among other examples throughout the thesis to illustrate various details of CALM or CADENA.

Chapter 3 provides an overview over the concepts of CALM. It describes the modeling elements of CALM and their relations, and gives rationale for the design choices made. Together with a description of the three modeling tiers inside CALM, this chapter presents the meta-model behind CALM, and discusses how it connects with CALM architectural models.

The syntax and interrelation semantics (structures) of core CALM are presented in depth in Chapter 4. Core CALM stands for the part of CALM that can be used to specify three-tiered interrelation models of software architectures. The three modeling tiers are each described with syntax generation rules and interpretation rules which build CALM structures out of the syntax (semantics). The end of Chapter 4 discusses how CALM is realized in CADENA.

For an easier understanding of the definitions in Chapter 4, it is helpful to consult the examples given in Chapter 5. The examples apply the formalisms to capture three widely used middleware architectures. The first example goes in-depth and demonstrates how a given syntactical construct is transferred into CALM structures. The following two examples are meant to show alternatives in the modeling details.

Chapter 6 illustrates how the core CALM constructs are used to realize a complex methodology of model driven design. This chapter outlines the concepts of refinement of architectural styles and shows how refinement is realized in CALM. Based on compliance of architectural models to architectural styles (as specified in *chp.* 4), this chapter describes how models can be migrated between styles. An example shows how these methods can be used to introduce new, useful, abstractions into existing middleware frameworks.

Finally, Chapter 7 shows how implementation/abstraction relations between architectural models from different architectural styles are used to define heterogeneous multi-models of complex systems. Chapter 7 demonstrates how abstract architectural styles can be turned into concrete platforms through implementation relations, and how concrete systems can be traceably connected to their abstract architectures.

The ideas of Chapter 6 and Chapter 7 are presented with much less formal rigor than those of Chapter 4 for two reasons. First, the concepts of core CALM (*chp.* 4) already almost completely entail the formal manipulations needed to realize the more complex ideas (*i. e.*, for the most part *chp.* 6 and *chp.* 7 are mere applications of the interrelation semantics of CALM). Second, the practical side (CADENA) lags behind in implementing the ideas, eventually the formalism will have to adapt to practical aspects which are not foreseeable without the experimentation platform of CADENA. The ideas in Chapter 6 and Chapter 7 should therefore be seen as conceptual, although the concepts are worked out to the necessary level of maturity to be applicable once tool support catches up.

In Chapter 8, CALM is contrasted to related work from the field of software architectures. Chapter 9 concludes and discusses directions of future work in the context of CALM and CADENA.

CHAPTER 2

EXAMPLES OF COMPONENT ARCHITECTURES

"*Further there are two sorts of truths: those of reason and those of fact. The truths of reason are necessary, their opposite impossible; The truths of fact are contingent, their opposite is possible. If a truth is necessary, the cause can be found through analysis, by resolving into simpler truths, until the primitives are reached.*"

— Gottfried Wilhelm Leibniz. Monadologie, 33.

The development of CALM was mostly driven by very concrete problems of large-scale project development and integration of heterogeneous systems. CALM targets descriptions of interconnections of a wide variety of possible (software) systems at various levels of abstraction which all have in common that they are centered around components and rely on what loosely can be identified as a middleware. To provide an intuition as to the nature of artefacts which, from the perspective of CALM, are considered component oriented, middleware based, software architectures (*i. e.*, the systems targeted by CALM), this chapter informally introduces a few small examples. The examples are meant to be "global" as they are picked up throughout this thesis to illustrate various aspects of CALM. Also, the examples are interrelated in the sense that they describe different levels of abstraction of what could be understood as a single computing system.

2.1 AN ABSTRACT SYSTEM OVERVIEW

Often in the early planning stages of a development project, a block diagram is drawn which identifies sub-systems that form concluded units and specifies the interaction in between them. The development of this diagram entails the process of decomposing the system functionality. Further, the diagram helps to understand the place and responsibilities of each sub-system in the overall project and allows to delegate the sub-systems to separate groups of developers.

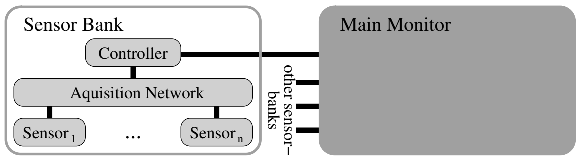


Figure 2.1: Schematic overview of the sensor bank

Figure 2.1 shows a block diagram of a simple sensor bank system where multiple autonomous sensor-banks continuously collect data at different physical locations and transmit the data to a central monitoring station. Each sensor-bank consists of some number of sensors, a controller unit which maintains a link to the monitor, and a local network (the acquisition network) which connects the sensors to the controller. The schematic visualization is helpful to identify the elements of the system, to clarify the separation of concerns, and to assign specific tasks to teams of developers. Nevertheless the schematic omits important details, for example the connection between the controller and the main monitor is inherently different from connections from a sensor to the acquisition network or between the network and the controller, the former being a remote connection, the latter two are local ones. Also, the box-diagram intentionally abstracts away the specifics of individual elements, for example the form of the acquisition network, or the nature of the sensors.

Although the diagram does not specify any particular infrastructure, it does contain information on a high level of abstraction about aspects of what sort of infrastructure the displayed functional units need to rely on. For example, connections from the sensors to the local acquisition network and on to the controller as a minimum need to be able to transmit the data generated by the sensors. The connections are directed, in as much as the end-points of the connections (sensor—network, network—controller) have clearly different roles in the communication. Further, the connection between controller and main monitor is non-local, it has to be able to bridge some (unspecified) spacial distance greater than that of the local connections.

In summary, even in this abstract description the individual components rely on some infrastructure which handles, for example, the communication and is abstracted by the lines between the components. This infrastructure is potentially trivial (if the concrete implementation of the communication is integrated in the components themselves and the infrastructure is just the cable between the components) or it can be sophisticated (if the implementation inside the components amounts to calling, for example, a send command which is understood by the logic of the connector, also if the infrastructure has additional responsibilities such as power supply *etc.*). From the perspective of CALM the infrastructure can be described as an abstract middleware.

2.2 Component oriented source-code organization

The previous example describes a system on a very abstract level, without giving concrete details about any possible implementation. Instead, the architecture could describe a wide variety of implementations unless more details are added as to environment, infrastructure, or internals of the components. CALM can capture this architecture formally due to the very general concept of middleware used in CALM. Any (even trivial) set of services which glue components together into a system counts as middleware or *platform* infrastructure in CALM.

(a) Data-link layer main assembly (b) Hardware clock wrapper

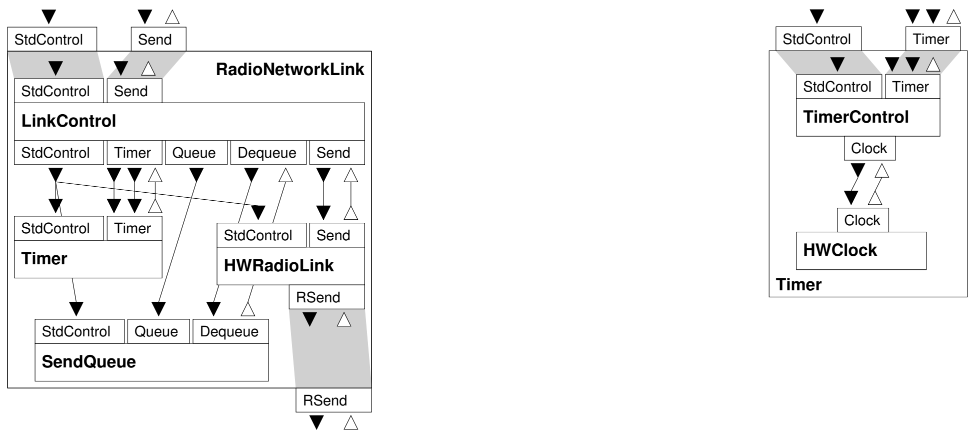


Figure 2.2: Data-link layer assembly in nesC

Nevertheless, the term “middleware-based” usually denotes much more specific contexts in which an existing infrastructure implementation offers a fixed set of services to components which are a priori planned for using this infrastructure. Figure 2.2, for example, shows an architecture using the nesC middleware for highly distributed embedded systems [28, 29]. The nesC component framework is chosen for this example for being representative of the concept of organizing source code in a component oriented way. The nesC infrastructure is written in a C dialect for the TinyOS operating system, the components are written in the same C dialect enhanced with nesC-specific commands which provide the infrastructure abstractions. An architecture, together with the component implementations and the infrastructure code, can be translated into standard C for TinyOS (*i. e.*, the nesC specific extensions are merged with the C parts) and compiled for the targeted computing platforms (both translation and compilation happens in a single, opaque, step). Therefore, nesC can be seen as a set of abstractions added to C for embedded systems.

The abstractions which the nesC infrastructure offers serve two purposes. First, they obviously simplify the communication between different components by abstracting low-level details of remote function calls. Second, the services enforce a certain, modular, structure to a system using them, which makes the functional units identifiable as components. The convenience of the standardized services enables the component oriented design. The abstractions can be mechanically removed, by translating the whole system into standard C, but the component-oriented organization is lost in this process.

A similar approach, which emphasizes the the structuring aspects of the component-oriented paradigm by offering standardized infrastructure abstractions which are implemented in the same or in a closely related language as the components, execute on the same operating systems (where often the middleware provides the glue between different operating systems for different components), and could potentially, together with the components, be translated into monolithic programs, can be identified for example in Enterprise Java Beans (EJB) [45], or the CORBA Component Model (CCM) [54]. In all three examples (nesC, EJB, CCM), the language constructs provided by the component framework are insufficient for implementing the components themselves, instead, standard languages on which the platform is based are also used for the business logic (*e. g.*, C for TinyOS in nesC, Java in EJB, Java, C, C++, *etc.*, for CCM).

The example in Figure 2.2 shows the architecture of a message sending system. The RadioNetworkLink (*fig.* 2.2(a)) consists of a controller (LinkControl), a send queue (SendQueue) to buffer messages, a timer (Timer) to determine re-send timeouts, and a hardware radio link (HWRadioLink) which performs the actual sending of data and receiving of acknowledgements. The timer component is itself realized by and architecture containing two components, a controller (TimerControl) and the actual clock (HWClock) (*fig.* 2.2(b)). The hierarchical organization of architectures, where sub-systems (Timer) can reside inside architecture elements of larger systems (RadioNetworkLink) is called *nesting*.

Compared to the example in Section 2.1, this example is much less abstract. Connecting lines between the components describe specific nesC communication services, icons at the end-points of the lines distinguish the directionality.[1] In fact, the diagrams of Figure 2.2 could be mechanically translated into nesC code skeletons. Both examples are interconnected: Later in this work, the architecture described by Figure 2.2 will be used to describe the implementation of the connection between the controller and the main monitor of the architecture in Figure 2.1.

2.3 Hardware building blocks

The previous section describes an example of the most common understanding of a middleware-based, component-oriented, architecture. As described, components in this understanding are abstract software units residing in a software-based environment.

Figure 2.3 illustrates an even more concrete take on the concept of integrating components on a platform. The architecture describes the hardware configuration of a radio transmission implement. The implement receives digital data from a connected computing device into a digital/analog transformer (D/A). The transformer relays the data in form of high/low voltage values to a combination of a frequency modulator and a phase key shifting device (FM, PKS) which transform the values into a wave/frequency pattern that can be used to send the data through electromagnetic waves. Through an amplifier (AMP), the wave pattern is connected to the antenna of the device. Incoming radio transmissions are decoded in a signal decoder (DEC) which also has control functions to adjust the power of the outgoing signal through a transmit power control (TPC) and the dynamic setting of the transfer channel through the dynamic frequency selection (DFS). Received signals are demodulated into digital signals by the analog/digital scanner (A/D) and transferred back to the computing device through a hardware interrupt (Interrupt).

[1] More detailed descriptions about nesC are found in later chapters.

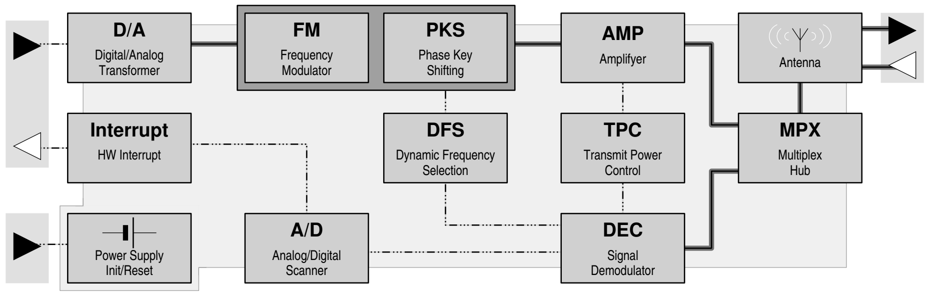


Figure 2.3: Physical layer assembly

In this architecture, components are fixed hardware module blocks with given contacts while the platform is a configurable hardware board. "Services" of the platform are, for example, energy supply for the modules and connection of their contacts. The architecture can contain abstract distinctions which are not manifested in the concrete implements. For example, the architecture can distinguish analog from digital connections which both map to the same sort of electrical connector. Nevertheless the overall component structure is fixed, independently from the abstractions of the infrastructure.

Again, this example has a possible connection to the previous architecture of Section 2.2, as the system described by this architecture can serve as the implementation of the HWRadioLink of Figure 2.2(a).

2.4 Summary

In general, CALM aims to capture specifications of interrelations (*i. e.*, architectures) on various levels of abstraction with various target domains. Commonality of these architecture is that they are decisively modular with all their business logic contained in clearly identifiable, separate, units (component-oriented) which in turn are glued together by some standardized service layer platform (middleware-based). Both the notion of components as well as of middleware are deliberately general and encompass entirely abstract planning constructs, organizational software containers, as well as physically tangible building blocks. Each time CALM distinguishes platform functionality from component functionality.

CHAPTER 3

THE ELEMENTS OF CALM

"*entia non sunt multiplicanda praeter necessitatem*"

— Lex parsimoniae

As laid out in Chapter 1, CALM targets *component-oriented*, *middleware-based*, *software architectures*. Unfortunately, although often used in literature and practice, none of these terms has a concise and generally accepted definition. This chapter gives the descriptions/definitions of these terms which the concepts and notions of CALM are based upon. Also, building on these notions, it provides the rationale for the choice of CALM's modeling primitives. Finally, embedded in this context the three-tier modeling layer concept of CALM is explained.

3.1 BOXES, DOTS, AND LINES

3.1.1 BOXES (COMPONENTS)

A GENERAL DEFINITION OF COMPONENTS

Central to the concepts of CALM is the notion of the *component*. Producing an exact definition of a component is not trivial. In [62] Szyperski offers three different definitions for the *software component* (Preface, Chapter 4, Chapter 20). To illustrate the difficulties to arrive at a widely accepted definition, he references many more given by other authors (Chapter 11). Nevertheless, his own definitions correspond in key aspects and only differ in their level of abstraction. In Chapter 4 he defines

Definition 3.1 (Software Component ([62], *p*. 41)) *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject of composition by third parties.*

CALM does not generally assume that a component is exclusively realized in software, but other aspects of this definition more or less coincide with the requirements of CALM. Namely, according to the definition a component is:

A UNIT OF COMPOSITION. A component is first and foremost a building block within a larger system of similar elements with which it can be composed in some way. It is not required that the component itself is atomic (*i. e.*, elementary and indivisible in some way), but it is required that it is a conceptually complete unit. Many definitions command this unit to be deployable independently, a requirement which emphasizes a certain level of completeness of the component. For CALM, it is sufficient to assume that the component can be deployed within a specific infrastructure which may complement its functionality.

WITH CONTRACTUALLY SPECIFIED INTERFACES. The property of being compositional is manifested through the interfaces of the component which allow interaction between the component and its environment. The definition requires that these interfaces have a contract associated to it, which on both sides allows an assume-guarantee approach. The component guarantees a specific input-output behavior if specified requirements are met by the environment and vice versa. Note that the definition does not state the level of detail of the contracts (*i. e.*, obviously, contracts can vary in whether or not they define timing, accuracy of results of computations, behavior in case of divergence, *etc.*). In practice, the actual form of the contract may vary greatly depending on the context of a component framework.

AND EXPLICIT CONTEXT DEPENDENCIES ONLY. The definition emphasizes the aspect of the deliverables on the environment-side given by the contractually specified interfaces by particularly mentioning that context dependencies need to be explicit. In the CALM view, the requirement of explicit context dependencies is entailed in the requirement to have contractually specified interfaces only. In other words, CALM does not distinguish between business-logic communication and technically necessary infrastructure interaction. In CALM both forms of exchange need to be explicitly defined as interfaces on a component if they are to be part of an architectural model (*i. e.*, not abstracted in the modeling process).

Further, the definition states that a component has the following two properties, it:

CAN BE DEPLOYED INDEPENDENTLY. CALM views components as embedded into a specific infrastructure. In fact, the requirement for a software component to be deployed independently can always only be seen relative to a computation environment, including existing software structures such as an operating system or a middleware layer or even other components which serve to fulfill specific context requirements of the component. CALM realizes that many aspects of this environment are only captured implicitly or simply neglected for reasonably being considered obvious (*e. g.*, any software component needs some kind of computing platform to run). Nevertheless the requirement does apply to the technical process of deployment, here independence can only mean that a component can be integrated into a system mechanically without knowledge about anything else than the part of the environment the component immediately interacts with.

AND IS SUBJECT OF COMPOSITION BY THIRD PARTIES. The component oriented approach emphasizes the distinction between the developer of a component and the user of a component (third party). This requirement states the motivation for all previous requirements. It entails various ideas of component oriented development, such as off-the-shelf components, massive re-use, *etc.*.

ASPECTS OF A CALM COMPONENT

One major difference between the approach to component oriented development by Szyperski versus the approach by CALM is the level of abstraction. Szyperski's definition is important in a practical setting of software development where the value of concrete chunks of program code highly depends on how fit they are for re-use in various concrete environments. CALM instead takes a more abstract, top-down, model-driven approach.

THE SHELL. The most important aspect of a component in CALM (shared with the definition above) is that all interaction with the component happens through contractually defined interfaces. Any interaction of a concrete component with a concrete environment which is not captured by a CALM model of that component has to be considered as abstracted away by the model; any such abstraction needs to be justifiable. CALM calls the definition of a component's interfaces the *shell* of the component. This shell is an abstract entity, CALM does not generally require that the shell protects the component from illegal interaction of the environment with the component's internals. Nevertheless, the shell does guarantee that the component behaves correctly (relative to the requirements of a component framework) if all interaction with it complies to the specification of its shell.

MODEL DRIVEN DEVELOPMENT. The shell of a component depends on the level of abstraction of the CALM model. CALM aims towards an evolutionary development, where the amount of information given by the shell is gradually increased from a high level of abstraction down to the requirements of a concrete system. Relating to the idea of moving from platform independent to platform specific model, development in CALM can be organized as a chain of more and more precise models.

The CALM component Based on the considerations above, an informal definition of a CALM component is the following:

Definition 3.2 (CALM Component) *A component is the abstraction of a functional unit wrapped into a shell which completely defines the component's interaction with the environment through typed interfaces.*

In the CALM formalisms (outlined in Chapter 4), a component is given exclusively through its shell. Considerations about the internals of a component are found in Chapter 7.

3.1.2 Dots (interfaces)

Most commonly, the term *interface* denotes a definition of all operations supported by a class or service (*e. g.*, publicly accessible fields, methods or procedures) in an abstract way (*i. e.*, the interface gives type and name of the fields, and signatures of the methods or procedures, but no implementations and no private fields). Another common meaning of the term interface is a loose group of interaction options (*e. g.*, user interface, programming interface, *etc.*).

In CALM an interface is the definition of an interaction point. As such, the interface stands on its own, which means it is not a part of a component. CALM interfaces abstract the contractual definitions required for the access points of a component. CALM does not define a process algebra or any other language to concretely define interaction contracts, instead CALM interfaces are nominally defined entities which carry specification in form of generic attributes. An informal definition of an interface in CALM is:

Definition 3.3 (CALM Interface) *An interface is an abstract, typed, definition of an interaction point.*

In this definition, *typed* refers to a nominal type which distinguishes interfaces (*i. e.*, an interface can be given simply through a name). A formal definition of this notion is outlined in Chapter 4.

3.1.3 Lines (connectors)

General considerations for architectural connections

The interaction of components with their environment happens through *connectors*. Essentially, any communication infrastructure (shared memory, token ring, bus, network, *etc.*) and any form of communication (message passing, remote procedure/method calls, semaphores, *etc.*) can be abstracted as a connector.

In [58], Shaw argues that connectors deserve "first class status", which means that, for a reasonable and useful modeling of a software architecture, not only the components, but also the nature and form of connections has to be included into the model. The paper states: "For a system to work well, however, the relations among components, or connectors, require as much design and development attention as the components." ([58], *p.* 22). In [3], Allen and Garlan present a system to describe communication protocols realized by connectors based on the communicating sequential processes calculus (CSP) [33]. Again, the connection is emphasized as a crucial part of architectural design. "In particular, the 'lines' connecting the computational elements of such a design clearly have a different status than the computational elements (the 'boxes'), and further, those lines may often represent abstractions with their own nontrivial semantics." ([3], *p.* 242). Finally, in [62], Szyperski states that "A connector, when zooming in, can easily have substantial complexity and really ask for partitioning into components itself." ([62], *p.* 429). Szyperski adds that the impression is created that the concept of the component and the concept of the connector are dual to each other, but he does not follow the thought any further.

In essence, all cited authors see the connectors in an architecture as non-trivial entities. They argue for the necessity to view connectors as an element of equal importance and complexity as the components within an architectural model.

The CALM connector

CALM does consider the connector the dual of the component (*i. e.*, it is given by a shell which defines its access points). The concept of the connector in CALM is based on the following observations about component framework infrastructure (most of which are already discussed in literature as seen above):

Communication is complex. A simple line from one access point of one component to another access point of another component suggests a one-to-one communication link. While most connections between components actually can be described as such (at some level of abstraction), sometimes it is necessary in an architectural model to consider how other communication influences this link. For example, if the link denoted by a connector abstracts a shared memory communication between the two access points, any other access points which communicate on the same line influence the data transfer. Similar considerations might apply on a bus communication, if the architectural model needs to be precise. Therefore, CALM considers also connectors which have more than two access points.[1]

Infrastructure services more than communication. Most component frameworks are based on complex middleware which offers more than communication services. For example, timeout signals, a generic database access, or even specific power supplies, can be part of a complex infrastructure (note that CALM was applied to systems with real-time requirements such as aviation software or to distributed embedded system networks; *i. e.*, systems where the context requirements of a component which have to be captured by the architectural model include more than communication). Since CALM requires that all context dependencies of a component are captured through interface definitions, the environment needs a structure which can connect to access points which model non-communication requirements of a component. Therefore, CALM allows to define connectors with less than two access points (unary connectors) to abstractly capture such infrastructure units.[2]

Services not necessarily atomic. As discussed above, services (communication or general) are often non-trivial and warrant complex implementations themselves. This observation suggests that connectors also can be seen as functional units abstracted by a shell which defines the connector's access points. The service unit abstracted by the shell can then be handled as a black box similar to components.

Based on these considerations, an informal definition of a CALM connector is closely related to that of a component (*def.* 3.2)

Definition 3.4 (CALM Connector) *A connector is the abstraction of a service unit wrapped into a shell which completely defines the connector's interaction with the environment through typed interfaces.*

While they are very similar in structure, the main distinction between the component and the connector in CALM is that components model *business logic*, while connectors model *infrastructure* (informally expressed in their definitions as functional unit or service unit). This difference is not structural, but rather manifested in the relations between components and connectors and the different handling of the two through the modeling tiers of CALM.

3.2 Three tiers of modeling in CALM

CALM uses three distinct modeling tiers to specify architectures. Above these three tiers resides a conceptual meta-model which determines the expressiveness of CALM by providing the notions of the modeling primitives. The highest and most abstract tier is called the *style* tier, the middle layer is called the *module* tier, while the lowest layer is called the *scenario* tier. The use of three modeling tiers is motivated by two major considerations.

[1]Note though that even an implementation of a connection through some central unit (such as a bus) can be displayed as multiple one-to-one connections in an architectural model if the infrastructure guarantees the that the behavior of the central unit is equivalent to what would be expected from individual connections.

[2]Syntactically, the CALM language allows zero-ary connectors too. This "feature" just simplifies the grammar, it is unlikely that zero-ary connectors become useful.

3.2.1 Framework, components, and integration

One motivation for CALM's three tiers springs from the observation that in component oriented development three tasks can naturally be distinguished. First, a framework or platform has to be developed, second, individual components have to be created, and third, the components have to be integrated on the platform to form a system or product. According to our industry collaborators, these tasks are usually performed by separate teams. The division between these tasks also forms the basis of the proposed development models in [11] (*chp.* 6, *p.* 316*ff*).

Platform design

The component framework or platform defines and implements services (communication, allocation, notification, *etc.*) which the components can use. Relying on a thoroughly designed and specified platform simplifies the task of component development substantially, since context dependencies and functional requirements of a component can then easily be expressed relative to the platform infrastructure. Many recurring, low-level, programming tasks can be abstracted by the platform definition so that the component design can focus on high-level business logic implementation.

Consequently, there are various commercial and non-commercial general purpose component frameworks (Microsoft's .NET, the CORBA Component Model, Enterprise Java Beans, the Gnome Bonobo model, *etc.*) and domain-specific component models (PRiSM, nesC, *etc.*). Often, the component model exists as an abstract specification for which various implementations are available (*e. g.*, the CORBA Component Model is implemented for example by OpenCCM [55], CIAO [10], MicroCCM [50], or EJCCM [18]). The developer of the component framework (often called the *system architect*, or, in the context of commercial product families, the *product line architect*) can either chose an existing platform, adapt an existing platform to specific needs, or design a new, domain specific, component framework.

Component development

The component development produces business logic units which interact with the given framework. Teams of component developers may split into two groups, the general component developers and the application specific component developers.

System integration

Integrating the components into a cooperative network supported by the platform infrastructure is a non-trivial task, since system-wide aspects of the development have to be considered (quality of service, real-time aspects, inclusion of features, *etc.*). If the platform is well designed and the available components represent a suitable decomposition of the functionalities of a family of software systems, the system integration should be capable of rapidly producing robust systems within that family.

Matching the development tasks with CALM's tiers

Figure 3.1 illustrates how the three tasks influence each other. The platform defines shapes of components and provides infrastructure elements, the component development produces individual components and the system integration builds the product out of these building blocks.

In CALM, the model of a component infrastructure is captured at the style tier. The name *style* tier derives from the notion that the specification of the platform of a component system shapes the form of system architectures on that platform. The platform specification is therefore called the *architectural style*.

Individual components are captured in CALM by their type (informally: their shell). Libraries (called modules) of such types, together with interface types, are specified on the *module* tier.

Finally, the *scenario* tier of CALM models component architectures (also called component scenarios) as assemblies of instantiated types. As depicted in Figure 3.1, artefacts for these assemblies are drawn immediately from the style tier as well as from the module tier.

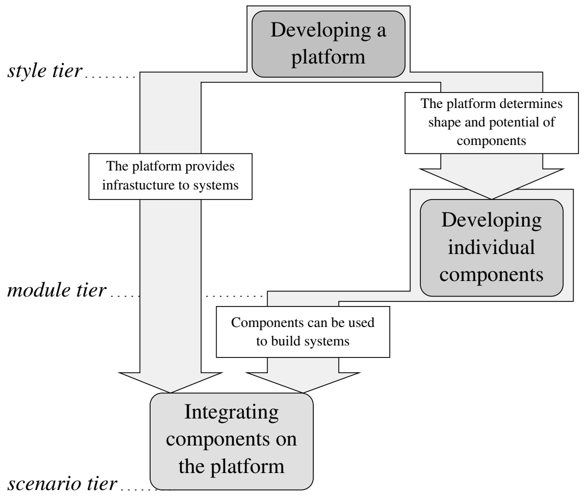

Developing a platform
style tier
The platform determines shape and potential of components
The platform provides infrastructure to systems
Developing individual components
module tier
Components can be used to build systems
Integrating components on the platform
scenario tier

Figure 3.1: Three distinct tasks in component system development

3.2.2 KINDS—TYPES—INSTANCES

The other motivation to use a three tiered approach is the connection to the hierarchy of *kinds*, *types*, and *terms* known from type theory. Types are named abstractions of the structures of terms. Named abstractions in general are so common in programming languages and computing that they are taken for granted: variables in an imperative language are named abstractions of memory locations, constants are named abstractions for values, *etc.*. Types are names of classes of terms (*e. g.*, `int`, `bool`, `string`, ...). Kinds are to types what types are to terms, which means kinds are named abstractions of the structure of types.

In *higher-order typed lambda calculus* [57, 56], kinds are built from a single atomic kind (often written $*$ and pronounced "type"), where $*$ is the kind of all elementary (or *proper*) types (*e. g.*, `int`, `bool`, also function types such as `int` $\rightarrow$ `int`), and $* \rightarrow *$ is the kind of higher-order functions (functions from types to types, *type operators*), $* \rightarrow * \rightarrow *$ the kind of functions from types to type operators, *etc.*.

CALM applies the principle of introducing types of types (kinds) taken from the theory of higher-order functions (and also borrows the terminology of *kinds*, *types*, and *instances* or terms from higher-order typing), but does so from a different angle. The declaration in CALM is top-down, which means instead of abstracting the structure of terms to types and abstracting the structure of types to kinds, CALM defines kinds on the style tier to provide named structures which can be populated with named types on the module tier, and instantiated into concrete assemblies on the scenario tier. Instead of a single constructor "$*$" and the function kind "$kind_1 \rightarrow kind_2$", kinds in CALM are created as stand-alone structures within the three categories component, connector, and interface.

The correspondence to higher-order lambda calculus is therefore remote, nevertheless the idea to introduce kinds as types of types adapts nicely to the realm or architectures, where families of types naturally exist which cannot be traversed (*i. e.*, an interface type cannot be turned into a component type or inherit from a connector type). CALM strives to use kinds, which define the shapes of types, to capture the possibilities and requirements of a component framework.

3.3 THE CALM META-MODEL

As described in Section 3.1 three basic categories of elements (*i. e.*, components, interfaces, connectors) are considered sufficient to describe a substantial range of software architectures. Section 3.2 informally introduces the three modeling tiers of CALM. The kinds defined at the style tier follow a fixed meta-model given through three meta-kinds, one for each category. A CALM model then describes interrelations between elements of these three categories on various levels.

3.3.1 Clarification of terms

The description of concepts which CALM is based on relies on a terminology which is used loosely in literature, with the concrete meaning of most terms varying from publication to publication. While CALM notions are all formally captured (see *sec.* 4), it turns out to be a great help if the informal terms are also clarified and used consistently.

The terms *kind*, *type*, and *instance* are taken from type theory and denote the respective level of abstraction of an entity. The term *meta-kind* describes a conceptual entity of CALM which is used to define kinds.

CALM styles define *architectural styles*, with the term *product line architecture* sometimes used as a synonym for architectural style. Product line architecture should not be confused with *software architecture* for the former describes a style, the latter describes a concrete architecture within a style. More precisely, a software architecture is a concrete interaction model of the functional units of a software system expressed in terms of components, connectors and interfaces. Such a model is called *assembly*, if only the topology is considered, or *scenario*, if aspects beyond the topology are specified (particularly implementation specifics of functional units). Sometimes, non-topological data is referred to as *meta-data* of a scenario, a term which unfortunately is easily falsely associated with meta-kinds.

In CALM, the terms *component*, *connector*, and *interface* denote *categories*. An entity has the category component (interface, connector) if it is ultimately derived from a component (interface, connector) meta-kind. This allows, for example, to talk about component types although all types belong to a specific kind and therefore should be called <*kind-name*>-types correctly; or about component instances, which correctly should be called <*type-name*>-instances.

The access points of types or instances of category component are called *ports*. Ports can only be declared, if a respective *port option* is defined on the kind from which a type/instance is derived. Port options are characterized by three main properties. First, the number of ports complying to the port options which can be or have to be declared on any type within this kind is expressed by an integer interval called *multiplicity*. For each port within a port option, the minimum and maximum fan-out is captured by an integer interval called *multiplexity*. Finally, a port option references an entity of category interface of the respective level of abstraction (*i. e.*, port options of component meta-kinds reference an interface meta-kind, port options of kinds reference interface kinds).

The access points of types or instance of category connector are called *roles*. This nomenclature derives from the idea that from the viewpoint of a connector the entities communicating over any one of its access points take on a specific role in a communication associated with that access point, such as sender, receiver, client, server, *etc.*. Besides the name (role instead of port) the genesis and structure of access points of connector meta-kinds, kinds, types, and instances is analogous to that of components (*i. e.*, there are role options with multiplicity, multiplexity, and interface specification, and roles derived from these).

Interfaces in CALM are not access points *per se*, instead they are definitions of access points. Since each interface defines a point of interaction there are a minimum of two roles associated with it: one entity *provides* the interface, all other entities sharing the access point *use* the interface. Whether an interface is provided or used by a port/role is called the *parity* of the port/role. CALM requires that ports can only connect to roles of the opposite parity. Ports cannot connect to other ports, or to roles of the same parity, roles cannot connect to other roles or to ports of the same parity. This guarantees that every interface in an assembly is complete, having both the provider and the user side present.

3.3.2 Interrelations of the CALM meta-kinds

Figure 3.2 displays the relations between the categories in CALM in a way loosely taken from UML graphic syntax.[3] There are three independent entities, the component meta-kind, the interface meta-kind, and the connector meta-kind. Each of these entities can have any number of attributes. Attributes only exist as contents of a meta-kind. 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. 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). This difference is expressed by the outlined diamond arrow head as opposed to the solid one.

[3]The diagram is not exactly UML, since the entities described are neither objects nor classes nor components in the UML sense.

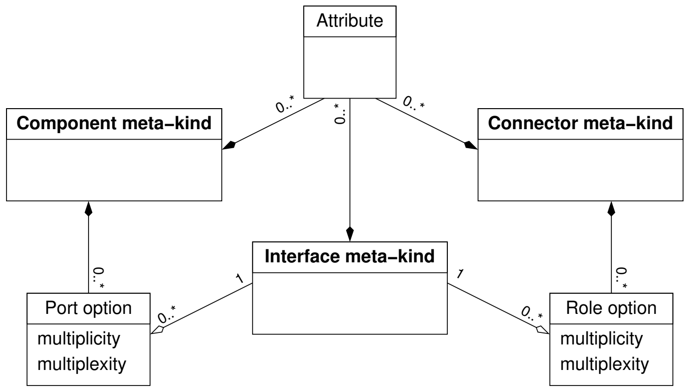


Figure 3.2: Informal meta-model of CALM

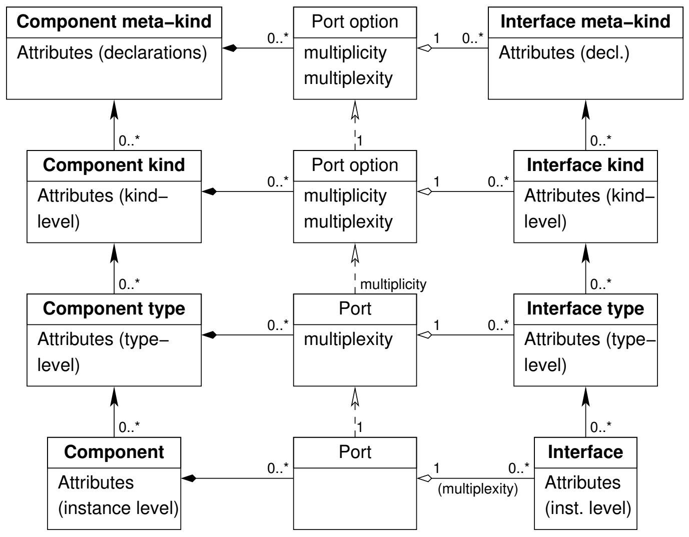


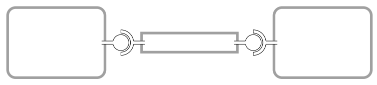
Figure 3.3: Modeling tiers relations (component and interface)

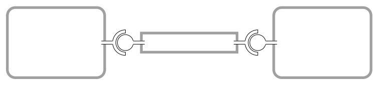
Each of the three basic meta-kinds can be used to define kinds, and subsequently types and instances through the CALM modeling tiers, always with the object to enrich the precision of their interrelations. Figure 3.3 illustrates the development of these interrelations from the meta-kind to the instance 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 again through their ports associate with interface instances taken from the respective types. The actual number of interface instances depends on the port parity and multiplexity.

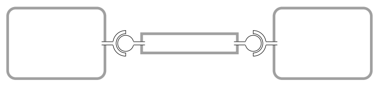
3.3.3 A possible extension to the current CALM meta-model

The meta-model of CALM is currently arranged with the three categories described above and their interrelations. Nevertheless, the meta-model is not entirely fixed, instead CALM tries to be open to relatively easy change and experimentation. This openness is manifested in the fact that the categories are tangible through the meta-kinds which are part of the modeling framework. To introduce new notions into CALM it should be sufficient to create a new category of meta-kind and integrate it with the existing ones. This section discusses an example for a possible extension of the meta-model to illustrate this concept.

One possible extension to the current meta-model which has been discussed is a connection declaration which resides above the actual connector. The problem which this connection declaration tries to address is that the parity of a port/role, which has to be defined at the meta-kind level, may at times fix aspects of a directionality of a connector at a premature level.


(a) directed, provided-used – provided-used


(b) directed, used-provided – used-provided


(c) not directed, connector provides interfaces

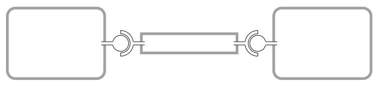

(d) not directed, connector uses interfaces

Figure 3.4: Directionality implied by port/role parities

To illustrate this problem, consider a connector kind which models a one-to-one connection of equal interfaces. Figure 3.4 displays four possible ways of planning a one-to-one communication with different parity arrangements. In the figure, the symbol “−○” describes the provides side of the interface, and “−(” describes the uses side. Figure 3.4(a) and 3.4(b) describe a directed communication, where one component provides the interface on one side of the connector and the other uses the corresponding interface on the other side of the connector. In both cases (*fig.* 3.4(a), 3.4(b)) the implication of directionality springs from the fact that the connector serves simply as a replicator of the components' ports, and that the components' ports have opposite parity implies that they have different roles/tasks in the communication.

Figure 3.4(c) and 3.4(d) describe a situation where the component's ports have the same parity, and therefore an equal status of the two components in terms of their roles in the communication could be assumed. Two options exist to create such an undirected connector, either the connector provides the interfaces on both sides (*fig.* 3.4(c)), maybe implying a more “active” connection service, or it uses the interfaces on both sides (*fig.* 3.4(d)).

CALM has been criticized for the reason that such an early commitment to directedness or undirectedness of a connection prohibits conceptual architectures, which only define that two components do communicate but purposefully leaves the exact nature of that communication unspecified to allow refinement in multiple directions.

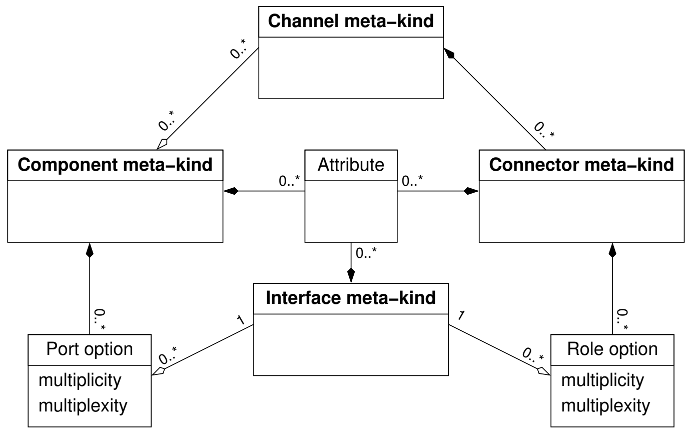


Figure 3.5: An extension to the meta-model of CALM

One possibility to address this problem is to require that every connector is part of an entity of a new category, in this example called *channel*, which is shared among a number of components. Figure 3.5 illustrates a possible extension to the current CALM meta-model by integrating a channel meta-kind into the model as shown in Figure 3.2. In this new model, the connectors become dependent part of a channel, so a conceptual architecture can declare channels only and thereby constrain the use of connectors to specific groups of components. Note that the case where only one channel exists as container for all connectors and associated from all components is equivalent to the current, unconstrained, channel-free, model.

CHAPTER 4

SYNTAX AND STRUCTURAL INTERRELATIONS

"Hence whatever a wise man states he can always define, and what he so defines, he can always carry into practice; for the wise man will on no account have anything remiss in his definitions."

— Confucius. Analects, XIII.iii.7

4.1 CALM/CADENA NOTATIONAL BASICS

4.1.1 NOTATIONAL CONVENTIONS

For better readability in the presentation of formalism and examples, the following notational conventions are used. Concrete CALM syntax is printed in various `typewriter` fonts, with CALM general keywords printed in **`bold`**, and keywords introduced within the examples printed in ***`bold italic`***. Sets of syntactic entities (*i. e.*, non-terminals of the syntax-generation rules) are printed in sansserif.

CALM formalisms make extensive use of (lookup-) maps, which are sets of pairs (id, s) of an identifier id and the structure term s denoted by the identifier. Such pairs will usually be written as $id : s$. The addition of a new element $e = id : s$ to an existing map M (*i. e.*, the transition from M to $M \cup \{(id : s)\}$) will be written with the new element in brackets as $M[id : s]$.

The syntax of CALM will be given inductively through syntax generation rules of the form

$$\frac{premises}{conclusion}$$

where $premises$ is a set of expressions of the form $t_1 \in s_1, \ldots, t_n \in s_n$ and $conclusion$ is a single expression $t \in s$ with $t_1, \ldots, t_n$ being syntactical terms and t being a syntactical term which contains $t_1, \ldots, t_n$ as subterms, and $s, s_1, \ldots, s_n$ being syntactical sets (non-terminals). Generally, the terms $t_1, \ldots t_n$ will be given as variables $v_1, \ldots, v_n$ to summarize a set of rules into a single rule template, where each individual rule can be obtained by replacing each variable v_i by a term $t_i \in s_i$. Every such rule template is equivalent to a production of a Backus-Naur (BNF) grammar of the form

$$s ::= t[s_1/v_1, \ldots, s_n/v_n]$$

(*i. e.*, s produces t with all variables $v_1, \ldots, v_n$ replaced by the respective non-terminals $s_1, \ldots, s_n$), which means that a complete BNF grammar can be obtained by mechanically translating the rule templates.

The interpretation of the semantically sound subset of each syntactic set s is given through interpretation relations $\xrightarrow{s}$ of the form

$$e \vdash \xrightarrow{s} : s \times u$$

where e is the environment containing already defined structures and u is the result of the interpretation, which is most often manifested in an update of the environment under which the terms $t \in s$ are interpreted. Interpretation relations are inductively defined through rule templates which closely correlate to the syntax generation rules (*i. e.*, for each syntax generation rule template there is at least one interpretation rule template). The respective rule templates have the same general form as the syntax generation rules, but the premises are more complex, containing also predicates about the environment, and the conclusion describes a set of elements of an interpretation relation instead of elements of the respective syntactic set.

The complete syntax generation rules of the CALM core language can be found in Appendix A, the syntax interpretation rules in Appendix B.

4.1.2 Elementary sets

Among the sets of syntactic entities, a specific set Identifier of alphanumeric identifiers $id \in$ Identifier is assumed to be available. Also, it is assumed that these identifiers $id \in$ Identifier are capable of efficient referencing, which means that each identifier can be mechanically linked to the structure it denotes in a way that it is sufficient to use a name (*i. e.*, the identifier) in lieu of a structure. Further, the base syntax of CALM requires numeric literals of non-negative integers plus a special, non-numeric, character. Specifically, two sets of numerals are assumed, the set of natural numbers including zero (non-negative integers), written $\mathbb{N}_0$, and the same set with an additional symbol $\star$ (*i. e.*, $\mathbb{N}_0 \cup \{\star\}$), written $\mathbb{N}_0^\star$, where $\star$ stands for a number $z \notin \mathbb{N}_0$ with $z \geq n$ for all $n \in \mathbb{N}_0$. For simplicity, the syntactic numerals are identified with the numbers they represent. Finally, the syntactic sets Type-Spec and Literal are considered elementary in this work for reasons laid out in Section 4.2.1.

4.2 The style tier

A CALM style specifies the available elements of a component oriented architecture in a two-stage process. First, so called *meta-kinds* are declared in an incremental, compositional, way. Meta-kinds are abstract collections of structural specifications visible only on the style tier of CALM (see also *sec.* 3.3). In a second step, the meta-kinds are used to define named *kinds*, fundamental families of architectural elements of fixed structure which are visible to the lower tiers of CALM. It is the collection of kinds, which can be populated with types on the module tier (according to typing constraints also set by the style) and instantiated on the scenario tier, that forms the essence of a CALM architectural style.

Strictly speaking, CALM does not feature “component types” or “interface types” or “connector types”, neither are there “component instances” *etc.*. Rather, types are defined within kinds, instances are constructed within types. For example, if a component kind c is defined within a style, types declared and used within this kind on the lower tiers of CALM are “c-types”, and if a c-type t is declared within c, then instances of this type are t-instances. Nevertheless, instances of a type within a kind of category component will often be referred to as component instances when the specific kind or type are irrelevant for the considered properties, similarly types of component kinds can at times be referred to as component types.

CALM records the meta-kinds specified on the style tier in a meta-kind lookup table $\gamma \in \Gamma$. Elements of a mapping γ are pairs $id : s_{\{i,l,c\}}$ of identifiers and the structures of either interface, connector, or component meta-kinds they denote. Similarly, kinds are recorded in a kind table $\kappa \in \mathrm{K}$. Further, a CALM style defines a type system for platform-specific attribute types $\hat{\mathrm{T}}$ (*sec.* 4.2.1), as well as a set of type-assertions Ξ which constrain the use of types of architectural elements on the lower tiers of CALM.

In summary, a CALM style is a named four-tuple $(\gamma, \kappa, \hat{\mathrm{T}}, \Xi)$. The elements of this tuple are explained in detail in the following sections, with an initial overview given in Table 4.1.

4.2.1 Base-elements of the style tier

Attribute Types

While most of the CALM structure is nominal (*i. e.*, based on declared structural interrelations of named entities), *attributes* can be used for a more detailed, precise, specification of all architectural elements. The type system for CALM’s attributes is part of a separate effort, therefore only some basic ideas will be laid out in this section.

Attribute types	$\hat{\tau} \in \hat{\mathrm{T}}$	Meta-kind level attribute type set for progressive valuation.
	$\hat{\mathrm{T}} \in \hat{\mathcal{A}}$	Set of all meta-kind level attribute type sets.
Attribute maps of meta-kinds	$\hat{\alpha} \in \hat{\mathrm{A}}$	$\hat{\mathrm{A}} =$ $\mathsf{Identifier} \rightharpoonup (\mathbb{a}\ \hat{\mathrm{T}})$
Role options of meta-kinds	$\hat{\rho} \in \hat{\mathrm{R}}$	$\hat{\mathrm{R}} =$ $\mathsf{Identifier} \rightharpoonup (\mathbb{r}\ (\{\mathtt{uses}, \mathtt{provides}\} \times \mathsf{Identifier}^{*} \times \mathcal{Q} \times \mathcal{Q}))$
Port options of meta-kinds	$\hat{\pi} \in \hat{\Pi}$	$\hat{\Pi} =$ $\mathsf{Identifier} \rightharpoonup (\mathbb{p}\ (\{\mathtt{uses}, \mathtt{provides}\} \times \mathsf{Identifier}^{*} \times \mathcal{Q} \times \mathcal{Q}))$
Meta-kind mapping, recording interface, connector, and component meta-kinds	$\gamma \in \Gamma$	$\Gamma =$ $\mathsf{Identifier} \rightharpoonup (\mathbb{i}\ \hat{\mathrm{A}}) \times (\mathbb{l}\ (\hat{\mathrm{A}} \times \hat{\mathrm{R}})) \times (\mathbb{c}\ (\hat{\mathrm{A}} \times \hat{\Pi}))$
Attribute maps of kinds	$\bar{\alpha} \in \bar{\mathrm{A}}$	$\bar{\mathrm{A}} =$ $\mathsf{Identifier} \rightharpoonup (\mathbf{a}\ \bar{\mathrm{T}})$
Role options of kinds	$\bar{\rho} \in \bar{\mathrm{R}}$	$\bar{\mathrm{R}} =$ $\mathsf{Identifier} \rightharpoonup (\mathbf{r}\ (\{\mathtt{uses}, \mathtt{provides}\} \times \mathsf{Identifier}^{\dagger} \times \mathcal{Q} \times \mathcal{Q}))$
Port options of kinds	$\bar{\pi} \in \bar{\Pi}$	$\bar{\Pi} =$ $\mathsf{Identifier} \rightharpoonup (\mathbf{p}\ (\{\mathtt{uses}, \mathtt{provides}\} \times \mathsf{Identifier}^{\dagger} \times \mathcal{Q} \times \mathcal{Q}))$
Kind mapping, recording interface, connector, and component kinds	$\kappa \in \mathrm{K}$	$\mathrm{K} =$ $\mathsf{Identifier} \rightharpoonup (\mathbf{i}\ \bar{\mathrm{A}}) \times (\mathbf{l}\ (\bar{\mathrm{A}} \times \bar{\mathrm{R}})) \times (\mathbf{c}\ (\bar{\mathrm{A}} \times \bar{\Pi}))$
Type constraints	$\xi \in \Xi$	Constraints limiting the use of interface types on the module tier.
		$\xi =$ $\mathit{variable}_1 \sim \mathit{variable}_2$
		$\xi =$ $\mathrm{type}(\mathit{kind}_0\,.\mathit{port\text{-}option}_0) \sim \mathit{variable}$
		$\xi =$ $\mathrm{type}(\mathit{kind}_1\,.\mathit{port\text{-}option}_1) \sim \mathrm{type}(\mathit{kind}_2\,.\mathit{port\text{-}option}_2)$
Style mapping, the collection of CALM architectural styles within a project	$\sigma \in \Sigma$	$\Sigma =$ $\mathsf{Identifier} \rightharpoonup (\mathbf{s}\ (2^{\hat{\mathrm{T}}} \times \Gamma \times \mathrm{K} \times 2^{\Xi}))$

[*] The identifier needs to denote an interface meta-kind.
[†] The identifier needs to denote an interface kind.

Table 4.1: CALM style structure sets and symbols

The possibility to attach attributes to architectural elements at any stage of inception (*i. e.*, to element kinds, types, or instances) is constrained by the meta-kind which the respective architectural element ultimately derives from. This follows the rationale of CALM that the architectural style precisely defines the information captured within architectures which follow this style. To enable the addition of information to architectural elements on all tiers of CALM, the attribute type system has a built-in mechanism to mark the stage at which an individual attribute can be introduced or valuated. The mechanism is based on two concepts, which are type constructors with flexible arity and explicit binding times.

The most important concept of CALM's attribute type system is the notion of *binding time*. A type specification can contain the keywords **`STYLE`**, **`MODULE`**, or **`SCENARIO`**, which mark the type for valuation at the kind, type, or instance level respectively. Combined with the use of type constructors with flexible arity (*e. g.*, **`list`**), the binding time allows to introduce and valuate attributes declared at the style level at any stage within the progressive CALM hierarchy.

As an example, consider the common perception of interfaces as collections of method signatures. In an architectural model using this view, a respective set of method signatures has to be specified for each interface at the type level. To declare the attribute which holds this information at the type level, the architect would introduce an attribute type (*e. g.*, `signature`) as a structure (**`struct`**) containing the name (*e. g.*, as a CALM builtin **`STRING`**), the return type (constructed as an enumeration of the platform types), and a list of arguments. Then, on the interface meta-kind, a type-level list of elements of type `signature` can be used as receptacle for the type information:

```
attribute methods : MODULE signature list;
```

The binding time modifier **MODULE** and the flexible arity of the type constructor **list** allow to specify an arbitrary number of method signatures for each interface type of any kind derived from this meta-kind by giving the appropriate literal at CALM's module tier:

$$\texttt{methods = \{}\mathit{signature}_1\texttt{, \ldots, }\mathit{signature}_n\texttt{ \};}$$

Generally speaking, the binding time modifiers **STYLE**, **MODULE**, and **SCENARIO** are used within the attribute types at the meta-kind level for types $\hat{\tau} \in \hat{\mathrm{T}}$. Through the hierarchy of CALM the attributes are gradually valuated, and the respective modifiers eliminated. The gradual valuation progresses the respective type through the CALM hierarchy

$$\begin{array}{ccccccc} id : \hat{\tau} & \rightsquigarrow & id : \bar{\tau} & \rightsquigarrow & id : \tau & \rightsquigarrow & id : \dot{\tau} \\ \textit{meta-kind} & & \textit{kind} & & \textit{type} & & \textit{value} \end{array}$$

where $\hat{\tau}$ is the meta-kind level structure denoted by the attribute name id, $\bar{\tau}$ the kind level structure, τ the type level structure, and $\dot{\tau}$ the instance level structure. Note that the instance level structure is always completely valuated, type and kind level structures might be, depending on the modifiers used. Further, the transition does not necessarily mean a change in structure, for example kind and type structure are equal if the binding time modifier **MODULE** does not appear in the type. Finally, delayed valuation allows for simple parametric kinds, types, or values.

CALM attribute types can be either *defined* (*i. e.*, constructed from existing types and from the CALM base types **INT**, **CHAR**, **STRING**, **BOOLEAN**, and **ENUM** by use of type constructors such as **struct**, **list**, *etc.*) or just *declared*. Declared types are type names which open a door for tool support of type constructs which would be hard or complicated to express within the CALM type definition language. This tool support is realized in CADENA through the plug-in mechanism.

Generally, a syntactic set of type specifiers $t \in$ Type-Spec is assumed (containing, *e. g.*, the aforementioned basic types **INT**, **CHAR**, ..., and the constructors **list**, **struct**, **union**, ..., as well as the binding time modifiers, declared names, *etc.*) which can express specifications to build a set of attribute types $\hat{\tau} \in \hat{\mathrm{T}}$. This syntactic set is interpreted under a given attribute type set $\hat{\mathrm{T}}$ through the interpretation relation

$$\hat{\mathrm{T}} \vdash \xrightarrow{\textsf{Type-Spec}} : \textsf{Type-Spec} \times \hat{\mathrm{T}}$$

which maps type specifiers $t \in$ Type-Spec to the attribute types $\hat{\tau} \in \hat{\mathrm{T}}$ they denote.

Syntactically a type definition in the syntactic set Attribute-Type-Spec is given through the rule template

$$\frac{id \in \textsf{Identifier}, \quad t \in \textsf{Type-Spec}}{\texttt{typedef}\ id = t \in \textsf{Attribute-Type-Spec}} \tag{4.1}$$

while a type declaration is given through

$$\frac{id \in \textsf{Identifier}}{\texttt{typedecl}\ id \in \textsf{Attribute-Type-Spec}} \tag{4.2}$$

The syntactic set Attribute-Type-Spec is interpreted with a given attribute type set $\hat{\mathrm{T}}$ through the interpretation relation

$$\hat{\mathrm{T}} \vdash \xrightarrow{\textsf{Attribute-Type-Spec}} : \textsf{Attribute-Type-Spec} \times \hat{\mathcal{A}}$$

with $\hat{\mathcal{A}}$ being the set of all possible sets $\hat{\mathrm{T}}$. The interpretation relation is defined through the rules

$$\frac{id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\hat{\mathrm{T}}), \quad t \in \textsf{Type-Spec}, \quad \hat{\mathrm{T}} \vdash t \xrightarrow{\textsf{Type-Spec}} \hat{\tau}^t}{\hat{\mathrm{T}} \vdash \texttt{typedef}\ id = t \xrightarrow{\textsf{Attribute-Type-Spec}} \hat{\mathrm{T}}[id : \hat{\tau}^t]} \tag{4.3}$$

and

$$\frac{id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\hat{\mathrm{T}})}{\hat{\mathrm{T}} \vdash \texttt{typedecl}\ id \xrightarrow{\textsf{Attribute-Type-Spec}} \hat{\mathrm{T}}[id : \hat{\tau}^{id}]} \tag{4.4}$$

where $\hat{\tau}^{id}$ is a new, nominally declared, type with unspecified structure.

Ranges

A *range* $q \in \mathcal{Q}$ is a possibly infinite interval of non-negative integers denoted by elements of the syntactic set Range. Ranges are used in the CALM style as a part of the port/role-option specification (*sec.* 4.2.4 and 4.2.5). A term $s \in \text{Range}$ has the form $s = \texttt{[}n\texttt{..}m\texttt{]}$, with $n \in \mathbb{N}_0$ and $m \in \mathbb{N}_0^\star$, $n \leq m$. The set Range is therefore defined through the two rule-patterns

$$\frac{n \in \mathbb{N}_0, \quad m \in \mathbb{N}_0^\star, \quad n \leq m}{\texttt{[}n\texttt{..}m\texttt{]} \in \text{Range}} \tag{4.5}$$

and

$$\frac{n \in \mathbb{N}_0}{\texttt{[}n\texttt{]} \in \text{Range}} \tag{4.6}$$

The term $s = \texttt{[}n\texttt{..}m\texttt{]}$ denotes the closed interval $\{n, \ldots, m\} \subseteq \mathbb{N}_0$ if m is a number (*i. e.*, $m \in \mathbb{N}_0$), and the open interval $\{n, \ldots\} \subseteq \mathbb{N}_0$ if $m = \star$. If $n > m$, the interval denoted by $\texttt{[}n\texttt{..}m\texttt{]}$ is undefined, the term is illegal in CALM. Further, $s = \texttt{[}n\texttt{..}n\texttt{]}$ can be abbreviated to $s = \texttt{[}n\texttt{]}$. Formally, the semantics of elements of Range is given through the interpretation relation

$$\xrightarrow{\text{Range}} : \text{Range} \times \mathcal{Q},$$

defined through the following rule patterns

$$\frac{n, m \in \mathbb{N}_0, \quad n \leq m}{\texttt{[}n\texttt{..}m\texttt{]} \xrightarrow{\text{Range}} \{n, \ldots, m\}}, \qquad \frac{n \in \mathbb{N}_0}{\texttt{[}n\texttt{..}\star\texttt{]} \xrightarrow{\text{Range}} \{n, \ldots\}} \tag{4.7}$$

and

$$\frac{n \in \mathbb{N}_0}{\texttt{[}n\texttt{]} \xrightarrow{\text{Range}} \{n\}} \tag{4.8}$$

In the following, the syntactical form $\texttt{[}n\texttt{..}m\texttt{]}$ will be used in lieu of the denoted interval $\{n, \ldots, m\}$ for simplicity whenever the distinction is clear.

The use of ranges within the lower tiers of CALM warrants an arithmetic operation to be defined. Namely, ranges can be *shifted* by a subtrahend $c \in \mathbb{N}_0$ with

$$\texttt{[}n\texttt{..}m\texttt{]} - c = \begin{cases} \texttt{[}n-c\texttt{..}m-c\texttt{]}, & \text{if } n \geq c \\ \texttt{[0..}m-c\texttt{]}, & \text{if } n < c, m \geq c \\ \text{undefined}, & \text{if } m < c \end{cases},$$

where

$$\star > c \quad \text{and} \quad \star - c = \star \quad \text{for all } c \in \mathbb{N}_0.$$

This operation represents a situation where c positions in the interval are already "occupied". Note that the third case, $m < c$, could be defined as $\texttt{[}n\texttt{..}m\texttt{]} - c = \texttt{[0..0]}$ instead of being undefined, nevertheless if the case occurs in a practical situation it indicates an error in the specification (see sections on port- and role-options below).

4.2.2 Attributes

An attribute on a meta-kind in CALM is a pair of an identifier $id \in \text{Identifier}$ and a pair $(\mathfrak{a}, \hat{\tau}) \in \{\mathfrak{a}\} \times \hat{\mathrm{T}}$ where $\mathfrak{a}$ is a generic constructor which identifies the structure as an attribute on a meta-kind, and $\hat{\tau} \in \hat{\mathrm{T}}$ is a meta-kind level attribute-type. Attributes of each architectural element are recorded in individual attribute mappings $\hat{\alpha} \in \hat{\mathrm{A}}$ with

$$\hat{\alpha} \in \hat{\mathrm{A}}, \quad \hat{\mathrm{A}} \subseteq \text{Identifier} \times (\{\mathfrak{a}\} \times \hat{\mathrm{T}}).$$

Necessarily, the elements $\hat{\alpha}$ of $\hat{\mathrm{A}}$ are partial functions, each identifier id in Identifier maps to at most one structure $(\mathfrak{a}, \hat{\tau})$. Therefore, $\hat{\mathrm{A}}$ can be written as

$$\hat{\mathrm{A}} = \text{Identifier} \to^{fin} \{(\mathfrak{a}, \hat{\tau}) | \hat{\tau} \in \hat{\mathrm{T}}\},$$

or shorthand:

$$\hat{\mathrm{A}} = \mathsf{Identifier} \rightharpoonup (\mathbb{a}\ \hat{\mathrm{T}}).$$

Attribute mappings in CALM are built through elements of the syntactic set $\mathsf{Attribute}$, which is defined by the rule template

$$\frac{id \in \mathsf{Identifier}, \quad t \in \mathsf{Type\text{-}Spec}}{\texttt{\textbf{attribute}}\ id\ \texttt{\textbf{:}}\ t \in \mathsf{Attribute}} \tag{4.9}$$

Each element of the set $\mathsf{Attribute}$ specifies a single attribute which is added to an existing, possibly empty, attribute mapping $\hat{\alpha}$ through the $\mathsf{Attribute}$ interpretation relation

$$\xrightarrow{\mathsf{Attribute}} : \hat{\mathrm{T}}, \hat{\alpha} \vdash \mathsf{Attribute} \times \hat{\mathrm{A}}$$

which is defined by the rule template

$$\frac{id \in \mathsf{Identifier}, \quad id \notin \mathrm{dom}(\hat{\alpha}), \quad t \in \mathsf{Type\text{-}Spec}, \quad \hat{\mathrm{T}} \vdash t \xrightarrow{\mathsf{Type\text{-}Spec}} \hat{\tau}}{\hat{\mathrm{T}}, \hat{\alpha} \vdash \texttt{\textbf{attribute}}\ id\ \texttt{\textbf{:}}\ t \xrightarrow{\mathsf{Attribute}} \hat{\alpha}[id : (\mathbb{a}\ \hat{\tau})]} \tag{4.10}$$

The attribute mapping $\hat{\alpha} \in \hat{\mathrm{A}}$ then allows a lookup of the attribute through its identifier id :

$$\hat{\alpha} \vdash id \mapsto (\mathbb{a}\ \hat{\tau}).$$

4.2.3 Interface meta-kinds

Interfaces in CALM are characterized by their (kind-, type-, and instance-) name and the set of attributes associated with them. Therefore, CALM formalizes interfaces as named collections of attributes, recorded in attribute mappings. The interface meta-kind is a pair of a name $id \in \mathsf{Identifier}$ and the interface meta-kind structure given through its attribute map $\hat{\alpha}$ together with the generic constructor $\mathbb{i}$.

$$id : (\mathbb{i}\ \hat{\alpha})$$

Since the attribute map $\hat{\alpha}$ is the defining structural part of the interface meta-kind id it can be referred as the type of id.

The body of an interface meta-kind is specified by the syntactic set $\mathsf{Interface\text{-}MK\text{-}Body}$. It is defined as either empty or a semicolon-separated list of attribute declarations of the syntactic set $\mathsf{Attribute}$ by the rule templates

$$\frac{}{\epsilon \in \mathsf{Interface\text{-}MK\text{-}Body}} \tag{4.11}$$

$$\frac{t_1, t_2 \in \mathsf{Interface\text{-}MK\text{-}Body}}{t_1\texttt{\textbf{;}}\ t_2 \in \mathsf{Interface\text{-}MK\text{-}Body}} \tag{4.12}$$

and

$$\frac{a \in \mathsf{Attribute}}{a \in \mathsf{Interface\text{-}MK\text{-}Body}} \tag{4.13}$$

Under a given meta-kind level attribute type set $\hat{\mathrm{T}}$ and an initial attribute mapping $\hat{\alpha}$, the interpretation relation

$$\xrightarrow{\mathsf{Interface\text{-}MK\text{-}Body}} \hat{\mathrm{T}}, \hat{\alpha} \vdash \mathsf{Interface\text{-}MK\text{-}Body} \times \hat{\mathrm{A}}$$

is given through the rule templates

$$\frac{}{\hat{\mathrm{T}}, \hat{\alpha} \vdash \epsilon \xrightarrow{\mathsf{Interface\text{-}MK\text{-}Body}} \hat{\alpha}} \tag{4.14}$$

for an empty interface meta-kind body,

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Interface\text{-}MK\text{-}Body}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash t_1 \xrightarrow{\mathsf{Interface\text{-}MK\text{-}Body}} \hat{\alpha}_1, \\ \hat{\mathrm{T}}, \hat{\alpha}_1 \vdash t_2 \xrightarrow{\mathsf{Interface\text{-}MK\text{-}Body}} \hat{\alpha}_2 \end{array}}{\hat{\mathrm{T}}, \hat{\alpha}_0 \vdash t_1\texttt{\textbf{;}}\ t_2 \xrightarrow{\mathsf{Interface\text{-}MK\text{-}Body}} \hat{\alpha}_2} \tag{4.15}$$

for sequential composition, and

$$\frac{a \in \mathsf{Attribute}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\mathsf{Attribute}} \hat{\alpha}_1}{\hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\mathsf{Interface\text{-}MK\text{-}Body}} \hat{\alpha}_1} \tag{4.16}$$

for an attribute declaration.

The complete interface meta-kind specification is defined by the rule template

$$\frac{id \in \mathsf{Identifier}, \quad i \in \mathsf{Extends\text{-}list}, \quad b \in \mathsf{Interface\text{-}MK\text{-}Body}}{\mathtt{metainterface}\ id\ i\ \{\ b\ \} \in \mathsf{Meta\text{-}Kind\text{-}Spec}} \tag{4.17}$$

The list of identifiers i of the syntactic set Extends-list serves to include existing interface meta-kind attribute maps as defined in Section 4.2.9. The interpretation relation for the syntactic set Meta-Kind-Spec will be given in Section 4.2.8, after component and connector meta-kinds are introduced.

4.2.4 Ports options

Component meta-kinds specify the ability to define ports on components through *port options*. A component meta-kind can have multiple port options which each describe a family of possible ports that can be established on any type which lives within kinds derived from the meta-kind. Each family of ports is specified through a *parity*, which is either **provides** or **uses**, an identifier which serves as a keyword to add ports to the family on the module tier of CALM, an identifier which specifies the interface meta-kind which describes the structure of interfaces associated through the port, and two ranges which describe the *multiplicity* and the *multiplexity* of ports within the family.

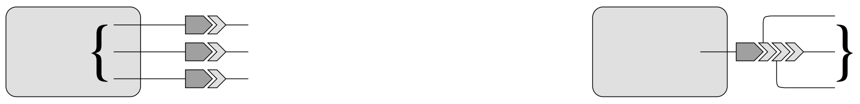

(a) Multiplicity (b) Multiplexity

Figure 4.1: Multiplicity *vs.* multiplexity

The *multiplicity* (*fig.* 4.1(a)) of a port option is a range specifying the minimum and maximum number of ports within the family described by the port option. The *multiplexity* (*fig.* 4.1(b)) of a port option is a range specifying the minimum and maximum fan-out/fan-in of a single port within the family described by the port option. Note that, because the number of ports is part of the type of a component, the multiplicity is a constraint for the module tier of CALM, while the multiplexity, constraining the connectivity of a component inside an assembly, comes into effect on the scenario tier.

Port options are in the syntactic set Port-Option defined by the rule templates

$$\frac{t_1, t_2 \in \mathsf{Range}, \quad i_1, i_2 \in \mathsf{Identifier}}{\mathtt{uses}\ t_1\ i_1\ \mathtt{:}\ i_2\ t_2\mathtt{;} \in \mathsf{Port\text{-}Option}} \tag{4.18}$$

and

$$\frac{t_1, t_2 \in \mathsf{Range}, \quad i_1, i_2 \in \mathsf{Identifier}}{\mathtt{provides}\ t_1\ i_1\ \mathtt{:}\ i_2\ t_2\mathtt{;} \in \mathsf{Port\text{-}Option}} \tag{4.19}$$

Port options are recorded in a port option mapping $\hat{\pi} \in \hat{\Pi}$ as a tuple

$$\hat{\pi} \supseteq \mathsf{Identifier} \times (\{\mathbb{p}\} \times (\{\mathtt{uses}, \mathtt{provides}\} \times \mathsf{Identifier} \times \mathcal{Q} \times \mathcal{Q}))$$

in the form

$$\hat{\pi} \ni id_0 : (\mathbb{p}\ (p, id_i, q_1, q_2))$$

where id_0 is the name (module-level keyword) of the port option, $\mathbb{p}$ is a generic constructor identifying the structure as a port option, p is the parity, q_1 the multiplicity range, and q_2 the multiplexity range.

The interpretation of the syntactic set Port-Option involves a lookup in the global meta-kind table γ to verify the identifier id_i specifying the interface meta-kind which provides the structure for interfaces in the given port family. It is defined by the rule templates

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\pi}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\pi} \vdash \texttt{\textbf{uses}}\ t_1\ id_1\ \texttt{\textbf{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Port\text{-}Option}} \hat{\pi}[id_1 : (\mathbb{p}\ (\texttt{\textbf{uses}}, id_2, q_1, q_2))]} \tag{4.20}$$

for families of used ports, and

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\pi}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\pi} \vdash \texttt{\textbf{provides}}\ t_1\ id_1\ \texttt{\textbf{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Port\text{-}Option}} \hat{\pi}[id_1 : (\mathbb{p}\ (\texttt{\textbf{provides}}, id_2, q_1, q_2))]} \tag{4.21}$$

for families of provided ports.

4.2.5 Role options

Role options on connector meta-kinds serve the same function as port options on component meta-kinds, which is to specify families of interaction points that can/must be added to the architectural element on the type level. Therefore, syntactically and in interpretation, role options are analogous to port options. The syntax rule templates are

$$\frac{t_1, t_2 \in \mathsf{Range}, \quad i_1, i_2 \in \mathsf{Identifier}}{\texttt{\textbf{uses}}\ t_1\ i_1\ \texttt{\textbf{:}}\ i_2\ t_2\texttt{\textbf{;}} \in \mathsf{Role\text{-}Option}} \tag{4.22}$$

and

$$\frac{t_1, t_2 \in \mathsf{Range}, \quad i_1, i_2 \in \mathsf{Identifier}}{\texttt{\textbf{provides}}\ t_1\ i_1\ \texttt{\textbf{:}}\ i_2\ t_2\texttt{\textbf{;}} \in \mathsf{Role\text{-}Option}} \tag{4.23}$$

Role options are recorded in mappings $\hat{\rho} \in \hat{\mathrm{R}}$, again analogous to port options, in the form

$$\hat{\rho} \ni id_0 : (\mathbb{r}\ (p, id_i, q_1, q_2))$$

with $\mathbb{r}$ being the generic constructor which identifies the structure as a role option on the meta-kind level. The syntactic set Role-Option is interpreted according to the rule templates

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\rho}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\rho} \vdash \texttt{\textbf{uses}}\ t_1\ id_1\ \texttt{\textbf{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Role\text{-}Option}} \hat{\rho}[id_1 : (\mathbb{r}\ (\texttt{\textbf{uses}}, id_2, q_1, q_2))]} \tag{4.24}$$

and

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\rho}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\rho} \vdash \texttt{\textbf{provides}}\ t_1\ id_1\ \texttt{\textbf{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Role\text{-}Option}} \hat{\rho}[id_1 : (\mathbb{r}\ (\texttt{\textbf{provides}}, id_2, q_1, q_2))]} \tag{4.25}$$

4.2.6 Component meta-kinds

Component meta-kinds are name collections of port options, specifying the constraints for adding ports to components derived from the meta-kind, and attributes, specifying the data associated with components derived from the meta-kind. Formally, the structure is

$$id : (\mathbb{c}\ (\hat{\alpha}, \hat{\pi}))$$

with id being the name of the component meta-kind, $\mathbb{c}$ being a generic constructor for component meta-kind structures, and $\hat{\alpha}$ and $\hat{\pi}$ being the attribute and port option mappings.

Analogous to interface meta-kinds, the body of component meta-kinds is in the syntactic set Component-MK-Body, which is defined by the rules

$$\frac{}{\epsilon \in \textsf{Component-MK-Body}} \tag{4.26}$$

for an empty body,

$$\frac{t_1, t_2 \in \textsf{Component-MK-Body}}{t_1\,\texttt{;}\; t_2 \in \textsf{Component-MK-Body}} \tag{4.27}$$

for sequential composition,

$$\frac{a \in \textsf{Attribute}}{a \in \textsf{Component-MK-Body}} \tag{4.28}$$

for attribute definitions, and

$$\frac{p \in \textsf{Port-Option}}{p \in \textsf{Component-MK-Body}} \tag{4.29}$$

for port option definitions.

Given a set of attribute types $\hat{\mathrm{T}}$, a meta-kind map γ, an existing attribute mapping $\hat{\alpha}$, and an existing port option mapping $\hat{\pi}$, the interpretation relation for the syntactic set Component-MK-Body

$$\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\pi} \vdash \xrightarrow{\textsf{Component-MK-Body}} \textsf{Component-MK-Body} \times (\hat{\mathrm{A}}, \hat{\Pi})$$

is given by the rule templates

$$\frac{}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\pi} \vdash \epsilon \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}, \hat{\pi}} \tag{4.30}$$

for an empty body,

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Component-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi}_0 \vdash t_1 \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}_1, \hat{\pi}_1, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}_1, \hat{\pi}_1 \vdash t_2 \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}_2, \hat{\pi}_2 \end{array}}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi}_0 \vdash t_1\,\texttt{;}\; t_2 \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}_2, \hat{\pi}_2} \tag{4.31}$$

for sequential composition,

$$\frac{a \in \textsf{Attribute}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\textsf{Attribute}} \hat{\alpha}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi} \vdash a \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}_1, \hat{\pi}} \tag{4.32}$$

for attribute definitions, and

$$\frac{p \in \textsf{Port-Option}, \quad \gamma, \hat{\pi}_0 \vdash p \xrightarrow{\textsf{Port-Option}} \hat{\pi}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\pi}_0 \vdash p \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}, \hat{\pi}_1} \tag{4.33}$$

for port option definitions.

The syntax of the complete component meta-kind is given by the rule option

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Extends-list}, \quad b \in \textsf{Component-MK-Body}}{\texttt{metacomponent}\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Meta-Kind-Spec}} \tag{4.34}$$

Again, the syntactic set Extends-list serves to mix in additional attributes and port options as explained in Section 4.2.9.

4.2.7 Connector meta-kinds

Connector meta-kinds are declared analogously to component meta-kinds with the only exception that instead of a map of port options they feature a map of role options

$$id : (\mathbb{r}\ (\hat{\alpha}, \hat{\rho}))$$

with $\mathbb{r}$ being the constructor to indicate connector meta-kinds. Syntactically, the rule templates

$$\frac{}{\epsilon \in \textsf{Connector-MK-Body}} \quad (4.35) \qquad \frac{t_1, t_2 \in \textsf{Connector-MK-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Connector-MK-Body}} \quad (4.36)$$

and

$$\frac{a \in \textsf{Attribute}}{a \in \textsf{Connector-MK-Body}} \quad (4.37) \qquad \frac{r \in \textsf{Role-Option}}{r \in \textsf{Connector-MK-Body}} \quad (4.38)$$

describe the set Connector-MK-Body analogous to the Component-MK-Body. The interpretation relation

$$\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\rho} \vdash \xrightarrow{\textsf{Connector-MK-Body}} \textsf{Connector-MK-Body} \times (\hat{\mathrm{A}}, \hat{\mathrm{R}})$$

is also defined analogously through the rule templates

$$\frac{}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\rho} \vdash \epsilon \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}, \hat{\rho}} \quad (4.39)$$

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Connector-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho}_0 \vdash t_1 \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}_1, \hat{\rho}_1, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}_1, \hat{\rho}_1 \vdash t_2 \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}_2, \hat{\rho}_2 \end{array}}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho}_0 \vdash t_1\texttt{;}\ t_2 \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}_2, \hat{\rho}_2} \quad (4.40)$$

$$\frac{a \in \textsf{Attribute}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\textsf{Attribute}} \hat{\alpha}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho} \vdash a \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}_1, \hat{\rho}} \quad (4.41)$$

and

$$\frac{r \in \textsf{Role-Option}, \quad \gamma, \hat{\rho}_0 \vdash r \xrightarrow{\textsf{Role-Option}} \hat{\rho}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\rho}_0 \vdash r \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}, \hat{\rho}_1} \quad (4.42)$$

Finally, the complete connector meta-kind is given by the syntax rule template

$$\frac{\mathit{id} \in \textsf{Identifier}, \quad i \in \textsf{Extends-list}, \quad b \in \textsf{Connector-MK-Body}}{\texttt{metaconnector}\ \mathit{id}\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Meta-Kind-Spec}} \quad (4.43)$$

4.2.8 The Set of Meta-Kinds

The meta-kind map $\gamma \in \Gamma$, which contains all interface, component, and connector meta-kinds, is the first fundamental set of a CALM style. As laid out in the preceding sections, it has the form

$$\Gamma = \textsf{Identifier} \rightharpoonup (\mathbb{i}\ \hat{\mathrm{A}}) \cup (\mathbb{c}\ (\hat{\mathrm{A}} \times \hat{\Pi})) \cup (\mathbb{l}\ (\hat{\mathrm{A}} \times \hat{\mathrm{R}})),$$

where the constituent mappings have the form

$$\begin{aligned} \hat{\mathrm{A}} &= \textsf{Identifier} \rightharpoonup (\mathbb{a}\ \hat{\mathrm{T}}) \\ \hat{\Pi} &= \textsf{Identifier} \rightharpoonup (\mathbb{p}\ (\{\texttt{uses}, \texttt{provides}\} \times \textsf{Identifier} \times \mathcal{Q} \times \mathcal{Q})) \\ \hat{\mathrm{R}} &= \textsf{Identifier} \rightharpoonup (\mathbb{r}\ (\{\texttt{uses}, \texttt{provides}\} \times \textsf{Identifier} \times \mathcal{Q} \times \mathcal{Q})) \end{aligned}$$

It is built through the interpretation of the syntactic set Meta-Kind-Spec

$$\hat{\mathrm{T}}, \hat{\alpha} \vdash \xrightarrow{\textsf{Meta-Kind-Spec}} : \textsf{Meta-Kind-Spec} \times \hat{\mathrm{A}}$$

according to the rule templates

$$\frac{\begin{array}{c} \mathit{id} \in \textsf{Identifier}, \quad \mathit{id} \notin \mathrm{dom}(\gamma), \quad i \in \textsf{Extends-list}, \quad \gamma \vdash i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}_0, \emptyset, \emptyset, \\ b \in \textsf{Interface-MK-Body}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash b \xrightarrow{\textsf{Interface-MK-Body}} \hat{\alpha}_1 \end{array}}{\hat{\mathrm{T}}, \gamma \vdash \texttt{metainterface}\ \mathit{id}\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\textsf{Meta-Kind-Spec}} \gamma[\mathit{id} : (\mathbb{i}\ \hat{\alpha}_1)]} \quad (4.44)$$

for interface meta-kinds,

$$\frac{id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\gamma), \quad i \in \textsf{Extends-list}, \quad \gamma \vdash i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}_0, \emptyset, \hat{\pi}_0, \quad b \in \textsf{Component-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi}_0 \vdash b \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}_1, \hat{\pi}_1}{\hat{\mathrm{T}}, \gamma \vdash \mathtt{metacomponent}\ id\ i\ \{\ b\ \} \xrightarrow{\textsf{Meta-Kind-Spec}} \gamma[id : (\mathbb{c}\ (\hat{\alpha}_1, \hat{\pi}_1))]} \tag{4.45}$$

for component meta-kinds, and

$$\frac{id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\gamma), \quad i \in \textsf{Extends-list}, \quad \gamma \vdash i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}_0, \hat{\rho}_0, \emptyset, \quad b \in \textsf{Connector-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho}_0 \vdash b \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}_1, \hat{\rho}_1}{\hat{\mathrm{T}}, \gamma \vdash \mathtt{metaconnector}\ id\ i\ \{\ b\ \} \xrightarrow{\textsf{Meta-Kind-Spec}} \gamma[id : (\mathbb{l}\ (\hat{\alpha}_1, \hat{\rho}_1))]} \tag{4.46}$$

for connector meta-kinds.

4.2.9 Weaving in additional attributes and port/role options

Through a mechanism akin to multiple inheritance in object oriented languages, interface, component and connector meta-kinds can include the attribute, port, and role maps from previously declared meta-kinds of the same category. Nevertheless, as opposed to the concept of inheritance, the attributes, ports options, or role options cannot be overloaded, instead the inclusion of existing maps amounts to a special case of disjoint union, where particularly the left sides of the mappings have to be different.

The syntactical set Extends-list serves to build initial maps from existing meta-kinds which are then extended with the current meta-kind specification as shown in Section 4.2.3, 4.2.6, 4.2.7. It is defined by the rule templates

$$\frac{}{\epsilon \in \textsf{Extends-list}} \tag{4.47}$$

and

$$\frac{i \in \textsf{Identifier-list}}{\mathtt{extends}\ i \in \textsf{Extends-list}} \tag{4.48}$$

with the syntactical set Identifier-list being defined through

$$\frac{id \in \textsf{Identifier}}{id \in \textsf{Identifier-list}} \tag{4.49}$$

and

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}}{id\,\mathtt{,}\,i \in \textsf{Identifier-list}} \tag{4.50}$$

The interpretation of Extends-list is a lookup in the given meta-kind map γ,

$$\gamma \vdash \xrightarrow{\textsf{Extends-list}} : \textsf{Identifier} \times (\hat{\mathrm{A}} \times \hat{\mathrm{R}} \times \hat{\Pi})$$

according to the rule templates

$$\frac{}{\gamma \vdash \epsilon \xrightarrow{\textsf{Extends-list}} \emptyset, \emptyset, \emptyset} \tag{4.51}$$

and

$$\frac{i \in \textsf{Identifier-list}, \quad \gamma \vdash i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}, \hat{\rho}, \hat{\pi}}{\gamma \vdash \mathtt{extends}\ i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}, \hat{\rho}, \hat{\pi}} \tag{4.52}$$

where a given Identifier-list is used to build up the initial attribute map according to the rule templates

$$\frac{id \in \textsf{Identifier}, \quad \gamma \vdash id : (\mathbb{i}\ \hat{\alpha})}{\gamma \vdash id \xrightarrow{\textsf{Extends-list}} \hat{\alpha}, \emptyset, \emptyset}, \quad \frac{id \in \textsf{Identifier}, \quad \gamma \vdash id : (\mathbb{l}\ (\hat{\alpha}, \hat{\rho}))}{\gamma \vdash id \xrightarrow{\textsf{Extends-list}} \hat{\alpha}, \hat{\rho}, \emptyset}, \tag{4.53}$$

$$\frac{id \in \textsf{Identifier}, \quad \gamma \vdash id : (\mathbb{c}\ (\hat{\alpha}, \hat{\pi}))}{\gamma \vdash id \xrightarrow{\textsf{Extends-list}} \hat{\alpha}, \emptyset, \hat{\pi}}$$

and

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad \gamma \vdash id \xrightarrow{\text{Extends-list}} \hat{\alpha}_1, \hat{\rho}_1, \hat{\pi}_1, \\ i \in \text{Identifier-list}, \quad \gamma \vdash i \xrightarrow{\text{Extends-list}} \hat{\alpha}_2, \hat{\rho}_2, \hat{\pi}_2, \\ \text{dom}(\hat{\alpha}_1) \cap \text{dom}(\hat{\alpha}_2) = \emptyset, \quad \text{dom}(\hat{\rho}_1) \cap \text{dom}(\hat{\rho}_2) = \emptyset, \quad \text{dom}(\hat{\pi}_1) \cap \text{dom}(\hat{\pi}_2) = \emptyset \end{array}}{\gamma \vdash id\,\texttt{,}\,i \xrightarrow{\text{Extends-list}} \hat{\alpha}_1 \cup \hat{\alpha}_2, \hat{\rho}_1 \cup \hat{\rho}_2, \hat{\pi}_1 \cup \hat{\pi}_2} \tag{4.54}$$

Note that the identifiers $id \in$ Identifier used to denote the attributes, port options, and role options (referred in the rule template as the domains $\text{dom}(\hat{\alpha})$, $\text{dom}(\hat{\rho})$, $\text{dom}(\hat{\pi})$), need to be disjoint in the respective unions. The correct-by-construction approach of CADENA does not allow identifiers to be re-used at all, nevertheless attributes and port options on component meta-kinds as well as attributes and role options on connector meta-kinds live in separate name-spaces in CADENA, so that the tool technically would not require, for example, attribute and port option names on a component meta-kind to be disjoint in actual models. Nevertheless, the requirement could be introduced into the formalism as well.

4.2.10 LITERALS AND CONSTANTS

When a kind is declared from a given meta-kind, some attributes declared on the meta-kind need to be (partially) valuated. More precisely, the binding time modifier **STYLE** has to be eliminated from the types through partial valuation. The details of this process are part of a separate effort (*sec.* 4.2.1), nevertheless in most cases this means that certain attributes have to be completely valuated.

Generally, the partial valuation necessary for the transition of an attribute from the meta-kind to the kind level is captured through a type transition of the meta-kind level attribute type $\hat{\tau} \in \hat{\mathrm{T}}$ to a corresponding kind level attribute type $\bar{\tau} \in \bar{\mathrm{T}}$

$$id : \hat{\tau} \rightsquigarrow id : \bar{\tau}$$

where $\hat{\tau}$ is equal to $\bar{\tau}$ if the binding time modifiers are at most **MODULE** or $\bar{\tau}$ is a value of the specified type if the binding time of $\hat{\tau}$ is **STYLE** entirely.

To (partially) valuate attributes, CALM features syntactic literals of (partial) values $v \in$ Literal with kind level types $v : \bar{\tau}$ (*i. e.*, these literals can contain typed variables/attributes with binding modifiers being at most **MODULE**). An attribute valuation (syntactic set Attribute-Valuation) is given through the rule template

$$\frac{id \in \text{Identifier}, \quad v \in \text{Literal}}{id \;\texttt{=}\; v \in \text{Attribute-Valuation}} \tag{4.55}$$

The interpretation relation

$$\hat{\alpha}, \bar{\alpha}, \mathcal{C} \vdash \xrightarrow{\text{Attribute-Valuation}} \text{Attribute-Valuation} \times \bar{\mathrm{T}}$$

is defined by the rule template

$$\frac{id \in \text{Identifier}, \quad id : \hat{\tau} \in \hat{\alpha}, \quad id \notin \text{dom}(\bar{\alpha}), \quad v \in \text{Literal}, \quad \hat{\mathrm{T}}, \hat{\tau}, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \bar{\tau}}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}, \mathcal{C} \vdash id \;\texttt{=}\; v \xrightarrow{\text{Attribute-Valuation}} \bar{\alpha}[id : \bar{\tau}]} \tag{4.56}$$

The identifier id has to be defined in an attribute mapping $\hat{\alpha}$, and the type of v has to be correct. Further, to prevent double valuations within the same mapping, id cannot be in the domain of the mapping $\bar{\alpha}$.

Since the valuations can be repetitive as well as complex, CALM offers the possibility to define typed constants $c \in \mathcal{C}$ for better organization. A defined constant can be used in literals in lieu of the literal they stand for. They are defined according to the rule template

$$\frac{id \in \text{Identifier}, \quad t \in \text{Type-Spec}, \quad v \in \text{Literal}}{id \;\texttt{:}\; t \;\texttt{=}\; v \in \text{Constant}} \tag{4.57}$$

and interpreted according to the rule template

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id \notin \text{dom}(\mathcal{C}), \quad t \in \text{Type-Spec}, \quad \hat{\mathrm{T}} \vdash t \xrightarrow{\text{Type-Spec}} \hat{\tau}, \\ v \in \text{Literal}, \quad \hat{\mathrm{T}}, \hat{\tau}, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \bar{\tau} \end{array}}{\hat{\mathrm{T}}, \mathcal{C} \vdash id \;\texttt{:}\; t \;\texttt{=}\; v \xrightarrow{\text{Constant}} \mathcal{C}[id : \bar{\tau}]} \tag{4.58}$$

4.2.11 Interface kinds

To define a kind of an architectural element in CALM, a finished meta-kind is *exported*, which means that a new name is defined and associated with the meta-kind's structure, which in turn is transformed to the kind level. Each kind separately copies and transfers the structure of the meta-kind it is derived from, the kind does not record a nominal reference to the meta-kind (*i. e.*, the original meta-kind is invisible outside the style tier, and relationships among meta-kinds, including equality, do not imply type compatibilities or similar correlations among kinds derived from them).[1]

The interface meta-kind structure $(\mathbb{i}\ \hat{\alpha})$ is transferred during export into an interface kind structure $(\mathbf{i}\ \bar{\alpha})$ with $\mathbf{i}$ being the generic constructor for interface kinds. This transfer means a partial valuation of the associated attributes. Consequently, the syntactical set Interface-Export-Spec, which forms the body of an interface kind definition

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad b \in \textsf{Interface-Export-Spec}}{\texttt{interfacekind}\ id_1\ \texttt{:}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Kind-Definition}} \tag{4.59}$$

is a (possibly empty) list of attribute valuations as given by the rule templates

$$\frac{}{\epsilon \in \textsf{Interface-Export-Spec}} \tag{4.60}$$

$$\frac{t_1, t_2 \in \textsf{Interface-Export-Spec}}{t_1\texttt{;}\ t_2 \in \textsf{Interface-Export-Spec}} \tag{4.61}$$

and

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Interface-Export-Spec}} \tag{4.62}$$

The interpretation of the interface kind definition is given through the rule template

$$\frac{\begin{array}{c} id_1, id_2 \in \textsf{Identifier}, \quad id_1 \notin \kappa, \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma, \quad b \in \textsf{Interface-Export-Spec}, \\ \hat{\mathrm{T}}, \hat{\alpha}, \emptyset, \mathcal{C} \vdash b \xrightarrow{\textsf{Interface-Export-Spec}} \bar{\alpha}_1, \quad \mathcal{A}_k^m(\hat{\alpha}) = \bar{\alpha}_2, \quad \bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2, \quad \mathcal{Q}_a^m(\hat{\alpha}, \bar{\alpha}) \end{array}}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C} \vdash \texttt{interfacekind}\ id_1\ \texttt{:}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\textsf{Kind-Definition}} \kappa[id_1 : (\mathbf{i}\ \bar{\alpha})], \Xi} \tag{4.63}$$

where the identifier id_1, the name of the interface kind which is created, has to be a new identifier, while id_2 denotes the interface meta-kind which is used as the structural template for the creation. The interface kind definition body $b \in$ Interface-Export-Spec serves to valuate the attributes in $\hat{\alpha}$ which need (partial) valuation on the kind level (binding-time modifier **`STYLE`**) to obtain a mapping $\bar{\alpha}_1$. Other attributes which do not need valuation on the kind level are copied to the kind by the function

$$\mathcal{A}_k^m : \hat{\mathrm{A}} \to \bar{\mathrm{A}}.$$

Obviously the partial valuation can only be correct if $\bar{\alpha}_1$ and $\bar{\alpha}_2$ form a partition of the complete kind attribute mapping $\bar{\alpha}$. The completeness of $\bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2$ is checked through a completeness predicate

$$\mathcal{Q}_a^m \subseteq \hat{\mathrm{A}} \times \bar{\mathrm{A}}.$$

Assuming correctness of the partial valuation within the interface kind body, the completeness predicate can be expressed through a simple comparison of the domains

$$\mathcal{Q}_a^m(\hat{\alpha}, \bar{\alpha}) \quad \Leftrightarrow \quad \mathrm{dom}(\hat{\alpha}) = \mathrm{dom}(\bar{\alpha})$$

Nevertheless, depending on the concrete form of the attribute type system (*sec.* 4.2.1), a more complex predicate can be defined. Result of the interpretation is an updated kind environment $\kappa[id_1 : (\mathbf{i}\ \bar{\alpha})]$ containing the new interface kind.

[1] The invisibility of the meta-kind on lower tiers of CALM mainly concerns relations (or the lack thereof) between types of different kinds, which are incompatible even if the kinds they live in are derived from the same meta-kind. Nevertheless structural relationships between meta-kinds do come into effect if types are transfered from one kind to another for example to re-use architectural elements over different platforms, a process called *migration*.

The interpretation of the syntactic set Interface-Export-Spec assembles a kind level attribute mapping $\bar{\alpha}_1$ according to the rule templates

$$\frac{}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}, \mathcal{C} \vdash \epsilon \xrightarrow{\text{Interface-Export-Spec}} \bar{\alpha}} \tag{4.64}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \text{Interface-Export-Spec}, \quad \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t_1 \xrightarrow{\text{Interface-Export-Spec}} \bar{\alpha}_1, \\ \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_1, \mathcal{C} \vdash t_2 \xrightarrow{\text{Interface-Export-Spec}} \bar{\alpha}_2 \end{array}}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t_1 \texttt{;}\ t_2 \xrightarrow{\text{Interface-Export-Spec}} \bar{\alpha}_2} \tag{4.65}$$

and

$$\frac{t \in \text{Attribute-Valuation}, \quad \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\text{Attribute-Valuation}} \bar{\alpha}_1}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\text{Interface-Export-Spec}} \bar{\alpha}_1} \tag{4.66}$$

4.2.12 Type variables and assertions

CALM allows to constrain the use of interface types in an architecture, a feature which mainly aims at the definition of role options on connector kinds that model preconceived infrastructure elements with limited flexibility, but can at times be useful to confine the interfaces of ports on components too. At the style tier, interface kinds are introduced, nevertheless the types which populate the kinds are generally not known at this stage. Therefore CALM allows to declare variables $v \in \mathcal{V}$ which stand for a fixed type in a given interface kind. The syntactical set Type-Var-Decl, defined by the rule template

$$\frac{i \in \text{Identifier-list}, \quad id \in \text{Identifier}}{\textbf{typevar}\ i\ \texttt{:}\ id \in \text{Type-Var-Decl}} \tag{4.67}$$

declares a set i of type variables within the kind id. Its interpretation is given by the rule template

$$\frac{i \in \text{Identifier-list}, \quad id \in \text{Identifier}, \quad id : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad \mathcal{V}_0, id \vdash i \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}_1}{\kappa, \mathcal{V}_0 \vdash \textbf{typevar}\ i\ \texttt{:}\ id \xrightarrow{\text{Type-Var-Decl}} \mathcal{V}_1} \tag{4.68}$$

where the identifier list $i \in$ Identifier-list is interpreted through the rule template

$$\frac{id \in \text{Identifier}, \quad id \notin \operatorname{dom}(\mathcal{V}_0), \quad i \in \text{Identifier-list}, \quad \mathcal{V}_0[id : k], k \vdash i \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}_1}{\mathcal{V}_0, k \vdash id\texttt{,}\, i \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}_1} \tag{4.69}$$

for a comma-separated sequence of identifiers $id\texttt{,}\, i$, and

$$\frac{id \in \text{Identifier}, \quad id \notin \operatorname{dom}(\mathcal{V})}{\mathcal{V}, k \vdash id \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}[id : k]} \tag{4.70}$$

for a single identifier id. In these rules, k is name of the interface kind in which the types denoted by the new variables live. The new variable $id : k$ is stored in the set of variables $\mathcal{V}$ and its name id can be used in lieu of the interface kind name k in component and connector kind definitions.

The variables allow to make assertions about unnamed types of interface kinds. Obviously, the re-use of the same variable denotes type equality, but CALM also allows to introduce explicit constraints about the types through *assertions*. The syntactic set Type-Var-Assert is defined through the rule template

$$\frac{id_1, id_2 \in \text{Identifier}, \quad \sim\ \in \{\texttt{<=}, \texttt{=}, \texttt{>=}\}}{\textbf{assert}\ id_1 \sim id_2 \in \text{Type-Var-Assert}} \tag{4.71}$$

which allows three infix binary relation symbols, `=` for equality, and `>=` and `<=` for sub-type relationships, where sub-typing means an asymmetric compatibility relation of the interface types (see *sec.* 4.3).

The interpretation of Type-Var-Assert includes every new assertion $\xi = id_1 \sim id_2$ into the set of type assertions Ξ, provided the variables are compatible, which means that they have to be declared in the same interface meta-kind. It is given through the rule templates

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : k_1, id_2 : k_2 \in \mathcal{V}, \quad k_1 = k_2}{\Xi, \mathcal{V} \vdash \textbf{assert}\ id_1 \texttt{=}\ id_2 \xrightarrow{\text{Type-Var-Assert}} \Xi[id_1 = id_2]} \tag{4.72}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad id_1 : k_1, id_2 : k_2 \in \mathcal{V}, \quad k_1 = k_2}{\Xi, \mathcal{V} \vdash \texttt{assert } id_1 \texttt{ <= } id_2 \xrightarrow{\textsf{Type-Var-Assert}} \Xi[id_1 \le id_2]} \tag{4.73}$$

and

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad id_1 : k_1, id_2 : k_2 \in \mathcal{V}, \quad k_1 = k_2}{\Xi, \mathcal{V} \vdash \texttt{assert } id_1 \texttt{ >= } id_2 \xrightarrow{\textsf{Type-Var-Assert}} \Xi[id_1 \ge id_2]} \tag{4.74}$$

In the current version of CALM, interface typing constraints $\xi \in \Xi$ apply (component-/connector-) type-locally only, which means that even for globally declared variables the equalities/inequalities they denote only have to hold inside every single component or connector type, but not across types or across kinds even. This current restriction is due to the fact that CADENA only implements local variables (in fact, local to single connector types only), and no experience has been collected as to what a practicable and widely useful semantics for broader spectrum typing constraints would be. Type-local interface typing constraints on the other hand have proven useful particularly for connectors.

4.2.13 Exporting port and role options

Essential for the creation of a component or connector kind is the transition of port/role options from the meta-kind to the kind level. Considering the structure of a port/role option on the meta-kind level

$$id : (\mathbb{p}\ (p, id_m, q_1, q_2)) \qquad \text{and} \qquad id : (\mathbb{r}\ (p, id_m, q_1, q_2)),$$

and the form of a port/role option on the kind level,

$$id : (\mathbf{p}\ (p, id_k, q_1, q_2)) \qquad \text{and} \qquad id : (\mathbf{r}\ (p, id_k, q_1, q_2)),$$

transitioning the port/role option essentially means to replace the interface meta-kind name id_m by an appropriate interface kind name id_k. An element of the syntactic set Export-Kind-Spec is therefore simply a pair of identifiers, one denoting the port/role option, the other denoting the interface kind which replaces the interface meta-kind, and the transfer operator `->`, as defined by the rule template

$$\frac{id_1, id_2 \in \textsf{Identifier}}{id_1 \texttt{ -> } id_2 \in \textsf{Export-Kind-Spec}} \tag{4.75}$$

For the interpretation of Export-Kind-Spec there are four cases. First, the identifier id_1 denotes a role option and the identifier id_2 a component kind directly (as opposed to a type variable as described in Section 4.2.12)

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbb{r}\ (p, id_m, q_1, q_2)) \in \hat{\rho}, \quad id_2 : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad \mathcal{D}(id_m, id_2)}{\gamma, \hat{\rho}, \bar{\rho}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \texttt{ -> } id_2 \xrightarrow{\textsf{Export-Kind-Spec}} \bar{\rho}[id_1 : (\mathbf{r}\ (p, id_2, q_1, q_2))], \Xi} \tag{4.76}$$

The predicate $\mathcal{D}$ assures that the interface kind $id_2 : (\mathbf{a}\ \bar{\alpha})$ is derived from the meta-kind denoted by id_m required by the role option (*i. e.*, $(id_m, id_2) \in \mathcal{D}$ iff $id_2 : (\mathbf{a}\ \bar{\alpha})$ is derived from $id_m : (\mathbb{a}\ \hat{\alpha})$). Second, identifier id_1 denotes a role option and identifier id_2 denotes a type variable $id_2 : id_k \in \mathcal{V}$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbb{r}\ (p, id_m, q_1, q_2)) \in \hat{\rho}, \quad id_2 : id_k \in \mathcal{V}, \quad \mathcal{D}(id_m, id_k)}{\begin{array}{c}\gamma, \hat{\rho}, \bar{\rho}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \texttt{ -> } id_2 \xrightarrow{\textsf{Export-Kind-Spec}} \bar{\rho}[id_1 : (\mathbf{r}\ (p, id_k, q_1, q_2))], \\ \Xi[\text{type}(this.id_1) = id_2]\end{array}} \tag{4.77}$$

here, $this.id_1$ stands for the fully qualified name of the role option. In cases four and three, id_1 denotes a port option instead, id_2 again stands for either an interface kind directly or for a type variable

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbb{p}\ (p, id_m, q_1, q_2)) \in \hat{\pi}, \quad id_2 : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad \mathcal{D}(id_m, id_2)}{\gamma, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \texttt{ -> } id_2 \xrightarrow{\textsf{Export-Kind-Spec}} \bar{\pi}[id_1 : (\mathbf{p}\ (p, id_2, q_1, q_2))], \Xi} \tag{4.78}$$

and

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbb{p}\ (p, id_m, q_1, q_2)) \in \hat{\pi}, \quad id_2 : id_k \in \mathcal{V}, \quad \mathcal{D}(id_m, id_k)}{\begin{array}{c}\gamma, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \texttt{ -> } id_2 \xrightarrow{\textsf{Export-Kind-Spec}} \bar{\pi}[id_1 : (\mathbf{p}\ (p, id_k, q_1, q_2))], \\ \Xi[\text{type}(this.id_1) = id_2]\end{array}} \tag{4.79}$$

Note that the cases where, instead of a meta-kind name, id_2 is a type variable name, lead to an update of the typing constraint set Ξ. For the complete style it is sufficient then to calculate the transitive hull of Ξ and eliminate the constraints which contain variables. The relations between the types of the interface kinds associated with specific roles and ports are thereby preserved without recording the variables. Note again that all typing constraints apply component-/connector-type-locally only.

4.2.14 Component kinds

To define a component kind, three things are necessary. First, a component meta-kind as template, second, a partial valuation of the attributes, and third, transferring the port options from the meta-kind to the kind level by selecting appropriate interface kinds. Syntactically, the component kind definition is given by the rule template

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad b \in \textsf{Component-Export-Spec}}{\texttt{componentkind}\ id_1\ \texttt{:}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Kind-Definition}} \tag{4.80}$$

with id_1 being the name of the new component kind, id_2 being the name of the meta-kind from which id_1 is derived, and b being the body of the component kind definition. The body b is given through six rule templates, which are

$$\frac{}{\epsilon \in \textsf{Component-Export-Spec}} \tag{4.81}$$

$$\frac{t_1, t_2 \in \textsf{Component-Export-Spec}}{t_1\texttt{;}\ t_2 \in \textsf{Component-Export-Spec}} \tag{4.82}$$

for an empty body and for sequential composition,

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Component-Export-Spec}} \tag{4.83}$$

$$\frac{t \in \textsf{Export-Kind-Spec}}{t \in \textsf{Component-Export-Spec}} \tag{4.84}$$

for attribute valuations and port option transfers, and

$$\frac{t \in \textsf{Type-Var-Spec}}{t \in \textsf{Component-Export-Spec}} \tag{4.85}$$

$$\frac{t \in \textsf{Type-Var-Assert}}{t \in \textsf{Component-Export-Spec}} \tag{4.86}$$

for new type variables and assertions over interface types.

The interpretation relations are given straightforwardly on the basis of the previous sections. The body $b \in \textsf{Component-Export-Spec}$ is interpreted by the relation

$$\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash \xrightarrow{\textsf{Component-Export-Spec}} : \textsf{Component-Export-Spec} \times (\bar{\mathrm{A}} \times \bar{\Pi} \times 2^{\Xi} \times 2^{\mathcal{V}})$$

where 2^{Ξ} (powerset of Ξ) is the set of all possible typing constraint sets, and $2^{\mathcal{V}}$ is the set of all possible sets of variables. The rule templates are

$$\frac{}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash \epsilon \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}, \Xi, \mathcal{V}} \tag{4.87}$$

for an empty body,

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Component-Export-Spec}, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash t_1 \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}_1, \bar{\pi}_1, \Xi_1, \mathcal{V}_1 \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_1, \hat{\pi}, \bar{\pi}_1, \Xi_1, \mathcal{V}_1, \mathcal{C} \vdash t_2 \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}_2, \bar{\pi}_2, \Xi_2, \mathcal{V}_2 \end{array}}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash t_1\texttt{;}\ t_2 \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}_2, \bar{\pi}_2, \Xi_2, \mathcal{V}_2} \tag{4.88}$$

(*i. e.*, sequential interpretation) for sequential composition,

$$\frac{t \in \textsf{Attribute-Valuation}, \quad \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\textsf{Attribute-Valuation}} \bar{\alpha}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}_1, \bar{\pi}, \Xi, \mathcal{V}} \tag{4.89}$$

(updating the attribute map $\bar{\alpha}$) for an attribute valuation, and

$$\frac{t \in \textsf{Export-Kind-Spec}, \quad \gamma, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Export-Kind-Spec}} \bar{\pi}_1, \Xi_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}_1, \Xi_1, \mathcal{V}} \tag{4.90}$$

(updating the port map $\bar{\pi}$) for fixing the interface kind of a port option. Further, (local) type variables can be introduced through

$$\frac{t \in \textsf{Type-Var-Decl}, \quad \kappa, \mathcal{V}_0 \vdash t \xrightarrow{\textsf{Type-Var-Decl}} \mathcal{V}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}_0, \mathcal{C} \vdash t \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}, \Xi, \mathcal{V}_1} \tag{4.91}$$

and interface type assertions added by

$$\frac{t \in \textsf{Type-Var-Assert}, \quad \Xi_0, \mathcal{V} \vdash t \xrightarrow{\textsf{Type-Var-Assert}} \Xi_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}, \Xi_1, \mathcal{V}} \tag{4.92}$$

The complete component kind is added to the kind environment κ according to the rule template

$$\frac{\begin{array}{c} id_1, id_2 \in \textsf{Identifier}, \quad id_1 \notin \kappa, \quad id_2 : (\mathbb{c}\,(\hat{\alpha}, \hat{\pi})) \in \gamma, \quad b \in \textsf{Component-Export-Spec}, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \emptyset, \hat{\pi}, \emptyset, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash b \xrightarrow{\textsf{Component-Export-Spec}} \bar{\alpha}_1, \bar{\pi}, \Xi_1, \mathcal{V}_1, \\ \mathcal{A}_k^m(\hat{\alpha}) = \bar{\alpha}_2, \quad \bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2, \quad \mathcal{Q}_a^m(\hat{\alpha}, \bar{\alpha}), \quad \mathcal{Q}_p^m(\hat{\pi}, \bar{\pi}) \quad \Xi_2 = \mathrm{cls}_{(\mathcal{V}_1 \backslash \mathcal{V}_0)}(\Xi_1) \end{array}}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash \texttt{componentkind}\; id_1 \texttt{ : } id_2 \texttt{ \{ } b \texttt{ \} } \xrightarrow{\textsf{Kind-Definition}} \kappa[id_1 : (\mathbf{c}\,(\bar{\alpha}, \bar{\pi}))], \Xi_2} \tag{4.93}$$

Analogous to the interpretation of interface kind definitions, attributes which are unchanged between the meta-kind and the kind level are transferred by the function $\mathcal{A}_k^m : \hat{\mathrm{A}} \to \bar{\mathrm{A}}$, the completeness of the kind level attribute environment is ensured by the predicate $\mathcal{Q}_a^m$. Similarly, the completeness of the port option exports is assured through the predicate $\mathcal{Q}_p^m : \hat{\Pi} \times \bar{\Pi}$. This predicate can be defined as

$$\mathcal{Q}_p^m(\hat{\pi}, \bar{\pi}) \qquad \Leftrightarrow \qquad \mathrm{dom}(\hat{\pi}) = \mathrm{dom}(\bar{\pi}),$$

(*i. e.*, simple completeness) assuming that the transition through the interpretation of the set Export-Kind-Spec is correct.

Locally declared interface type variables $v \in \mathcal{V}_1 \backslash \mathcal{V}_0$ are not added to the global set of type variables $\mathcal{V}_0$, instead the transitive closure of Ξ_1 is calculated and the new variables eliminated, so that only the (directly or indirectly declared) relations between the fully qualified port options are recorded in the interface type constraint set Ξ_2, which is part of the result of the interpretation of the component kind definition. The function

$$\mathrm{cls}_{\mathcal{V}} : 2^{\Xi} \to 2^{\Xi}, \qquad \mathrm{cls}_{\mathcal{V}}(\Xi_1) \mapsto \Xi_2$$

(where 2^{Ξ} is the set of all possible type constraint sets Ξ) calculates the transitive closure of Ξ_1 and eliminates all constraints which contain variables $v \in \mathcal{V}$ from the result.

4.2.15 Connector kinds

Connector kinds are defined from connector meta-kinds in exactly the same way as component kinds from component meta-kinds. Namely, the overall syntax is given by the rule template

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad b \in \textsf{Connector-Export-Spec}}{\texttt{connectorkind}\; id_1 \texttt{ : } id_2 \texttt{ \{ } b \texttt{ \} } \in \textsf{Kind-Definition}} \tag{4.94}$$

while the overall interpretation is given by the rule template

$$\frac{\begin{array}{c} id_1, id_2 \in \textsf{Identifier}, \quad id_1 \notin \kappa, \quad id_2 : (\mathbb{l}\,(\hat{\alpha}, \hat{\rho})) \in \gamma, \quad b \in \textsf{Connector-Export-Spec}, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \emptyset, \hat{\rho}, \emptyset, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash b \xrightarrow{\textsf{Connector-Export-Spec}} \bar{\alpha}_1, \bar{\rho}, \Xi_1, \mathcal{V}_1, \\ \mathcal{A}_k^m(\hat{\alpha}) = \bar{\alpha}_2, \quad \bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2, \quad \mathcal{Q}_a^m(\hat{\alpha}, \bar{\alpha}), \quad \mathcal{Q}_r^m(\hat{\rho}, \bar{\rho}), \quad \Xi_2 = \mathrm{cls}_{(\mathcal{V}_1 \backslash \mathcal{V}_0)}(\Xi_1) \end{array}}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash \texttt{connectorkind}\; id_1 \texttt{ : } id_2 \texttt{ \{ } b \texttt{ \} } \xrightarrow{\textsf{Kind-Definition}} \kappa[id_1 : (\mathbf{l}\,(\bar{\alpha}, \bar{\rho}))], \Xi_2} \tag{4.95}$$

Similar to components, the body b of a connector kind definition allows for attribute valuations and role option transitions

$$\frac{t \in \text{Attribute-Valuation}}{t \in \text{Connector-Export-Spec}} \quad (4.96) \qquad \frac{t \in \text{Export-Kind-Spec}}{t \in \text{Connector-Export-Spec}} \quad (4.97)$$

as well as local interface type variables and respective typing constraints.

$$\frac{t \in \text{Type-Var-Decl}}{t \in \text{Connector-Export-Spec}} \quad (4.98) \qquad \frac{t \in \text{Type-Var-Assert}}{t \in \text{Connector-Export-Spec}} \quad (4.99)$$

Finally, the interpretation of the syntactic set Connector-Export-Spec is analogous to the interpretation of Component-Export-Spec.

4.2.16 Style head and element inclusion

The elements described so far (*i. e.*, attribute type specifications, constants, interface type variable declarations and assertions, meta-kind declarations, and kind definitions) make the body of a CALM architectural style. Syntactically, the body of a style is given by the set Style-Body, which is defined by the rule templates

$$\frac{}{\epsilon \in \text{Style-Body}} \quad (4.100) \qquad \frac{t_1, t_2 \in \text{Style-Body}}{t_1\texttt{;}\ t_2 \in \text{Style-Body}} \quad (4.101)$$

for an empty body and sequential composition

$$\frac{t \in \text{Meta-Kind-Spec}}{t \in \text{Style-Body}} \quad (4.102) \qquad \frac{t \in \text{Kind-Definition}}{t \in \text{Style-Body}} \quad (4.103)$$

for meta-kinds and kinds,

$$\frac{t \in \text{Type-Var-Decl}}{t \in \text{Style-Body}} \quad (4.104) \qquad \frac{t \in \text{Type-Var-Assert}}{t \in \text{Style-Body}} \quad (4.105)$$

for interface type variables and assertions (constraint definitions), and

$$\frac{t \in \text{Attribute-Type-Spec}}{t \in \text{Style-Body}} \quad (4.106) \qquad \frac{t \in \text{Constant}}{t \in \text{Style-Body}} \quad (4.107)$$

for attribute types and attribute value constants. The interpretation of the style body is straightforward, as given, for example, for meta-kind declarations by the rule template

$$\frac{t \in \text{Meta-Kind-Spec}, \quad \hat{\mathrm{T}}, \gamma_0 \vdash t \xrightarrow{\text{Meta-Kind-Spec}} \gamma_1}{\hat{\mathrm{T}}, \gamma_0, \kappa, \Xi, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Style-Body}} \hat{\mathrm{T}}, \gamma_1, \kappa, \Xi, \mathcal{V}, \mathcal{C}} \quad (4.108)$$

or for the kind definitions by the rule template

$$\frac{t \in \text{Kind-Definition}, \quad \hat{\mathrm{T}}, \gamma, \kappa_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Kind-Definition}} \kappa_1, \Xi_1}{\hat{\mathrm{T}}, \gamma, \kappa_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Style-Body}} \hat{\mathrm{T}}, \gamma, \kappa_1, \Xi_1, \mathcal{V}, \mathcal{C}} \quad (4.109)$$

The rules for an empty body or for sequential composition, for type variables and constraints, and for attribute types and constants within Style-Body are omitted here, they can be found in Appendix B.

One of the main reasons to capture the architectural style within the overall model of the architecture is the possibility to combine, refine, enrich, or reduce styles easily and adapt the architectural models within the styles accordingly. The complete style specification, as given through the syntactic set Style, is defined by the rule option

$$\frac{id_0 \in \text{Identifier} \quad i \in \text{Import-Spec}, \quad b \in \text{Style-Body}}{\texttt{style}\ id_0\ i\ \texttt{\{}\ b\ \texttt{\}} \in \text{Style}} \quad (4.110)$$

where the import specification $i \in \text{Import-Spec}$ is defined by the rule templates

$$\frac{}{\epsilon \in \text{Import-Spec}} \quad (4.111) \qquad \frac{t \in \text{Union}}{\texttt{include}\ t \in \text{Import-Spec}} \quad (4.112)$$

If the import specification is nonempty, the keyword **`include`** is followed by a term $t \in \text{Union}$, which is an expression over identifiers denoting existing CALM architectural styles, using the operators `+` for union and `^` for intersection. In detail the syntax of the import specification is defined by the rule templates

$$\frac{t \in \text{Intersection}}{t \in \text{Union}} \quad (4.113) \qquad \frac{t_1 \in \text{Intersection}, \quad t_2 \in \text{Union}}{t_1 + t_2 \in \text{Union}} \quad (4.114)$$

$$\frac{t \in \text{Atom}}{t \in \text{Intersection}} \quad (4.115) \qquad \frac{t_1 \in \text{Atom}, \quad t_2 \in \text{Intersection}}{t_1 \ \hat{}\ t_2 \in \text{Intersection}} \quad (4.116)$$

$$\frac{id \in \text{Identifier}}{id \in \text{Atom}} \quad (4.117) \qquad \frac{t \in \text{Union}}{(\ t\ ) \in \text{Atom}} \quad (4.118)$$

The atom is either an identifier $id \in \text{Identifier}$, or a subterm $t \in \text{Union}$ in parenthesis. In case of the atom being and identifier, it needs to be the name of an existing CALM style. The interpretation of an atom of the import specification is given through the rule templates

$$\frac{id \in \text{Identifier}, \quad id : (\mathbf{s}\ (\hat{\mathrm{T}}, \gamma, \kappa, \Xi)) \in \sigma}{\sigma \vdash id \xrightarrow{\text{Atom}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \quad (4.119)$$

for an atom which denotes a single CALM style, and

$$\frac{t \in \text{Union}, \quad \sigma \vdash t \xrightarrow{\text{Union}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi}{\sigma \vdash (\ t\ ) \xrightarrow{\text{Atom}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \quad (4.120)$$

for an atom which contains a sub-term in parentheses. The interpretation of an intersection $t \in \text{Intersection}$ passes on the results of the involved atom in the unary case, as given by the rule template

$$\frac{t \in \text{Atom}, \quad \sigma \vdash t \xrightarrow{\text{Atom}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi}{\sigma \vdash t \xrightarrow{\text{Intersection}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \quad (4.121)$$

and combines the results in the binary case, as given by the rule template

$$\frac{\begin{array}{c} t_1 \in \text{Atom}, \quad t_2 \in \text{Intersection}, \quad \sigma \vdash t_1 \xrightarrow{\text{Atom}} \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \\ \sigma \vdash t_2 \xrightarrow{\text{Intersection}} \hat{\mathrm{T}}_2, \gamma_2, \kappa_2, \Xi_2 \end{array}}{\sigma \vdash t_1 \ \hat{}\ t_2 \xrightarrow{\text{Intersection}} \hat{\mathrm{T}}_1 \sqcap \hat{\mathrm{T}}_2, \gamma_1 \sqcap \gamma_2, \kappa_1 \sqcap \kappa_2, \Xi_1 \sqcap \Xi_2} \quad (4.122)$$

Here, the intersection operator $\sqcap$ requires uniqueness of names, which means that the set $S = S_1 \sqcap S_2$ is only defined if there is no element $e_1 = id_1 : s_1 \in S_1$ with $id_1 \in \text{dom}(S_2)$ but $e_1 \notin S_2$. In other words, if elements from S_1 and elements from S_2 have the same identifier but a different structure, the intersection is undefined.[2] Mainly, this implies

$$S_1 \sqcap S_2 \text{ defined} \quad \Rightarrow \quad \text{dom}(S_1 \sqcap S_2) = \text{dom}(S_1) \cap \text{dom}(S_2).$$

Similar, the interpretation of a union $t \in \text{Union}$ either passes on the results in the unary case

$$\frac{t \in \text{Intersection}, \quad \sigma \vdash t \xrightarrow{\text{Intersection}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi}{\sigma \vdash t \xrightarrow{\text{Union}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \quad (4.123)$$

[2]It is natural, and in fact intended for realization in CADENA, to require that elements included in such an intersection actually originate from the same definition, as opposed to just being structurally equal.

or relies on a specific union operation $\sqcup$ which requires unambiguous names analogously to the intersection operation $\sqcap$

$$\frac{t_1 \in \text{Intersection}, \quad t_2 \in \text{Union}, \quad \sigma \vdash t_1 \xrightarrow{\text{Intersection}} \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \quad \sigma \vdash t_2 \xrightarrow{\text{Union}} \hat{\mathrm{T}}_2, \gamma_2, \kappa_2, \Xi_2}{\sigma \vdash t_1 \; \texttt{+} \; t_2 \xrightarrow{\text{Union}} \hat{\mathrm{T}}_1 \sqcup \hat{\mathrm{T}}_2, \gamma_1 \sqcup \gamma_2, \kappa_1 \sqcup \kappa_2, \Xi_1 \sqcup \Xi_2} \tag{4.124}$$

Result of the interpretation of the import specification is a tuple $(\mathrm{T}, \gamma, \kappa, \Xi)$ which provides the initial sets for the interpretation of the set Style T_0, γ_0, κ_0, and Ξ_0, which are then supplemented by the style body $b \in$ Style-Body according to the rule template

$$\frac{id \in \text{Identifier}, \quad id \notin \operatorname{dom}(\sigma), \quad i \in \text{Import-Spec}, \quad \sigma \vdash i \xrightarrow{\text{Import-spec}} \hat{\mathrm{T}}_0, \gamma_0, \kappa_0, \Xi_0, \quad b \in \text{Style-Body}, \quad \hat{\mathrm{T}}_0, \gamma_0, \kappa_0, \Xi_0, \emptyset, \emptyset \vdash b \xrightarrow{\text{Style-Body}} \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \mathcal{V}, \mathcal{C}}{\sigma \vdash \texttt{style} \; id_0 \; i \; \texttt{\{} \; b \; \texttt{\}} \xrightarrow{\text{Style}} \sigma[id : (\mathrm{s}\ (\hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \operatorname{cls}_{\mathcal{V}}(\Xi_1)))]} \tag{4.125}$$

Interface type variables $v \in \mathcal{V}$ or literal constants $c \in \mathcal{C}$ are not collected from other styles, instead the respective sets are supplied as empty in the environment under which the style body is interpreted. Also, similar to the local case of component or connector kinds (*sec.* 4.2.14), type variables $v \in \mathcal{V}$ are eliminated within the set of interface type constraints Ξ_1 resulting from the interpretation of the style body after calculating the transitive closure.

4.3 The Module Tier

A CALM module is a library of types of component and interface kinds. Each module is defined within precisely one style, which means all kinds which are populated by the module are drawn from that specific style. The base for creating a module is therefore a kind mapping $\kappa \in \mathrm{K}$ with respective typing constraint set Ξ. A complete module is a pair of the module constructor $\mathbf{m}$ and the module structure, which is a pair of the attribute type set (recorded in its kind level form) $\bar{\mathrm{T}}$ and the mapping which records the types of architectural elements (of interface and component kinds) of this module $\psi \in \Psi$. An overview of the module elements is given in Table 4.2.

4.3.1 Constants

Next to the actual type declarations, constants $t \in$ Constant are the one top-level syntactical entity within a CALM module body. Much like the attributes on kinds on the style tier, which have to be partially valuated in the transition from the meta-kind to the kind level (see *sec.* 4.2.1, *sec.* 4.2.10, *sec.* 4.2.11), attributes on types have to be transferred from the kind to the type level through a partial valuation. Again, similar to the style tier, constants can be used to facilitate this valuation. Their syntax is given through the rule template

$$\frac{id \in \text{Identifier}, \quad t \in \text{Type-Spec}, \quad v \in \text{Literal}}{id \; \texttt{:} \; t \; \texttt{=} \; v \in \text{Constant}} \tag{4.126}$$

Their interpretation, given through the rule template

$$\frac{id \in \text{Identifier}, \quad id \notin \operatorname{dom}(\mathcal{C}), \quad t \in \text{Type-Spec}, \quad \bar{\mathrm{T}} \vdash t \xrightarrow{\text{Type-Spec}} \bar{\tau}, \quad v \in \text{Literal}, \quad \bar{\mathrm{T}}, \bar{\tau}, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \tau}{\bar{\mathrm{T}}, \mathcal{C} \vdash id \; \texttt{:} \; t \; \texttt{=} \; v \xrightarrow{\text{Constant}} \mathcal{C}[id : \tau]} \tag{4.127}$$

adds the constants to the module-wide set $\mathcal{C}$ of constants.

4.3.2 Type declarations for architecture elements

Core of a CALM module is the declaration of types of the kinds of architectural elements defined by the module's style. Each type declaration starts with a nominal reference to the kind within which the type is declared, followed by the name of the type. An identifier list of includes adds the structure of previously

Attribute types	$\bar{\tau} \in \bar{\mathrm{T}}$	Kind level attribute type set, and
	$\bar{\mathrm{T}} \in \bar{\mathcal{A}}$	set of all kind level attribute type sets.
	$\tau \in \mathrm{T}$	Type level attribute type set, and
	$\mathrm{T} \in \mathcal{A}$	set of all type level attribute type sets.
Attribute maps of types	$\alpha \in \mathrm{A}$	$\mathrm{A} = \mathsf{Identifier} \rightharpoonup (\mathrm{a}\ \mathrm{T})$
Ports on types	$\pi \in \Pi$	$\Pi = \mathsf{Identifier} \rightharpoonup (\mathsf{Identifier}^{*}\ (\{\mathbf{uses}, \mathbf{provides}\} \times \mathsf{Identifier}^{\dagger} \times \mathcal{Q}))$
Type mapping, recording interface and component types	$\psi \in \Psi$	$\Psi = \mathsf{Identifier} \rightharpoonup (\mathsf{Identifier}^{\ddagger}\ \mathrm{A}) \times (\mathsf{Identifier}^{\S}\ (\mathrm{A} \times \Pi))$
Type constraints	$\xi \in \Xi$	$\xi = \mathrm{type}(\mathit{kind}\texttt{.}\mathit{port\text{-}option}_1) \sim \mathrm{type}(\mathit{kind}\texttt{.}\mathit{port\text{-}option}_2)$
Module mapping, a library of CALM architectural elements (components and interfaces).	$\mu \in \mathrm{M}$	$\mathrm{M} = \mathsf{Identifier} \rightharpoonup (\mathbf{m}\ (2^{\bar{\mathrm{T}}} \times \Psi))$

[*] The identifier denotes a port-option name which doubles as *keyword* for introducing ports.
[†] The identifier needs to denote an interface type within the required kind.
[‡] The identifier is an interface kind name.
[§] The identifier is a component kind name.

Table 4.2: CALM module structure sets and symbols

defined types of the same kind. Syntactically, the type declaration is given by the set Type, which is defined in the rule template

$$\frac{id_1, id_2 \in \mathsf{Identifier}, \quad i \in \mathsf{Include}, \quad b \in \mathsf{Type\text{-}Body}}{id\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \in \mathsf{Type}} \tag{4.128}$$

Note that this syntax does not explicitly distinguish the category of the types' kind (*i. e.*, whether the type is declared within a component kind or interface kind), as it is rather meant to emphasize the kinds themselves. The distinction of the category is only given through the referenced kind's nature.

Types can include structure from previously declared types of the same kind. These are given through the identifiers representing their names. The list (if nonempty) is given with the keyword **include**, as defined by the rule templates

$$\frac{}{\epsilon \in \mathsf{Include}} \tag{4.129}$$

$$\frac{i \in \mathsf{Identifier\text{-}list}}{\mathbf{include}\ i \in \mathsf{Include}} \tag{4.130}$$

with the identifier list $i \in \mathsf{Identifier\text{-}list}$ being defined by the rule templates

$$\frac{id \in \mathsf{Identifier}}{id \in \mathsf{Identifier\text{-}list}} \tag{4.131}$$

$$\frac{id \in \mathsf{Identifier}, \quad i \in \mathsf{Identifier\text{-}list}}{id\texttt{,}i \in \mathsf{Identifier\text{-}list}} \tag{4.132}$$

The body of a type declaration is a (possibly empty) sequence of attribute valuations and port declarations, which have to comply with the attributes and port options given by the respective kind. The set Type-Body is given by the rule templates

$$\frac{}{\epsilon \in \mathsf{Type\text{-}Body}} \tag{4.133}$$

$$\frac{t_1, t_2 \in \mathsf{Type\text{-}Body}}{t_1\texttt{;}\ t_2 \in \mathsf{Type\text{-}Body}} \tag{4.134}$$

for an empty body and sequential composition, and

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Type-Body}} \quad (4.135) \qquad \frac{t \in \textsf{Port}}{t \in \textsf{Type-Body}} \quad (4.136)$$

for attribute valuations and port declarations. Attribute valuations in Attribute-Valuation have the form

$$\frac{id \in \textsf{Identifier}, \quad v \in \textsf{Literal}}{id = v \in \textsf{Attribute-Valuation}} \tag{4.137}$$

while port declarations in Port are given through the rule template

$$\frac{id_1, id_2, id_3 \in \textsf{Identifier}}{id_1\ id_2\ \mathbf{:}\ id_3 \in \textsf{Port}} \tag{4.138}$$

While syntactically the declaration of a type within an interface kind is equal to the declaration of a type within a component kind, the interpretation of both differs in details. Main difference is the fact that interface kinds do not offer port options. Depending on the category of the kind referenced in the type head, the interpretation of a type declaration splits into two rule templates, which are

$$\frac{\begin{array}{c} id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad id_2 \notin \mathrm{dom}(\psi), \quad i \in \textsf{Include}, \\ \psi, id_1 \vdash i \xrightarrow{\textsf{Include}} \alpha_0, \emptyset, \quad b \in \textsf{Type-Body}, \quad \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \emptyset, \emptyset, \mathcal{C} \vdash b \xrightarrow{\textsf{Type-Body}} \alpha_1, \emptyset \\ \mathcal{A}_t^k(\bar{\alpha}) = \alpha_2, \quad \alpha = \alpha_1 \cup \alpha_2, \quad \mathcal{Q}_a^k(\bar{\alpha}, \alpha) \end{array}}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}, \Xi \vdash id\ id\ i\ \{\ b\ \} \xrightarrow{\textsf{Type}} \psi[id_2 : (id_1\ \alpha_2)]} \tag{4.139}$$

for interface kinds (the port option mapping being empty), and

$$\frac{\begin{array}{c} id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbf{c}\ (\bar{\alpha}, \bar{\pi})) \in \kappa, \quad id_2 \notin \mathrm{dom}(\psi), \quad i \in \textsf{Include}, \\ \psi, id_1 \vdash i \xrightarrow{\textsf{Include}} \alpha_0, \pi_0, \quad b \in \textsf{Type-Body}, \quad \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi_0, \mathcal{C} \vdash b \xrightarrow{\textsf{Type-Body}} \alpha_1, \pi \\ \mathcal{A}_t^k(\bar{\alpha}) = \alpha_2, \quad \alpha = \alpha_1 \cup \alpha_2, \quad \mathcal{Q}_a^k(\bar{\alpha}, \alpha), \quad \mathcal{Q}_p^k(\bar{\pi}, \pi), \quad \mathcal{K}(\Xi, \pi) \end{array}}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}, \Xi \vdash id\ id\ i\ \{\ b\ \} \xrightarrow{\textsf{Type}} \psi[id_2 : (id_1\ (\alpha_2, \pi_2))]} \tag{4.140}$$

for component kind, which have a port option mapping $\bar{\pi}$. Similar to the transition between meta-kinds and kinds, attributes which do not require valuation at the type level are copied from the kind level by the function

$$\mathcal{A}_t^k : \bar{\mathrm{A}} \to \mathrm{A}.$$

The attribute map resulting from the partial valuation within the body together with the attribute map copied from the kind level have to be complete in as much as all attributes defined on the kind have to be handled properly. This is checked by the predicate

$$\mathcal{Q}_a^k \subseteq \bar{\mathrm{A}} \times \mathrm{A}$$

A similar predicate checks the completeness of the declared ports within the type's port map π against the requirements given by the kind's port option map $\bar{\pi}$[3]

$$\mathcal{Q}_p^k \subseteq \bar{\Pi} \times \Pi$$

Finally, the interface types associated with the declared ports of the type's port map π have to comply to the typing constraints given by the constraint set Ξ. Recall that a constraint $\xi \in \Xi$ has the form

$$\xi = \mathrm{type}(\textit{kind}\,\mathbf{.}\,\textit{port-option}_1) \sim \mathrm{type}(\textit{kind}\,\mathbf{.}\,\textit{port-option}_2)$$

where $\sim$ is either $=$, $\leq$, or $\geq$, with $=$ denoting type equality $\leq$ and $\geq$ denoting directed (asymmetric) type compatibility (*e. g.*, subtyping, see *sec.* 4.3.4). These constraints have to be checked for consistency with expressions given through the declaration of ports, which have the form

$$\mathrm{type}(\textit{type}\,\mathbf{.}\,\textit{port}) = \{t_0\},$$

[3]The definition of both predicates is straightforward.

where *type* is a type within a component kind kind_0 and *port* is a port declared within one of the kinds port options $\textit{port-option}_0$, so that

$$t_0 \in \mathrm{type}(\textit{kind}_0\texttt{.}\textit{port-option}_0).$$

It is apparent, that this way of using type constraints to restrict the use of interface types on ports (and roles) has limited expressiveness, nevertheless it proved sufficient in all component systems modeled so far. The predicate

$$\mathcal{K} \subseteq 2^{\Xi} \times \Pi$$

embodies the validation of a type's port map π against the interface typing constraints $\xi \in \Xi$.

Before evaluation of the body, a type can be complemented with the structures of existing types of the same kind. The list of inclusions $i \in \textsf{Include}$ is interpreted through the rule templates

$$\frac{}{\psi, id \vdash \epsilon \xrightarrow{\textsf{Include}} \emptyset, \emptyset} \tag{4.141}$$

in case of no inclusions and

$$\frac{i \in \textsf{Identifier-list}, \quad \psi, id \vdash i \xrightarrow{\textsf{Include}} \alpha, \pi}{\psi, id \vdash \texttt{include}\ i \xrightarrow{\textsf{Include}} \alpha, \pi} \tag{4.142}$$

for a nonempty list of inclusions. In both rules, id is the name of the kind within which the included types live. The identifier list $i \in \textsf{Identifier-list}$ is interpreted through the rule templates

$$\frac{id \in \textsf{Identifier}, \quad id : (id_i\ \alpha) \in \psi}{\psi, id_i \vdash id \xrightarrow{\textsf{Include}} \alpha, \emptyset} \tag{4.143}$$

for single interface types,

$$\frac{id \in \textsf{Identifier}, \quad id : (id_c\ (\alpha, \pi)) \in \psi}{\psi, id_c \vdash id \xrightarrow{\textsf{Include}} \alpha, \pi} \tag{4.144}$$

for single component types, and

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}, \quad id : (id_i\ \alpha_1) \in \psi, \quad \psi \vdash i \xrightarrow{\textsf{Include}} \alpha_2, \emptyset \\ \mathrm{dom}(\alpha_1) \cap \mathrm{dom}(\alpha_2) = \emptyset \end{array}}{\psi, id_i \vdash id\texttt{,}\, i \xrightarrow{\textsf{Include}} \alpha_1 \cup \alpha_2, \emptyset} \tag{4.145}$$

and

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}, \quad id : (id_c\ (\alpha_1, \pi_1)) \in \psi, \quad \psi \vdash i \xrightarrow{\textsf{Include}} \alpha_2, \pi_2, \\ \mathrm{dom}(\alpha_1) \cap \mathrm{dom}(\alpha_2) = \emptyset, \quad \mathrm{dom}(\pi_1) \cap \mathrm{dom}(\pi_2) = \emptyset \end{array}}{\psi, id_c \vdash id\texttt{,}\, i \xrightarrow{\textsf{Include}} \alpha_1 \cup \alpha_2, \pi_1 \cup \pi_2} \tag{4.146}$$

for sequences of interface or component types respectively. Main precondition is that the domains of the included attribute/port maps are disjoint to eliminate ambiguities. Result of the evaluation are initial attribute and port maps α and π which are extended by the attribute valuations and port declarations in the type body.

The type body is interpreted under the kind level type set $\bar{\mathrm{T}}$ which is needed to determine the correctness of attribute valuations, the module's type map ψ as defined so far, the kind level attribute and port option maps of the type's kind $\bar{\alpha}$ and $\bar{\pi}$, the type's initial type level attribute and port map α and π, and the global set of constants $c \in \mathcal{C}$. Result of the interpretation is a pair $\alpha', \pi' \in \mathrm{A}, \Pi$ which contains the type's final attribute and port mapping. The interpretation is defined by the rule templates

$$\frac{}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha, \bar{\pi}, \pi, \mathcal{C} \vdash \epsilon \xrightarrow{\textsf{Type-Body}} \alpha, \pi} \tag{4.147}$$

for an empty body, and

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Type-Body}, \quad \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi_0, \mathcal{C} \vdash t_1 \xrightarrow{\textsf{Type-Body}} \alpha_1, \pi_1 \\ \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_1, \bar{\pi}, \pi_1, \mathcal{C} \vdash t_2 \xrightarrow{\textsf{Type-Body}} \alpha_2, \pi_2 \end{array}}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi_0, \mathcal{C} \vdash t_1\texttt{;}\ t_2 \xrightarrow{\textsf{Type-Body}} \alpha_2, \pi_2} \tag{4.148}$$

for sequential composition of type body elements, with

$$\frac{t \in \mathsf{Attribute\text{-}Valuation}, \quad \bar{\mathrm{T}}, \bar{\alpha}, \alpha_0, \mathcal{C} \vdash t \xrightarrow{\mathsf{Attribute\text{-}Valuation}} \alpha_1}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi, \mathcal{C} \vdash t \xrightarrow{\mathsf{Type\text{-}Body}} \alpha_1, \pi} \tag{4.149}$$

for attribute valuations $t \in \mathsf{Attribute\text{-}Valuation}$, and

$$\frac{t \in \mathsf{Port}, \quad \psi, \bar{\pi}, \pi_0 \vdash t \xrightarrow{\mathsf{Port}} \pi_1}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha, \bar{\pi}_0, \pi, \mathcal{C} \vdash t \xrightarrow{\mathsf{Type\text{-}Body}} \alpha_1, \pi_1} \tag{4.150}$$

for port declarations in $t \in \mathsf{Port}$. The interpretation of a single attribute valuation on a type when defining it within a kind is closely analogous to that on a kind when exporting it from a meta-kind. The rule template

$$\frac{id \in \mathsf{Identifier}, \quad id : \bar{\tau} \in \bar{\alpha}, \quad id \notin \mathrm{dom}(\alpha), \quad v \in \mathsf{Literal}, \quad \bar{\mathrm{T}}, \bar{\tau}, \mathcal{C} \vdash v \xrightarrow{\mathsf{Literal}} \tau}{\bar{\mathrm{T}}, \bar{\alpha}, \alpha, \mathcal{C} \vdash id = v \xrightarrow{\mathsf{Attribute\text{-}Valuation}} \alpha[id : \tau]} \tag{4.151}$$

defines the interpretation relation. It is a precondition that the attribute is not yet listed in the type's attribute mapping to avoid overwriting of already defined values. A single port declaration $t \in \mathsf{Port}$ is interpreted according to the rule template

$$\frac{id_1, id_2, id_3 \in \mathsf{Identifier}, \quad id_1 : (\mathbf{p}\ (p, k, q_1, q_2)) \in \bar{\pi}, \quad id_2 \notin \mathrm{dom}(\pi), \quad id_3 : (k\ \alpha) \in \psi}{\psi, \bar{\pi}, \pi \vdash id_1\ id_2\ \mathbf{:}\ id_3 \xrightarrow{\mathsf{Port}} \pi[id_2 : (id_1\ (p, id_3, q_2))]} \tag{4.152}$$

Recall that the correct handling of the multiplexity q_1 of the port option $id_1 : (\mathbf{p}\ (p, k, q_1, q_2)) \in \bar{\pi}$ is a property which, instead of the single port, applies to the whole port map π of the type, and is checked for the complete type by the predicate $\mathcal{Q}_p^k \subseteq \bar{\Pi} \times \Pi$. A summary of the genesis of a port from the meta-kind's port option is given in Table 4.3.

$\hat{\pi} \vdash id_o : (\mathbb{p}\quad (p, \textit{interface-meta-kind}, q_1, q_2))$	The original port option on a meta-kind with parity p, interface meta-kind, multiplicity q_1 and multiplexity q_2.
$\downarrow \qquad\qquad \triangledown$	
$\bar{\pi} \vdash id_o : (\mathbf{p}\quad (p, \quad \textit{interface-kind}, \quad q_1, q_2))$	On the kind level, the shape (meta-kind) is replaced by a concrete interface kind.
$\searrow \qquad\qquad \triangledown$	
$\pi \vdash id_p : (id_o\ (p, \quad \textit{interface-type}, \quad q_2))$	The type declares a particular port within the port option id_o. The multiplicity is no longer relevant.

Table 4.3: Port genesis from meta-kind port option to concrete port on type

4.3.3 Module head and element inclusion

Together, constant definitions $c \in \mathsf{Constant}$ and type declarations $t \in \mathsf{Type}$ make the body of a CALM module. Syntactically, the complete module is given by the set Module, which is defined by the rule template

$$\frac{id_0, id_1 \in \mathsf{Identifier}, \quad i \in \mathsf{Input}, \quad b \in \mathsf{Module\text{-}Body}}{\texttt{module}\ id_0\ \texttt{of}\ id_1\ i\ \{\ b\ \} \in \mathsf{Module}} \tag{4.153}$$

where id_0 is the name of the module, id_1 is the style within which the module is defined, i is a list of module names of the modules which are included into the current, and b is the body. The interpretation of elements of Module revolves around looking up the style denoted by id_1 to obtain the set of kinds which can be used to declare types in the module. The rule template

$$\frac{\begin{array}{c} id_0, id_1 \in \mathsf{Identifier}, \quad id_0 \notin \mathrm{dom}(\mu), \quad id_1 : (\mathbf{s}\ (\hat{\mathrm{T}}, \gamma, \kappa, \Xi)) \in \sigma, \quad \mathcal{T}_k^m(\hat{\mathrm{T}}) = \bar{\mathrm{T}}_0, \\ i \in \mathsf{Input}, \quad \mu \vdash i \xrightarrow{\mathsf{Input}} \bar{\mathrm{T}}_1, \psi_0, \quad id_1 \mathrel{-\!\!\!\models} \psi_0, \quad \bar{\mathrm{T}} = \bar{\mathrm{T}}_0 \cup \bar{\mathrm{T}}_1 \\ b \in \mathsf{Module\text{-}Body}, \quad \bar{\mathrm{T}}, \kappa, \psi_0, \emptyset, \Xi \vdash b \xrightarrow{\mathsf{Type\text{-}Body}} \psi_1, \mathcal{C} \end{array}}{\sigma, \mu \vdash \texttt{module}\ id_0\ \texttt{of}\ id_1\ i\ \{\ b\ \} \xrightarrow{\mathsf{Module}} \mu[id_0 : (\mathbf{m}\ (\bar{\mathrm{T}}, \psi))]} \tag{4.154}$$

defines the interpretation of a module under a given style map $\sigma \in \Sigma$ and an existing module mapping $\mu \in \mathrm{M}$.

In this rule template, the function

$$\mathcal{T}_k^m : \hat{\mathcal{A}} \to \bar{\mathcal{A}}$$

translates the meta-kind level attribute type set $\hat{\mathrm{T}} \in \hat{\mathcal{A}}$ into the respective kind level attribute type set $\bar{\mathrm{T}} \in \bar{\mathcal{A}}$.[4] An additional attribute type set $\bar{\mathrm{T}}_1$ and an initial map of architectural element types ψ_0 are calculated from the list of input modules i.

It is central to the idea of model *migration* in CALM that modules, although they have to be defined within a specific style, are not bound to that style.[5] Instead, since styles can "inherit" a set of kinds from previously defined styles (*sec.* 4.2.16), a module complies to any style which features the complete set of kinds used within the module. The expression

$$id_1 \mathrel{+\!\!\models} \psi_0$$

checks that the type map ψ_0 indeed only contains types of kinds within the kind mapping κ of the style denoted by id_1.

Since the modules included through the input list i are not necessarily defined within the same style as the current module but only have to comply to that style, the attribute type set $\bar{\mathrm{T}}_1$ collected from the input list is not necessarily equal to the attribute type set $\bar{\mathrm{T}}_0$ computed from the module's style. Therefore, the union $\bar{\mathrm{T}}$ of both is taken to evaluate the module body b.

The input list $i \in \textsf{Input}$ is either empty or an identifier list preceded by the keyword **`input`**. The set $\textsf{Input}$ is defined by the rule templates

$$\frac{}{\epsilon \in \textsf{Input}} \qquad (4.155) \qquad\qquad \frac{i \in \textsf{Identifier-list}}{\texttt{input}\ i \in \textsf{Input}} \qquad (4.156)$$

And interpreted according to the rule templates

$$\frac{}{\mu \vdash \epsilon \xrightarrow{\textsf{Input}} \emptyset, \emptyset} \qquad (4.157) \qquad\qquad \frac{i \in \textsf{Identifier-list}, \quad \mu \vdash i \xrightarrow{\textsf{Input}} \bar{\mathrm{T}}, \psi}{\mu \vdash \texttt{input}\ i \xrightarrow{\textsf{Input}} \bar{\mathrm{T}}, \psi} \qquad (4.158)$$

where the identifier list $i \in \textsf{Identifier-list}$, syntactically given by the rule templates

$$\frac{id \in \textsf{Identifier}}{id \in \textsf{Identifier-list}} \qquad (4.159) \qquad\qquad \frac{id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}}{id\,\texttt{,}\,i \in \textsf{Identifier-list}} \qquad (4.160)$$

is interpreted through the rule templates

$$\frac{id \in \textsf{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}, \psi)) \in \mu}{\mu \vdash id \xrightarrow{\textsf{Input}} \bar{\mathrm{T}}, \psi} \qquad (4.161)$$

for the singleton list, and

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}_0, \psi_0)) \in \mu, \quad i \in \textsf{Identifier-list}, \quad \mu \vdash i \xrightarrow{\textsf{Input}} \bar{\mathrm{T}}_1, \psi_1 \\ \mathrm{dom}(\psi_0) \cap \mathrm{dom}(\psi_1) = \emptyset \end{array}}{\mu \vdash id\,\texttt{,}\,i \xrightarrow{\textsf{Input}} \bar{\mathrm{T}}_0 \cup \bar{\mathrm{T}}_1, \psi_0 \cup \psi_1} \qquad (4.162)$$

for a sequence of identifiers.

Finally, the module body is given through the syntactic set $\textsf{Module-Body}$. The set is defined through four rule templates, which are

[4]Note that the details of the attribute types are part of a separate effort. Nevertheless the transition mostly involves (partially) eliminating types with binding time specifier **`STYLE`**.

[5]For detailed discussion of model migration see Chapter 6.

$$\frac{}{\epsilon \in \textsf{Module-Body}} \quad (4.163) \qquad \frac{t_1, t_2 \in \textsf{Module-Body}}{t_1\,\texttt{;}\ t_2 \in \textsf{Module-Body}} \quad (4.164)$$

for an empty body and for sequential combination of body elements, and

$$\frac{t \in \textsf{Type}}{t \in \textsf{Module-Body}} \quad (4.165) \qquad \frac{t \in \textsf{Constant}}{t \in \textsf{Module-Body}} \quad (4.166)$$

for type declarations and constant definitions. The module body is interpreted under the attribute type set $\bar{\mathrm{T}}$, the kind mapping of the module's style κ, an initial type mapping ψ, a set of constants $\mathcal{C}$, and the style's interface typing constraint set Ξ. The interpretation is defined by the rule templates

$$\frac{}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}, \Xi \vdash \epsilon \xrightarrow{\textsf{Module-Body}} \psi, \mathcal{C}} \quad (4.167)$$

for an empty module body,

$$\frac{t_1, t_2 \in \textsf{Module-Body}, \quad \bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}_0, \Xi \vdash t_1 \xrightarrow{\textsf{Module-Body}} \psi_1, \mathcal{C}_1 \qquad \bar{\mathrm{T}}, \kappa, \psi_1, \mathcal{C}_1, \Xi \vdash t_1 \xrightarrow{\textsf{Module-Body}} \psi_2, \mathcal{C}_2}{\bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}_0, \Xi \vdash t_1\,\texttt{;}\ t_2 \xrightarrow{\textsf{Module-Body}} \psi_2, \mathcal{C}_2} \quad (4.168)$$

for sequentially composed body elements,

$$\frac{t \in \textsf{Type}, \quad \bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\textsf{Type}} \psi_1}{\bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\textsf{Module-Body}} \psi_1, \mathcal{C}} \quad (4.169)$$

for type declarations, and

$$\frac{t \in \textsf{Constant}, \quad \bar{\mathrm{T}}, \mathcal{C}_0 \vdash \xrightarrow{\textsf{Constant}} \mathcal{C}_1}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}_0, \Xi \vdash t \xrightarrow{\textsf{Module-Body}} \psi, \mathcal{C}_1} \quad (4.170)$$

for constant definitions.

4.3.4 Insufficiency of Structural Subtyping

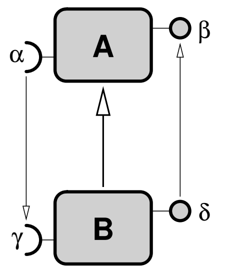
The fact that all interface and component types are essentially collections of constituents would suggest defining a subtyping relation through inclusion in analogy to object oriented subtyping through inheritance. For example, an obvious definition for a subtyping relation $\leq$ on component types would be

$$c_1 : (k\ (\alpha_1, \pi_1)) \leq c_2 : (k\ (\alpha_2, \pi_2)) \quad \Leftrightarrow \quad \alpha_1 \subseteq \alpha_2, \pi_1 \subseteq \pi_2.$$

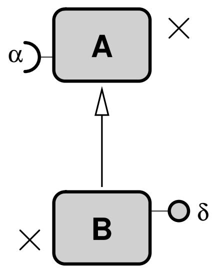
Nevertheless in the case of architecture models, this definition of subtyping does not capture a notion of directed type replacement compatibility as the term "subtyping" intuitively would require.

As a counterexample, consider a typical component framework setting with two kinds of connectors, a synchronous data connector and an asynchronous event connector in a data-pull, event-push architecture internally realized through remote method calls (*e. g.*, CCM, Bold Stroke/PRiSM, *etc.*). Data-pull means that the data is transported as the return value of a method call, which means that data flow and control flow have opposite directions. Event-push means that the event is transported as the argument of a method call, which means that data and control flow have the same direction.

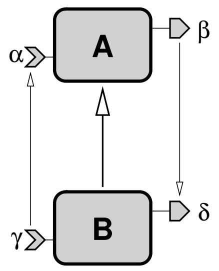
Figure 4.2 illustrates three cases within a data-pull, event-push architecture in which a component B can be considered subtype of a component A in as much as B could replace A in an environment fitting for A. Figure 4.2(a) shows A having two data interfaces α and β, with α having parity uses and β having parity provides. Therefore, α represents a method call, while β represents a method implementation. Component B features the same setup of interfaces, γ being an interface user and δ a provider. For B to be a subtype of A, any method call made on β has to be handled by δ too, which means that the type of the interface associated with δ has to be a subtype of that of β (covariance). Nevertheless, on the user side the situation reverses, as calls made by γ on the environment cannot exceed those made by α. Moreover, as Figure 4.2(b)



(a) CCM Data-interface



(b) Subtyping by addition/subtraction of ports



(c) CCM Event-interface

Figure 4.2: Inheritance with co- and contravariance

illustrates, B might not have any port matching α on A, as this means less requirements to the environment, while B can introduce ports such as δ without corresponding port on A, as this means more service to the environment. Finally, for event-push ports, where the data flows in the same direction as the control, the covariance/contravariance relation between data provider and data user reverses again.[6]

For the reasons outlined above, CALM does currently not have a general subtyping policy, nevertheless the type system does naturally allow to identify component types which can replace others in given environments.

4.4 The scenario tier

As the style tier defines kinds of architectural elements and the module tier fills these kinds with types, the CALM scenario tier instantiates the kinds and arranges the instances into a topology called *assembly*, which abstractly represents a system. Similar to the modules, there has to be at least one style which the assembly complies to.[7] The elements for instantiation are drawn from a map of types $\psi \in \Psi$ which is composed from explicitly listed modules and from a map of kinds $\kappa \in \mathrm{K}$ which is constructed from the collected set of styles which all included modules comply to.

The mechanism for instantiation is specific to each category of architectural elements. Types of component kinds are instantiated explicitly by referencing the type found in the type mapping ψ. For connector kinds no types are declared, hence the instances are manufactured directly from the kind found in κ together with an on-the-fly produced type for each connector instance. Finally, for each port/role with parity **provides** an interface instance is created implicitly with the fully qualified name of the port/role which provides it. To assemble the instances of architecture elements into networks of components and connectors, ports and roles are linked together through link clauses $\lambda \in \Lambda$ which associate one role instance with a list of port instances.[8]

An overview of the main elements of the CALM scenario tier is given in Table 4.4.

4.4.1 Constants

Analogous to the style and module tier, constants can be defined on the scenario tier to more conveniently valuate attributes. The syntactic set Constant is defined similar to the constants of the style and module tier by the rule template

$$\frac{id \in \text{Identifier}, \quad t \in \text{Type-Spec}, \quad v \in \text{Literal}}{id \;\mathbf{:}\; t = v \in \text{Constant}} \tag{4.171}$$

[6]It has therefore been proposed to assign the parity of a port always according to the control flow direction. Unfortunately, this solution is often unintuitive, and it does not handle the case where control flow is bidirectional as in combined event/command interfaces in nesC.

[7]For a discussion of *compliance*, see Chapter 6.

[8]Note that this is not a one-to-many association as ports can occur in multiple link clauses.

Attribute values	$\dot{\tau} \in \dot{\mathrm{T}}$ $\dot{\mathrm{T}} \in \dot{\mathcal{A}}$	Instance level attribute value set, and set of all instance level attribute value sets.
Attribute maps of values	$\dot{\alpha} \in \dot{\mathrm{A}}$	$\dot{\mathrm{A}} =$ Identifier $\rightharpoonup (a\ \dot{\mathrm{T}})$
Roles on instances	$\dot{\rho} \in \dot{\mathrm{R}}$	$\dot{\mathrm{R}} =$ Identifier*
Ports on instances	$\dot{\pi} \in \dot{\Pi}$	$\dot{\Pi} =$ Identifier†
Value mapping, recording interface, component and connector instances	$\theta \in \Theta$	$\Theta =$ Identifier $\rightharpoonup$ (Identifier‡ $\dot{\mathrm{A}}$) $\times$ (Identifier§ $(\dot{\mathrm{A}} \times \dot{\Pi})$) $\times$ (Identifier¶ $(\dot{\mathrm{A}} \times \dot{\mathrm{R}})$)
Port list (set of ports on component instances)		$\vec{p} = \{(id_1, id_2) \mid id_1 : (id_c\ (\dot{\alpha}, \dot{\pi})) \in \theta,$ with $id_2 \in \dot{\pi}\}$
Link clause	$\lambda \in \Lambda$	$\lambda = id_1\,.\,id_2 : \vec{p}$ $\Lambda = \{id_1\,.\,id_2 : \vec{p} \mid id_1 : (id_l\ (\dot{\alpha}, \dot{\rho})) \in \theta,$ with $id_2 \in \dot{\rho},\ \vec{p}$ Port list$\}$
Scenario mapping, recording instances of component, connector, and interface kind types.	$\zeta \in \mathrm{Z}$	$\mathrm{Z} =$ Identifier $\rightharpoonup (\mathbf{z}\ (\theta, \Lambda))$

* Identifier is a connector type's role name (since connector types are not declared on the module tier, names for types and roles are assigned automatically by CADENA.)
† Identifier needs to be a component's port name.
‡ Identifier denotes an attribute type.
§ Identifier denotes an component type.
¶ Identifier denotes an connector type.

Table 4.4: CALM scenario structure sets and symbols

Further, the interpretation of Constant is analogous to that of style and module tier

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id \notin \mathrm{dom}(\mathcal{C}), \quad t \in \text{Type-Spec}, \quad \mathrm{T} \vdash t \xrightarrow{\text{Type-Spec}} \tau, \\ v \in \text{Literal}, \quad \mathrm{T}, \tau, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \dot{\tau} \end{array}}{\mathrm{T}, \mathcal{C} \vdash id\ \mathbf{:}\ t = v \xrightarrow{\text{Constant}} \mathcal{C}[id : \dot{\tau}]} \tag{4.172}$$

Note that terms $\dot{\tau} \in \dot{\mathrm{T}}$ represent values, no part or subterm requires further valuation (see *sec.* 4.2.1).

4.4.2 Attribute Valuations

Like the constants in Constant, attribute valuations in Attribute-Valuation also closely correspond to the analogous constructs on style and module tier. Namely, the set Attribute-Valuation is defined by the rule template

$$\frac{id \in \text{Identifier}, \quad v \in \text{Literal}}{id = v \in \text{Attribute-Valuation}} \tag{4.173}$$

and interpreted by the rule template

$$\frac{id \in \text{Identifier}, \quad id : \tau \in \alpha, \quad id \notin \mathrm{dom}(\dot{\alpha}), \quad v \in \text{Literal}, \quad \mathrm{T}, \tau, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \dot{\tau}}{\mathrm{T}, \alpha, \dot{\alpha}, \mathcal{C} \vdash id = v \xrightarrow{\text{Attribute-Valuation}} \dot{\alpha}[id : \dot{\tau}]} \tag{4.174}$$

Again, as with the valuations on the style and module tier, the attribute cannot already be recorded in the value level attribute map (*i. e.*, $id \notin \mathrm{dom}(\dot{\alpha})$) to prevent overwriting of already set values.

4.4.3 Instances of types of component kinds

Instances of component kinds are created by elements of the syntactic set Component, which is given through the rule template

$$\frac{id_0, id_1 \in \textsf{Identifier}, \quad b \in \textsf{Component-Body}}{id_0\ id_1\ \{\ b\ \} \in \textsf{Component}} \tag{4.175}$$

Here, id_0 references a component type, while id_1 is a new name for the component instance to be created. Elements of Component are interpreted under a (type-level) attribute type set T, a type mapping for architectural elements ψ, an instance mapping for architectural elements θ, and a set of constants $\mathcal{C}$. The interpretation is given through the rule template

$$\frac{\begin{array}{c} id_0, id_1 \in \textsf{Identifier}, \quad id_0 : (id_k\ (\alpha, \pi)) \in \psi, \quad id_1 \notin \theta_0, \quad b \in \textsf{Component-Body}, \\ \mathcal{A}_v^t(\alpha) = \dot{\alpha}_0, \quad \mathrm{T}, \psi, \theta_0, \alpha, \emptyset, \pi, \mathcal{C} \vdash b \xrightarrow{\textsf{Component-Body}} \theta_1, \dot{\alpha}_1, \quad \dot{\alpha} = \dot{\alpha}_0 \cup \dot{\alpha}_1, \quad \mathcal{Q}_a^t(\alpha, \dot{\alpha}) \\ \mathcal{P}_v^t(\pi) = \dot{\pi}, \quad \mathrm{IntGen}(\theta_1, \pi) = \theta_2, \quad \mathcal{I}[\theta_2, \pi] \end{array}}{\mathrm{T}, \psi, \theta_0, \mathcal{C} \vdash id_0\ id_1\ \{\ b\ \} \xrightarrow{\textsf{Component}} \theta_2[id_1 : (id_0\ (\dot{\alpha}, \dot{\pi}))]} \tag{4.176}$$

A lookup of the type name id_0 in the type map ψ yields the component type's map of attributes α and of ports π. Similar to the situation while declaring types of architectural elements within the module tier, attributes in α which do not require further valuation are copied into an instance level attribute map $\dot{\alpha}_0$ by the function

$$\mathcal{A}_v^t : \mathrm{A} \to \dot{\mathrm{A}}.$$

The union of $\dot{\alpha}_0$ with the instance level attribute map $\dot{\alpha}_1$ resulting from the interpretation of the component body forms the instance's attribute map $\dot{\alpha}$. The completeness of this map is checked by the predicate

$$\mathcal{Q}_a^t \subseteq \mathrm{A} \times \dot{\mathrm{A}},$$

again analogous to the declaration of types on the module tier.

Ports on the component instance are created automatically if they do not provide interfaces which need further attribute valuation (see *sec.* 4.4.4). The function

$$\mathcal{P}_v^t : \Pi \to \dot{\Pi}$$

generates an instance port map $\dot{\pi} \in \dot{\Pi}$ from a type port map $\pi \in \Pi$. The function is defined as

$$\begin{aligned} \mathcal{P}_v^t : \pi \mapsto \dot{\pi} \quad \text{iff} \quad & \forall id : (p, id_i, q) \in \pi.\ id \in \dot{\pi}, \text{ and} \\ & \forall id \in \dot{\pi}.\ id : (p, id_i, q) \in \pi \end{aligned}$$

(*i. e.*, for each port in the type level port map, $\mathcal{P}_v^t$ generates a corresponding port of the same name in the instance level port map). Table 4.5 illustrates the genesis of a port from the meta-kind level port option down to the component instance port.

As mentioned above, interface instances are not created on their own, but as part of the instantiation of a component type (or the creation/instantiation of a connector type) whose port (role) provides the interface. Canonically, this creation of provided interface instances happens within the component instance body. Nevertheless, oftentimes the creation can be fully implicit, namely whenever no further attribute valuation is needed on the created interface (*i. e.*, no attributes of the interface contains the binding time specifier **SCENARIO** within their types), the interface instance can be produced directly from its type. Formally, if a component with the name id_c of type id_t is created, all ports

$$id_p : (\texttt{provides}, id_i, q) \in \pi$$

with an interface type id_i, given within the type map ψ as

$$id_i : (id_k\ \alpha) \in \psi,$$

that does not need further attribute valuation, which is the case if

$$\mathcal{Q}_a^t(\alpha, \mathcal{A}_v^t(\alpha))$$

$\hat{\pi} \vdash id_o : (\mathbb{p}\quad (p,$ *interface-meta-kind*$, q_1, q_2))$	The original port option on a meta-kind with parity p, interface meta-kind, multiplicity q_1 and multiplexity q_2.
$\downarrow \qquad\qquad \triangledown$	
$\bar{\pi} \vdash id_o : (\mathbf{p}\quad (p,$ *interface-kind*$, q_1, q_2))$	On the kind level, the shape (meta-kind) is replaced by a concrete interface kind.
$\searrow \qquad\qquad \triangledown$	
$\pi \vdash id_p : (id_o\ (p,$ *interface-type*$, q_2))$	The type declares a particular port within the port option id_o. The multiplicity is no longer relevant.
$\downarrow \qquad\qquad \triangledown$	
$\dot{\pi} \vdash id_p$ *interface-instance*	The instance is created implicitly with the port type-level name id_p. The associated interface instance, if the port has the parity p = **provides**, is created automatically with the qualified port name $id_c\texttt{.}id_p$ (id_c = component instance name).

Table 4.5: Port genesis from meta-kind port option to port instance (extension of Table 4.3)

(*i. e.*, the copied attribute map $\mathcal{A}_v^t(\alpha)$ is complete *wrt.* α), the interface instance can be produced implicitly as

$$id_c\texttt{.}id_p : (id_i\ \mathcal{A}_v^t(\alpha)).$$

Therefore, CALM provides the function

$$\mathrm{IntGen} : (\Theta \times \Pi) \rightarrow \Theta$$

which for all ports $id_p : (\texttt{provides}, id_i, q) \in \pi$ adds the respective interface $this\texttt{.}id_p : (id_i\ \mathcal{A}_v^t(\alpha))$ to the given instance map θ_0 if $\mathcal{Q}_a^t(\alpha, \mathcal{A}_v^t(\alpha))$ holds.

Within the premises of the rule template for interpreting Component, the function IntGen is applied to the instance mapping θ_1, which results from the interpretation of the component body $b \in$ Component-Body. Nevertheless, the function only complements the interfaces within θ_1 whose attributes do not need further valuation. Therefore, all provided interfaces whose attributes do need further valuation have to be created within the component body b. Hence the predicate

$$\mathcal{I} \subseteq \Theta \times \Pi$$

ensures the presence of all provided interfaces within the instance map, namely

$$\mathcal{I}(\theta, \pi) \Leftrightarrow \forall id_p : (\texttt{provides}, id_i, q) \in \pi.\ this\texttt{.}id_p : (id_i\ \dot{\alpha}) \in \theta$$

(*i. e.*, $\mathcal{I}$ holds if for all ports with parity **provides** in π there is a corresponding interface in θ).

4.4.4 The component instance body

As mentioned above, the component instance body $b \in$ Component-Body valuates the component's instance level attributes (binding time specifier **SCENARIO**), and creates the interfaces not created by the function IntGen. The syntactic set Component-Body therefore consists of attribute valuations and interface specifications. It is defined by the rule templates

$$\frac{}{\epsilon \in \text{Component-Body}} \quad (4.177) \qquad \frac{t_1, t_2 \in \text{Component-Body}}{t_1\texttt{;}t_2 \in \text{Component-Body}} \quad (4.178)$$

for empty component bodies and sequential composition of component body elements, and

$$\frac{t \in \text{Attribute-Valuation}}{t \in \text{Component-Body}} \quad (4.179) \qquad \frac{t \in \text{Interface}}{t \in \text{Component-Body}} \quad (4.180)$$

for attribute valuations and interface specifications. Elements of Component-Body are interpreted under the attribute type set T, the type and instance maps ψ and θ, the type level attribute mapping α, an instance level attribute map $\dot\alpha$, the type's port map π and the set of constants $\mathcal{C}$. The interpretation of the empty body returns the given type map ψ and attribute map $\dot\alpha$ as given by the rule template

$$\frac{}{\mathrm{T}, \psi, \theta, \alpha, \dot\alpha, \pi, \mathcal{C} \vdash \epsilon \xrightarrow{\textsf{Component-Body}} \theta, \dot\alpha} \tag{4.181}$$

Sequential composition of body elements are interpreted according to the rule template

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Component-Body}, \quad \mathrm{T}, \psi, \theta_0, \alpha, \dot\alpha_0, \pi, \mathcal{C} \vdash t_1 \xrightarrow{\textsf{Component-Body}} \theta_1, \dot\alpha_1, \\ \mathrm{T}, \psi, \theta_1, \alpha, \dot\alpha_1, \pi, \mathcal{C} \vdash t_2 \xrightarrow{\textsf{Component-Body}} \theta_2, \dot\alpha_2 \end{array}}{\mathrm{T}, \psi, \theta_0, \alpha, \dot\alpha_0, \pi, \mathcal{C} \vdash t_1\texttt{;}t_2 \xrightarrow{\textsf{Component-Body}} \theta_2, \dot\alpha_2} \tag{4.182}$$

Attribute valuations $t \in$ Attribute-Valuation update the attribute mapping $\dot\alpha$ (*sec.* 4.4.2)

$$\frac{t \in \textsf{Attribute-Valuation}, \quad \mathrm{T}, \alpha, \dot\alpha_0, \mathcal{C} \vdash t \xrightarrow{\textsf{Attribute-Valuation}} \dot\alpha_1}{\mathrm{T}, \psi, \theta, \alpha, \dot\alpha_0, \pi, \mathcal{C} \vdash t \xrightarrow{\textsf{Component-Body}} \theta, \dot\alpha_1} \tag{4.183}$$

while the interpretation of elements of Interface adds interface instances to the instance mapping θ

$$\frac{t \in \textsf{Interface}, \quad \mathrm{T}, \psi, \theta_0, \pi, \mathcal{C} \vdash t \xrightarrow{\textsf{Interface}} \theta_1}{\mathrm{T}, \psi, \theta_0, \alpha, \dot\alpha, \pi, \mathcal{C} \vdash t \xrightarrow{\textsf{Component-Body}} \theta_1, \dot\alpha} \tag{4.184}$$

The main component body elements besides the attribute valuations are the interface specifications $t \in$ Interface. The set Interface is defined by the rule template

$$\frac{id \in \textsf{Identifier}, \quad a \in \textsf{Attributes}}{id \texttt{ \{ } a \texttt{ \} } \in \textsf{Interface}} \tag{4.185}$$

where the set Attributes is a list of attribute valuations as defined by the rule templates

$$\frac{}{\epsilon \in \textsf{Attributes}} \tag{4.186}$$

$$\frac{t_1, t_2 \in \textsf{Attributes}}{t_1\texttt{;}t_2 \in \textsf{Attributes}} \tag{4.187}$$

for empty[9] lists and sequential composition of attribute valuations, and

$$\frac{a \in \textsf{Attribute-Valuation}}{a \in \textsf{Attributes}} \tag{4.188}$$

for a specific attribute valuation. The syntactic set Interface is interpreted according to the rule template

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id : (id_p\ (\texttt{provides}, id_i, q)) \in \pi, \quad id_i : (id_k\ \alpha) \in \psi, \quad \mathcal{A}_v^t(\alpha) = \dot\alpha_0, \\ a \in \textsf{Attributes}, \quad \mathrm{T}, \alpha, \emptyset, \mathcal{C} \vdash t \xrightarrow{\textsf{Attributes}} \dot\alpha_1, \quad \dot\alpha = \dot\alpha_0 \cup \dot\alpha_1, \quad \mathcal{Q}_a^t(\alpha, \dot\alpha) \end{array}}{\mathrm{T}, \psi, \theta, \pi, \mathcal{C} \vdash id \texttt{ \{ } a \texttt{ \} } \xrightarrow{\textsf{Interface}} \theta[this.id : (id_i\ \dot\alpha)]} \tag{4.189}$$

which requires the respective port to have the parity **`provides`**. The interface's attribute mapping is partly copied from the type to the instance level by the function $\mathcal{A}_v^t(\alpha)$, and complemented by the results of the interpretation of $t \in$ Attributes. Again, the completeness is checked by the predicate $\mathcal{Q}_a^t(\alpha, \dot\alpha)$. The set Attributes is then interpreted according to the rules

$$\frac{}{\mathrm{T}, \alpha, \dot\alpha, \mathcal{C} \vdash \epsilon \xrightarrow{\textsf{Attributes}} \dot\alpha} \tag{4.190}$$

$$\frac{t_1, t_2 \in \textsf{Attributes}, \quad \mathrm{T}, \alpha, \dot\alpha_0, \mathcal{C} \vdash t_1 \xrightarrow{\textsf{Attributes}} \dot\alpha_1, \quad \mathrm{T}, \alpha, \dot\alpha_1, \mathcal{C} \vdash t_2 \xrightarrow{\textsf{Attributes}} \dot\alpha_2}{\mathrm{T}, \alpha, \dot\alpha_0, \mathcal{C} \vdash t_1\texttt{;}t_2 \xrightarrow{\textsf{Attributes}} \dot\alpha_2} \tag{4.191}$$

and

$$\frac{a \in \textsf{Attribute-Valuation}, \quad \mathrm{T}, \alpha, \dot\alpha_0, \mathcal{C} \vdash a \xrightarrow{\textsf{Attribute-Valuation}} \dot\alpha_1}{\mathrm{T}, \alpha, \dot\alpha_0, \mathcal{C} \vdash a \xrightarrow{\textsf{Attributes}} \dot\alpha_1} \tag{4.192}$$

[9]Note that the syntactic possibility of empty attribute valuation lists allows to explicitly introduce attributes which are again implicitly added by the function IntGen. This has to be observed when implementing this function.

4.4.5 Instances of connector kinds

The creation of connector instances differs from that of component instances in two major points. First, no types are declared within connector kinds, instead, connector instances are drawn directly from the kind, the type being created on-the-fly for each connector instance. Therefore, interface typing rules (which for the components are already manifested within the component types) have still to be observed when creating connector instances. Also, no roles can be created implicitly, since no type exists as a template. Second, connector instance definitions also introduce the topology of the assembly, which means that each connector not only specifies its roles and type and creates all provided interfaces, it also captures the connection between its roles and ports of existing component instances.

Instances of connector kinds together with their respective types are created through the syntactic set Connector, which is defined by the rule template

$$\frac{id \in \textsf{Identifier}, \quad b \in \textsf{Connector-Body}}{id_1 \ \texttt{\{} \ b \ \texttt{\}} \in \textsf{Connector}} \tag{4.193}$$

where the identifier id denotes a connector kind. Elements of the set Connector are interpreted under the type level attribute type set T, the kind, type, and instance mappings κ, ψ, and θ, the link clauses set Λ, the set of constants $\mathcal{C}$, and the set of interface typing constraints Ξ. The interpretation rule template for the set Connector is

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id : (\mathbf{1}\ (\bar{\alpha}, \bar{\rho})), \quad \mathcal{A}_t^k(\bar{\alpha}) = \alpha, \quad \mathcal{A}_v^t(\alpha) = \dot{\alpha}_0, \quad b \in \textsf{Connector-Body} \\ \mathrm{T}, \psi, \theta_0, \alpha, \emptyset, \bar{\rho}, \emptyset, \Lambda_0, \mathcal{C} \vdash b \xrightarrow{\textsf{Connector-Body}} \theta_1, \dot{\alpha}_1, \rho, \Lambda_1, \quad \dot{\alpha} = \dot{\alpha}_0 \cup \dot{\alpha}_1, \quad \mathcal{R}_v^t(\rho) = \dot{\rho}, \\ \mathcal{K}(\Xi, \rho), \quad \mathcal{Q}_r^k(\bar{\rho}, \rho), \quad \mathcal{Q}_a^k(\bar{\alpha}, \alpha), \quad \mathcal{Q}_a^t(\alpha, \dot{\alpha}), \quad id_l^t, id_l^v \in \textsf{Identifier}, \\ id_l^t \notin \mathrm{dom}(\psi), \quad id_l^v \notin \mathrm{dom}(\theta), \quad \mathit{this} = id_l^v \end{array}}{\mathrm{T}, \kappa, \psi, \theta, \Lambda_0, \mathcal{C}, \Xi \vdash id \ \texttt{\{} \ b \ \texttt{\}} \xrightarrow{\textsf{Connector}} \psi[id_l^t : (id\ (\alpha, \rho))], \theta[id_l^v : (id_l^t\ (\dot{\alpha}, \dot{\rho}))], \Lambda_1} \tag{4.194}$$

The identifier id is looked up in the kind map κ to retrieve its attribute and role option mappings $\bar{\alpha}$ and $\bar{\rho}$. The attribute map $\bar{\alpha}$ is then translated twice, through the functions

$$\mathcal{A}_t^k : \bar{\mathrm{A}} \to \mathrm{A}, \quad \text{and,} \quad \mathcal{A}_v^t : \mathrm{A} \to \dot{\mathrm{A}}$$

The resulting instance level attribute map $\dot{\alpha}_0$ is then complemented with the attribute map $\dot{\alpha}_1$ resulting from the interpretation of the body $b \in$ Connector-Body.[10] Further, the interpretation of the body yields an updated instance map θ which contains all interfaces provided by the connector, a type level role map, and an updated link clauses set Λ_1. The function

$$\mathcal{R}_v^t : \mathrm{R} \to \dot{\mathrm{R}}$$

is defined analogously to the function $\mathcal{P}_v^t$ (*sec.* 4.4.3), which means

$$\begin{array}{ll} \mathcal{R}_v^t : \rho \mapsto \dot{\rho} \quad \text{iff} & \forall id : (p, id_i, q) \in \rho.\ id \in \dot{\rho}, \ \text{ and} \\ & \forall id \in \dot{\rho}.\ id : (p, id_i, q) \in \rho \end{array}$$

(*i. e.*, for each role in the type level role map, $\mathcal{R}_v^t$ generates a corresponding role of the same name in the instance level role map). It yields the corresponding instance level role map $\dot{\rho}$ to the type level map ρ resulting from the interpretation of the body. Finally, four predicates check the completeness of the results, namely

$$\mathcal{K} \subseteq 2^{\Xi} \times \mathrm{R}$$

verifies that all typing constraints in Ξ are honored by the type level role mapping ρ,

$$\mathcal{Q}_r^k \subseteq \bar{\mathrm{R}} \times \mathrm{R}$$

[10]To be absolutely precise, the type level mapping α is not necessarily complete (the original mapping $\bar{\alpha}$ might contain types with the binding time specifier **`MODULE`**). For the formalism it seemed overly complicated to include type level valuations while every instance has its own type, although they are not formally forbidden on the meta-kind level. In Cadena nevertheless, the problem is solved through delayed valuation, which is discussed in Section 4.5.

checks whether the multiplicity constraints of the role options in $\bar{\rho}$ are correctly manifested in ρ,

$$\mathcal{Q}_a^k \subseteq \bar{\mathrm{A}} \times \mathrm{A}, \quad \text{and,} \quad \mathcal{Q}_a^t \subseteq \mathrm{A} \times \dot{\mathrm{A}}$$

confirm the completeness of the attribute mappings against their respective predecessor.

Since the connector instance and its type are both not named explicitly (only the kind has a name), CALM introduces random names as references for them. Specifically, the premises of the interpretation rule template contain the identifiers id_l^t and id_l^v, of which only two properties are required:

$$id_l^t, id_l^v \in \textsf{Identifier},$$

and

$$id_l^t \notin \mathrm{dom}(\psi), \quad id_l^v \notin \mathrm{dom}(\theta)$$

(*i. e.*, they are both in Identifier, but not in the domain of their respective type or instance mappings ψ and θ). Here, id_l^t is the (randomly selected) name of the newly created connector type, and id_l^v the (randomly selected) name of the newly created connector instance (therefore $this$ is set to id_l^v).

Further, note that unlike the instantiation of component types no implicit generation of interface instances is performed. This is, as mentioned above, due to the fact that no template for creating roles (connector type) exists a priori. As a result of the evaluation of an element of Connector the type map ψ is fitted with the new connector type, the instance map θ, which already contains the new interfaces, is complemented with the new connector instance, and the list of link clauses is updated.

4.4.6 The connector instance body

The body of a connector type/instance specification has two major elements, the attribute valuations, and the binding specifications. The syntactic set Connector-Body is defined by the rule templates

$$\frac{}{\epsilon \in \textsf{Connector-Body}} \quad (4.195) \qquad \frac{t_1, t_2 \in \textsf{Connector-Body}}{t_1 \mathbin{;} t_2 \in \textsf{Connector-Body}} \quad (4.196)$$

for an empty body and for sequential composition, and

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Connector-Body}} \quad (4.197) \qquad \frac{t \in \textsf{Binding}}{t \in \textsf{Connector-Body}} \quad (4.198)$$

for attribute valuations and bindings. Elements of Connector-Body are interpreted under the type level attribute type set T, type and instance mapping (ψ and θ), attribute maps for type and instance level, and role maps for kind and type level (α, $\dot{\alpha}$, $\bar{\rho}$, and ρ), the list of link clauses (Λ), and the set of constants ($\mathcal{C}$). The interpretation rule templates are straightforward:

$$\frac{}{\mathrm{T}, \psi, \theta, \alpha, \dot{\alpha}, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash t \xrightarrow{\textsf{Connector-Body}} \theta, \dot{\alpha}, \rho, \Lambda} \quad (4.199)$$

for an empty body,

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Connector-Body}, \quad \mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}_0, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t_1 \xrightarrow{\textsf{Connector-Body}} \theta_1, \dot{\alpha}_1, \rho_1, \Lambda_1, \\ \mathrm{T}, \psi, \theta_1, \alpha, \dot{\alpha}_1, \bar{\rho}, \rho_1, \Lambda_1, \mathcal{C} \vdash t_2 \xrightarrow{\textsf{Connector-Body}} \theta_2, \dot{\alpha}_2, \rho_2, \Lambda_2 \end{array}}{\mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}_0, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t_1 \mathbin{;} t_2 \xrightarrow{\textsf{Connector-Body}} \theta_2, \dot{\alpha}_2, \rho_2, \Lambda_2} \quad (4.200)$$

for sequential composition,

$$\frac{t \in \textsf{Attribute-Valuation}, \quad \mathrm{T}, \alpha, \dot{\alpha}_0, \mathcal{C} \vdash a \xrightarrow{\textsf{Attribute-Valuation}} \dot{\alpha}_1}{\mathrm{T}, \psi, \theta, \alpha, \dot{\alpha}_0, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash t \xrightarrow{\textsf{Connector-Body}} \theta, \dot{\alpha}_1, \rho, \Lambda} \quad (4.201)$$

for attribute valuations, and

$$\frac{\mathrm{T}, \psi, \theta_0, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t \xrightarrow{\textsf{Binding}} \theta_1, \rho_1, \Lambda_1}{\mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t \in\xrightarrow{\textsf{Connector-Body}} \theta_1, \dot{\alpha}, \rho_1, \Lambda_1} \quad (4.202)$$

for binding specifiers.

The purpose of binding specifiers $t \in \text{Binding}$ is threefold. First, it creates a role on the connector instance which immediately involves creating the respective role on the instance's type. The type level role map ρ records the new role.[11] Next, the binding creates an interface instance for every interface provided by the role. This explicit creation involves all provided interfaces of a connector, not just those which need attribute valuation (as seen with component instances). Finally, each thus created role is stored with a list of ports it connects to. Syntactically, the set Binding is defined by two role templates.

$$\frac{id \in \text{Identifier}, \quad l \in \text{Port-list}}{id = l \in \text{Binding}} \tag{4.203}$$

describes bindings for roles which either do not create interfaces (parity **uses**), or create interfaces which do not need further attribute valuation.

$$\frac{id \in \text{Identifier}, \quad l \in \text{Port-list}, \quad a \in \text{Attributes}}{id \ \{ \ a \ \} \ = l \in \text{Binding}} \tag{4.204}$$

describes bindings for roles which provide interfaces that do need further attribute valuation (manifested by $a \in \text{Attributes}$ in braces). The interpretation of elements of Binding involves the type level attribute type set T, type and instance mapping (ψ and θ), role maps for kind and type level ($\bar{\rho}$ and ρ), the list of link clauses (Λ), and the set of constants ($\mathcal{C}$). For roles with parity **uses**, the interpretation of Binding is given by the rule template

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (\mathbf{r}\ (\mathbf{uses}, id_k, q_1, q_2)) \in \bar{\rho}, \quad l \in \text{Port-list}, \\ \mathbf{provides}, \theta \vdash l \xrightarrow{\text{Port-list}} \vec{p}, id_i, \quad id_i : (id_k\ \alpha) \in \psi, \quad id_r^t \in \text{Identifier}, \quad id_r^t \notin \text{dom}(\rho) \end{array}}{\mathrm{T}, \psi, \theta, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash id = l \xrightarrow{\text{Binding}} \theta, \rho[id_r^t : (id\ (\mathbf{uses}, id_i, q_2))], \Lambda[this.id_r^t : \vec{p}]} \tag{4.205}$$

The identifier id is looked up in the role option map $\bar{\rho}$ of the component kind to retrieve parity, interface kind, multiplicity and multiplexity. This data is used to create the role in the connector type's role map ρ.

The list of port specifiers $l \in \text{Port-list}$, which defines the ports that will be linked to the role, is interpreted under the opposite parity (in this case: **provides**) than that of the role itself (in this case: **uses**) and the instance map θ. In result it yields a vector $\vec{p}$ of port specifiers (id_1, id_2) where id_1 is the name of a component instance and id_2 is a port of id_1, and an interface type id_i which has to be of the correct kind id_k. It is the type of all interfaces associated with all ports in the list. For the result of the interpretation a new role

$$id_r^t : (id\ (\mathbf{uses}, id_i, q_2))$$

with the (randomly selected) name id_r^t is added to the type level role map ρ, and a new clause

$$\lambda = this.id_r^t : \vec{p}$$

is added to the list of link clauses Λ to reflect the link between the new role and all ports in its port list.

For roles with parity **provides** the interpretation falls into two cases. First, roles which provide interfaces whose type does not require instance level valuation of attributes are interpreted according to the rule template

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (\mathbf{r}\ (\mathbf{provides}, id_k, q_1, q_2)) \in \bar{\rho}, \quad l \in \text{Port-list}, \\ \mathbf{uses}, \theta \vdash l \xrightarrow{\text{Port-list}} \vec{p}, id_i, \quad id_i : (id_k\ \alpha) \in \psi, \quad \mathcal{A}_v^t(\alpha) = \dot{\alpha}, \quad \mathcal{Q}_a^t(\alpha, \dot{\alpha}), \\ id_r^t \in \text{Identifier}, \quad id_r^t \notin \text{dom}(\rho), \quad this.id_r^t \notin \text{dom}(\theta) \end{array}}{\begin{array}{c} \mathrm{T}, \psi, \theta, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash id = l \xrightarrow{\text{Binding}} \theta[this.id_r^t : (id_i\ \dot{\alpha})], \\ \rho[id_r^t : (id\ (\mathbf{provides}, id_i, q_2))], \Lambda[this.id_r^t : \vec{p}] \end{array}} \tag{4.206}$$

In addition to the case of a role with parity **uses**, this rule includes a transition of the interface type's attributes to the instance level through the function

$$\mathcal{A}_v^t : \mathrm{A} \to \dot{\mathrm{A}}.$$

[11] The type level role map ρ is sufficient to create the instance level role map $\dot{\rho}$ implicitly, therefore the instance level role map is not built at the same time. See Section 4.4.5.

The fact that no further valuation is needed is reflected in the predicate

$$\mathcal{Q}^t_a \subseteq \mathrm{A} \times \dot{\mathrm{A}}$$

which verifies that the transition of attributes into $\dot{\alpha}$ without further valuation is in fact, complete with respect to α. Besides adding the new role to the role map ρ and the new link clause to the link list Λ, the new provided interface

$$this.id^t_r : (id_i\ \dot{\alpha})$$

is added to the instance map θ.

Second, roles which provide interfaces that have a type which does require instance level valuation of attributes are interpreted according to the rule template

$$\frac{\begin{array}{c} id \in \mathsf{Identifier}, \quad id : (\mathbf{r}\ (\mathtt{provides}, id_k, q_1, q_2)) \in \bar{\rho}, \\ l \in \mathsf{Port\text{-}list}, \quad \mathtt{uses}, \theta \vdash l \xrightarrow{\mathsf{Port\text{-}list}} \vec{p}, id_i, \\ id_i : (id_k\ \alpha) \in \psi, \quad \mathcal{A}^t_v(\alpha) = \dot{\alpha}_0, \quad a \in \mathsf{Attributes}, \quad \mathrm{T}, \alpha, \emptyset, \mathcal{C} \vdash a \xrightarrow{\mathsf{Attributes}} \dot{\alpha}_1, \\ \dot{\alpha} = \dot{\alpha}_0 \cup \dot{\alpha}_1, \quad \mathcal{Q}^t_a(\alpha, \dot{\alpha}), \quad id^t_r \in \mathsf{Identifier}, \quad id^t_r \notin \mathrm{dom}(\rho), \quad this.id^t_r \notin \mathrm{dom}(\theta) \end{array}}{\begin{array}{c} \mathrm{T}, \psi, \theta, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash id\ \mathtt{\{}\ a\ \mathtt{\}}\ \mathtt{=}\ l \xrightarrow{\mathsf{Binding}} \theta[this.id^t_r : (id_i\ \dot{\alpha})], \\ \rho[id^t_r : (id\ (\mathtt{provides}, id_i, q_2))], \Lambda[this.id^t_r : \vec{p}] \end{array}} \tag{4.207}$$

In addition to the previous rule, this rule interprets the given attribute list $a \in$ Attributes to complement the instance level attribute mapping $\dot{\alpha}$ of the newly created interface instance.

The port list $l \in$ Port-list serves two main purposes within the binding specification. First, it yields a list of port specifiers which denote the ports in the system which the new role is linked to. Second, by looking up the interface type of each port involved, it yields the interface type of the new role, which hence does not have to be given explicitly. The set Port-list is defined by the rule templates

$$\frac{p \in \mathsf{Port}}{p \in \mathsf{Port\text{-}list}} \tag{4.208}$$

$$\frac{p \in \mathsf{Port}, \quad l \in \mathsf{Port\text{-}list}}{p\mathtt{,}\,l \in \mathsf{Port\text{-}list}} \tag{4.209}$$

with the base case (the syntactic set Port) defined by the rule template

$$\frac{id_1, id_2 \in \mathsf{Identifier}}{id_1\,\mathtt{.}\,id_2 \in \mathsf{Port}} \tag{4.210}$$

Port-list is interpreted with a given parity c according to the rule templates

$$\frac{p \in \mathsf{Port}, \quad c, \theta \vdash p \xrightarrow{\mathsf{Port}} (id_c, id_p), id_i}{c, \theta \vdash p \xrightarrow{\mathsf{Port\text{-}list}} \{(id_c, id_p)\}, id_i} \tag{4.211}$$

for a singleton list, and

$$\frac{p \in \mathsf{Port}, \quad c, \theta \vdash p \xrightarrow{\mathsf{Port}} (id_c, id_p), id_i, \quad l \in \mathsf{Port\text{-}list}, \quad c, \theta \vdash l \xrightarrow{\mathsf{Port\text{-}list}} \vec{p}, id_i}{c, \theta \vdash p\mathtt{,}\,l \xrightarrow{\mathsf{Port\text{-}list}} \vec{p}[(id_c, id_p)], id_i} \tag{4.212}$$

for a list with head p and tail l. Note that both head and tail must evaluate to the same interface type id_i, which will be the interface type of the role which the ports link to. Elements of Port are interpreted according to the rule template

$$\frac{id_1, id_2 \in \mathsf{Identifier}, \quad id_1 : (id_c\ (\dot{\alpha}, \dot{\pi})) \in \theta, \quad id_2 : (id_p\ (c, id_i, q)) \in \dot{\pi}}{c, \theta \vdash id_1\,\mathtt{.}\,id_2 \xrightarrow{\mathsf{Port}} (id_1, id_2), id_i} \tag{4.213}$$

Note that the port's parity c has to match the one given. This ensures that provides-roles only connect to uses ports and uses-roles only to provides ports.

4.4.7 Completing the assembly

The whole assembly is described by elements of the set Scenario. It is defined by the rule template

$$\frac{id \in \text{Identifier}, \quad i \in \text{Identifier-list}, \quad b \in \text{Scenario-Body}}{\texttt{scenario}\ id\ \texttt{includes}\ i\ \texttt{\{}\ b\ \texttt{\}} \in \text{Scenario}} \tag{4.214}$$

where $i \in$ Identifier-list is (as opposed to previous identifier lists on the module or style tier) a *nonempty* list of identifiers. Identifier-list is defined by the rule templates

$$\frac{id \in \text{Identifier}}{id \in \text{Identifier-list}} \tag{4.215}$$

$$\frac{id \in \text{Identifier}, \quad i \in \text{Identifier-list}}{id\texttt{,}i \in \text{Identifier-list}} \tag{4.216}$$

for the case of a single identifier and for sequential composition of two lists. The body $b \in$ Scenario-Body is defined by the rule templates

$$\frac{}{\epsilon \in \text{Scenario-Body}} \tag{4.217}$$

$$\frac{t_1, t_2 \in \text{Scenario-Body}}{t_1\texttt{;}\ t_2 \in \text{Scenario-Body}} \tag{4.218}$$

for an empty body and sequential composition,

$$\frac{t \in \text{Component}}{t \in \text{Scenario-Body}} \tag{4.219}$$

$$\frac{t \in \text{Connector}}{t \in \text{Scenario-Body}} \tag{4.220}$$

for component and connector instances, and

$$\frac{t \in \text{Constant}}{t \in \text{Scenario-Body}} \tag{4.221}$$

for definitions of constants.

An element of Scenario is interpreted under the style mapping σ, the module mapping μ, and the scenario mapping ζ. Result of the interpretation is an updated scenario map ζ which contains the new assembly. The interpretation is defined by the rule template

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id \notin \text{dom}(\zeta), \quad i \in \text{Identifier-list}, \quad \mu \vdash i \xrightarrow{\text{Include}} \bar{\mathrm{T}}_0, \psi, \\ \text{Collect}(\mathcal{S}(\psi)) = (\hat{\mathrm{T}}, \gamma, \kappa, \Xi), \quad \mathcal{T}_k^m(\hat{\mathrm{T}}) = \bar{\mathrm{T}}_1, \quad \bar{\mathrm{T}} = \bar{\mathrm{T}}_0 \cup \bar{\mathrm{T}}_1, \quad \mathcal{T}_t^k(\hat{\mathrm{T}}) = \mathrm{T}, \\ b \in \text{Scenario-Body}, \quad \mathrm{T}, \kappa, \psi, \emptyset, \emptyset, \emptyset, \Xi \vdash b \xrightarrow{\text{Scenario-Body}} \psi_r, \theta, \Lambda, \mathcal{C}, \quad \mathcal{M}(\Lambda), \quad \mathcal{S}(\psi_r) \neq \emptyset \end{array}}{\sigma, \mu, \zeta \vdash \texttt{scenario}\ id\ \texttt{includes}\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\text{Scenario}} \zeta[id : (\mathbf{z}\ (\theta, \Lambda))]} \tag{4.222}$$

where the identifier list $i \in$ Identifier-list is the list of modules which the assembly draws its types from. It is interpreted according to the rule templates

$$\frac{id \in \text{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}, \psi)) \in \mu}{\mu \vdash id \xrightarrow{\text{Include}} \bar{\mathrm{T}}, \psi} \tag{4.223}$$

which interprets a single identifier $id \in$ Identifier as the name of a module and returns the module's attribute type set $\bar{\mathrm{T}}$ and type map ψ, and

$$\frac{id \in \text{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}_1, \psi_1)) \in \mu, \quad i \in \text{Identifier-list}, \quad \mu \vdash i \xrightarrow{\text{Include}} \bar{\mathrm{T}}_2, \psi_2}{\mu \vdash id\texttt{,}i \xrightarrow{\text{Include}} \bar{\mathrm{T}}_1 \sqcup \bar{\mathrm{T}}_2, \psi_1 \sqcup \psi_2} \tag{4.224}$$

which combines the attribute type sets and type maps from head and tail of the identifier list. The union symbol $\sqcup$ stands for a union with unique names, which unambiguously map to their respective structures. For example for the attribute type set $\bar{\mathrm{T}} = \bar{\mathrm{T}}_1 \sqcup \bar{\mathrm{T}}_2$ that means that

$$\forall id_1 : \bar{\tau}_1, id_2 : \bar{\tau}_2 \in \bar{\mathrm{T}}.\ \ id_1 = id_2 \Rightarrow \bar{\tau}_1 = \bar{\tau}_2$$

has to hold (*i. e.*, if the type name is the same, the type specification has to be the same). The result is a single kind level attribute type set $\bar{\mathrm{T}}_0$ and a single type map ψ.

The function

$$\mathcal{S} : \Psi \rightarrow 2^{\textsf{Identifier}}$$

is defined on top of the compliance relation $\dashv\models$ (see *sec.* 4.3.3) which holds between a style name id with $id : (\mathsf{s}\ (\hat{\mathrm{T}}, \gamma, \kappa, \Xi))$ and a type map ψ if all types in ψ live in kinds of κ. $\mathcal{S}$ is defined as

$$id \in \mathcal{S}(\psi) \quad \Leftrightarrow \quad id \dashv\models \psi$$

(*i. e.*, $\mathcal{S}(\psi)$ is the set of all styles (by name) which ψ complies to). On this set, the function

$$\mathrm{Collect} : 2^{\textsf{Identifier}} \rightarrow \hat{\mathcal{A}} \times \Gamma \times \mathrm{K} \times 2^{\Xi}$$

takes the set of style names and creates a collected style specification $(\hat{\mathrm{T}}, \gamma, \kappa, \Xi))$ out of the given styles by forming name-unambiguous unions (see above) of the sets in the given style's specifications. The constituents of this collected style's specification serve as the base for the interpretation of the assembly's body.

The predicate

$$\mathcal{M} \subseteq \mathcal{L},$$

applied to the list of link clauses resulting from the interpretation of the assembly's body, verifies whether all multiplexity constraints of the ports and roles within the assembly are observed.[12]

Note that the type map ψ_r is complemented with the on-the-fly created types for the connector instances. Since the kind mapping κ which they are drawn from is collected from possibly multiple styles, it is not automatically given that there is still a single style which entails all kinds used by the assembly. Therefore, a second application of $\mathcal{S}$ on the type mapping ψ_r resulting from the interpretation of the body verifies that there is at least one style which the assembly complies to.

Finally, elements of the set Scenario-Body are interpreted according to the rule templates

$$\frac{}{\mathrm{T}, \kappa, \psi, \theta, \Lambda, \mathcal{C}, \Xi \vdash \epsilon \xrightarrow{\textsf{Scenario-Body}} \psi, \theta, \Lambda, \mathcal{C}} \tag{4.225}$$

and

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Scenario-Body}, \quad \mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}_0, \Xi \vdash t_1 \xrightarrow{\textsf{Scenario-Body}} \psi_1, \theta_1, \Lambda_1, \mathcal{C}_1, \\ \mathrm{T}, \kappa, \psi_1, \theta_1, \Lambda_1, \mathcal{C}_1, \Xi \vdash t_2 \xrightarrow{\textsf{Scenario-Body}} \psi_2, \theta_2, \Lambda_2, \mathcal{C}_2 \end{array}}{\mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}_0, \Xi \vdash t_1 \texttt{;}\ t_2 \xrightarrow{\textsf{Scenario-Body}} \psi_2, \theta_2, \Lambda_2, \mathcal{C}_2} \tag{4.226}$$

for an empty body and for sequential composition of body elements,

$$\frac{t \in \textsf{Component}, \quad \mathrm{T}, \psi, \theta_0, \mathcal{C} \vdash t \xrightarrow{\textsf{Component}} \theta_1}{\mathrm{T}, \kappa, \psi, \theta_0, \Lambda, \mathcal{C}, \Xi \vdash t \xrightarrow{\textsf{Scenario-Body}} \psi, \theta_1, \Lambda, \mathcal{C}} \tag{4.227}$$

and

$$\frac{t \in \textsf{Connector}, \quad \mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\textsf{Connector}} \psi_1, \theta_1, \Lambda_1}{\mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\textsf{Scenario-Body}} \psi_1, \theta_1, \Lambda_1, \mathcal{C}} \tag{4.228}$$

for component and connector instantiations, and

$$\frac{t \in \textsf{Constant}, \quad \mathrm{T}, \mathcal{C}_0 \vdash t \xrightarrow{\textsf{Constant}} \mathcal{C}_1}{\mathrm{T}, \kappa, \psi, \theta, \Lambda, \mathcal{C}_0, \Xi \vdash t \xrightarrow{\textsf{Scenario-Body}} \psi, \theta, \Lambda, \mathcal{C}_1} \tag{4.229}$$

for the definition of constants.

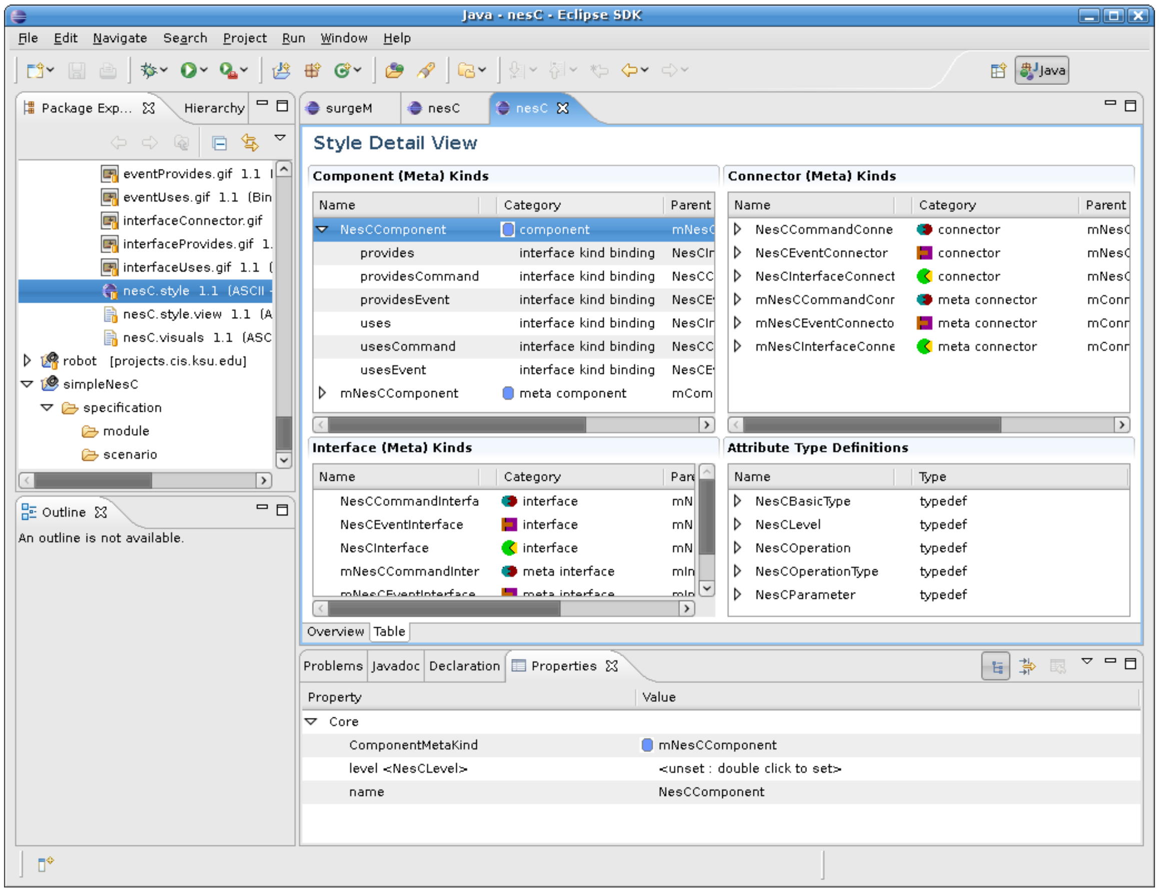
[12] While this is a non-local property and therefore in the formalism is easier to check globally, CADENA checks the property incrementally (see *sec.* 4.5)

4.5 Realization of CALM in Cadena

CALM was developed as a conceptual language, and various aspects of CALM make it hard to use it in its written form in practice. For example, the kinds defined on the style do not only carry their own structure, but also their own language (*e. g.*, the kind names are needed to create types, or port-option names are needed to create ports; these are essentially keywords of a style-specific language, so are platform specific attribute type names, connector names, *etc.*). While the creation of style-specific languages is desired and meant to simplify the use of CALM (*e. g.*, by offering the possibility to introduce already established terminology for specific middleware frameworks into CALM models as meaningful keywords), one consequence is that a user of CALM has to memorize new keywords for each style, which can be confusing even if the same user created the respective style himself, yet much more so if the style was created by another person.

Therefore, from the start of the CALM project, Cadena 2 [8, 7] was created in parallel to make the CALM concepts accessible. As CALM is the base concept for Cadena 2, the capabilities of Cadena 2 influenced the features of CALM.

4.5.1 Adaptive form- and graph-based editors


Java - nesC - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Package Exp... Hierarchy
surgeM nesC nesC
Outline
Problems Javadoc Declaration Properties
Overview Table

eventProvides.gif 1.1
eventUses.gif 1.1 (Bin
interfaceConnector.gif
interfaceProvides.gif 1.
interfaceUses.gif 1.1 (
nesC.style 1.1 (ASCII
nesC.style.view 1.1 (A
nesC.visuals 1.1 (ASC
robot [projects.cis.ksu.edu]
simpleNesC
specification
module
scenario

An outline is not available.

Style Detail View

Component (Meta) Kinds

Name	Category	Parent
NesCComponent	component	mNesC
provides	interface kind binding	NesCIr
providesCommand	interface kind binding	NesCC
providesEvent	interface kind binding	NesCE
uses	interface kind binding	NesCIr
usesCommand	interface kind binding	NesCC
usesEvent	interface kind binding	NesCE
mNesCComponent	meta component	mCom

Connector (Meta) Kinds

Name	Category	Parent
NesCCommandConne	connector	mNesC
NesCEventConnector	connector	mNesC
NesCInterfaceConnect	connector	mNesC
mNesCCommandConr	meta connector	mConr
mNesCEventConnecto	meta connector	mConr
mNesCInterfaceConne	meta connector	mConr

Interface (Meta) Kinds

Name	Category	Pare
NesCCommandInterfa	interface	mN
NesCEventInterface	interface	mN
NesCInterface	interface	mN
mNesCCommandInter	meta interface	mIn
mNesCEventInterface	meta interface	mIn

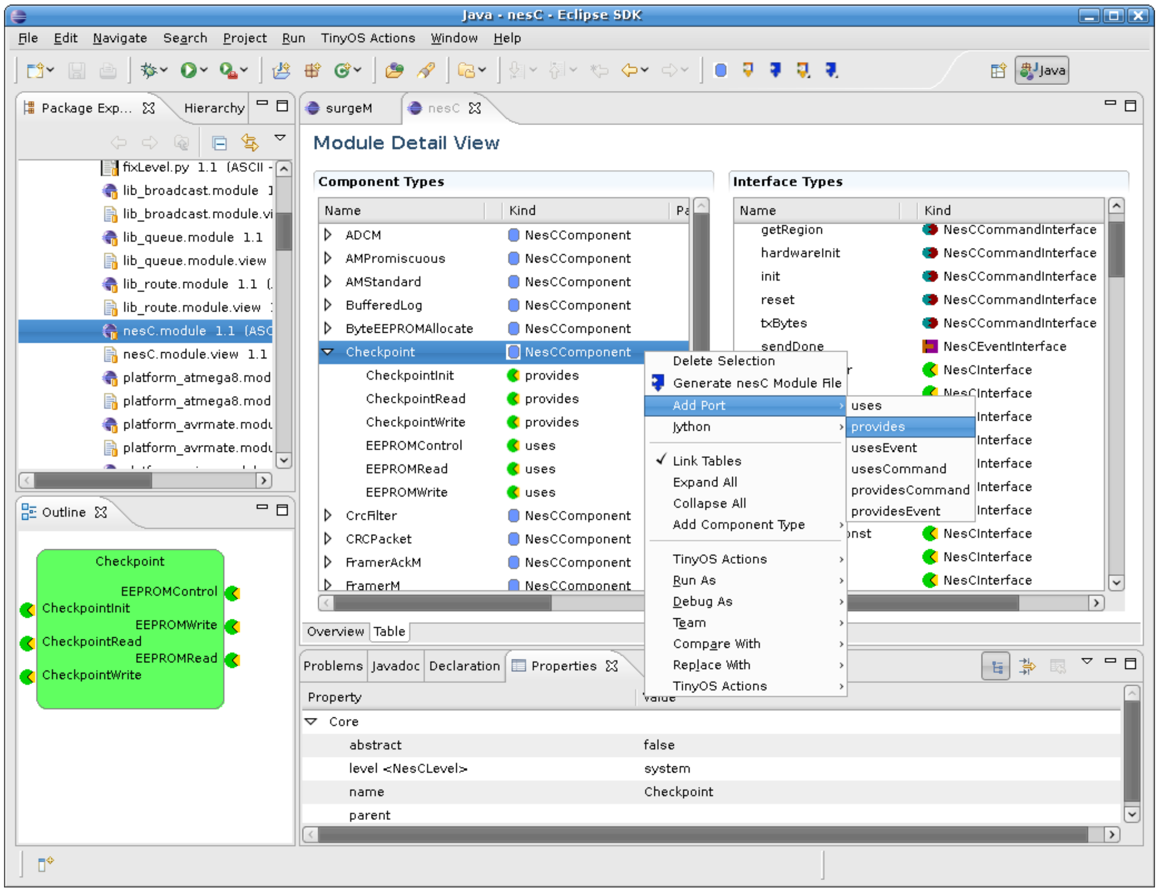
Attribute Type Definitions

Name	Type
NesCBasicType	typedef
NesCLevel	typedef
NesCOperation	typedef
NesCOperationType	typedef
NesCParameter	typedef

Property	Value
Core	
ComponentMetaKind	mNesCComponent
level <NesCLevel>	<unset : double click to set>
name	NesCComponent

Figure 4.3: Cadena style editor (defining nesC)

Figure 4.3 shows the Cadena style editor displaying a nesC style which slightly differs from the one in Listing 5.1 (*i. e.*, the main component kind is called **`NesCComponent`** instead of **`nesCModule`**, bare command and event interfaces are included in the model). The form based editing of architectural styles in Cadena is very close to the CALM textual form. In the Figure, the **`NesCComponent`** kind is opened, the port-options and their associations with interface kinds is visible. Note the user-defined port option names `provides`, `providesCommand`, `providesEvent`, `uses`, `usesCommand`, and `usesEvent`.



Component Types

Name	Kind
ADCM	NesCComponent
AMPromiscuous	NesCComponent
AMStandard	NesCComponent
BufferedLog	NesCComponent
ByteEEPROMAllocate	NesCComponent
Checkpoint	NesCComponent
CheckpointInit	provides
CheckpointRead	provides
CheckpointWrite	provides
EEPROMControl	uses
EEPROMRead	uses
EEPROMWrite	uses
CrcFilter	NesCComponent
CRCPacket	NesCComponent
FramerAckM	NesCComponent
FramerM	NesCComponent

Interface Types

Name	Kind
getRegion	NesCCommandInterface
hardwareInit	NesCCommandInterface
init	NesCCommandInterface
reset	NesCCommandInterface
txBytes	NesCCommandInterface
sendDone	NesCEventInterface
	NesCInterface
	NesCInterface
	Interface
	Interface
	Interface
	Interface
	Interface
nst	NesCInterface
	NesCInterface
	NesCInterface

Property	Value
Core	
abstract	false
level <NesCLevel>	system
name	Checkpoint
parent	

Figure 4.4: CADENA module editor (adding a port in nesC)

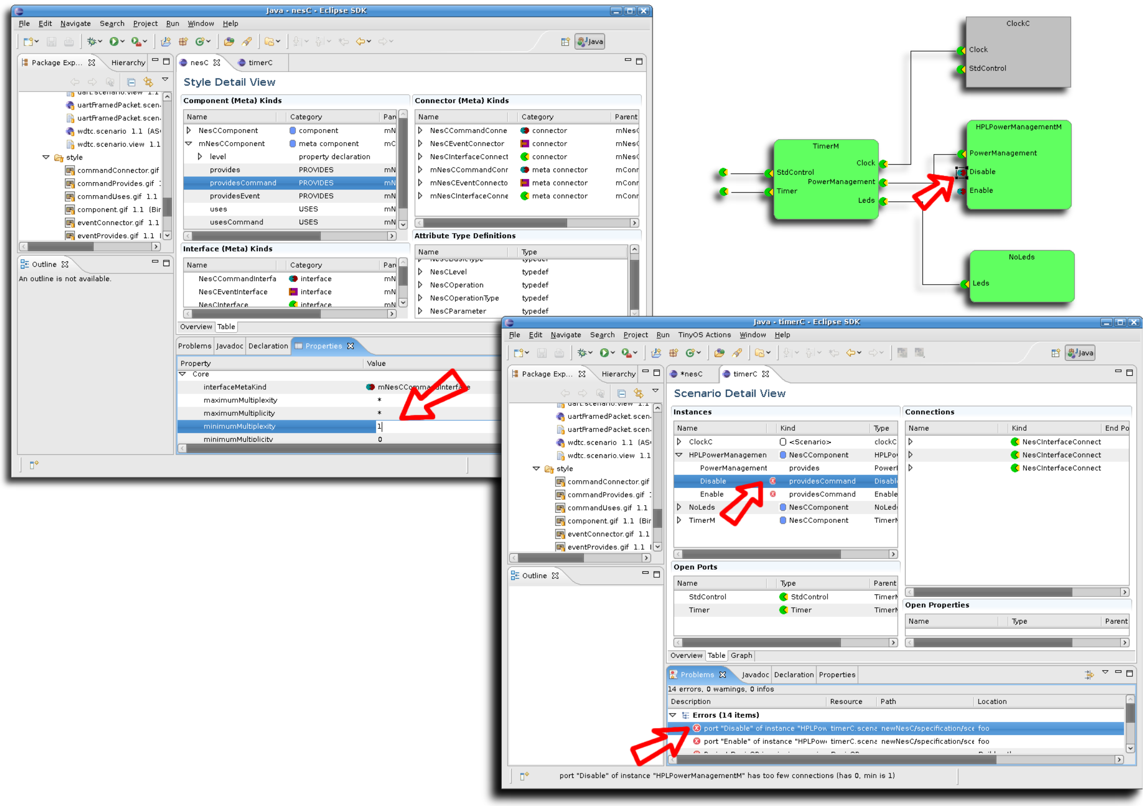
Figure 4.4 shows the module editor in CADENA, with a module within the nesC style defined by Figure 4.3. Analogous to the CALM textual form, the CADENA module tier declares types of component and interface kinds. To support the module developer, the editor automatically adapts to the architectural style. The highlighted ***NesCComponent*** 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.

4.5.2 EMF AUTOMATIC REFERENCING

Another problem of textual CALM in practical use is the massive interconnectedness between declared entities throughout the tiers. Especially, any change to an established style can orphan modules and scenarios which had been complying to that style before. Yet for a realistic development process it cannot be expected that a style is perfect before modules and assemblies are defined and never subject to change afterwards.

To address this issue, CADENA needs to be able to immediately propagate any changes to a model, regardless of whether the changes happen at the style, the module, or the scenario tier, to any place affected by the change, and report any problem arising with existing artefacts due to the change. To accomplish this, CADENA makes extensive use of the Eclipse Modeling Framework (EMF) [6, 19] which allows to connect objects through a reference and notification service. Through EMF, each object in CADENA gets notified if any object it directly depends on gets updated.[13] This update transitively propagates through all dependents.

[13] Therefore, EMF also guarantees the referencing property which is assumed for CALM's identifiers (see also 4.1.2).



Java - nesC - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Package Exp... Hierarchy
nesC timerC
Outline
An outline is not available.
Overview Table
Problems Javadoc Declaration Properties
ClockC
Clock
StdControl
TimerM
StdControl
Timer
Clock
PowerManagement
Leds
HPLPowerManagementM
PowerManagement
Disable
Enable
NoLeds
Leds
Java - timerC - Eclipse SDK
File Edit Navigate Search Project Run TinyOS Actions Window Help
*nesC timerC
Overview Table Graph

Style Detail View

Component (Meta) Kinds

Name	Category	Pan
NesCComponent	component	mN
mNesCComponent	meta component	mC
level	property declaration	
provides	PROVIDES	mN
providesCommand	PROVIDES	mN
providesEvent	PROVIDES	mN
uses	USES	mN
usesCommand	USES	mN

Connector (Meta) Kinds

Name	Category	Parent
NesCCommandConne	connector	mNesC
NesCEventConnector	connector	mNesC
NesCInterfaceConnect	connector	mNesC
mNesCCommandConr	meta connector	mConr
mNesCEventConnecto	meta connector	mConr
mNesCInterfaceConne	meta connector	mConr

Interface (Meta) Kinds

Name	Category	Pan
NesCCommandInterfa	interface	mN
NesCEventInterface	interface	mN
NesCInterface	interface	mN

Attribute Type Definitions

Name	Type
NesCLevel	typedef
NesCOperation	typedef
NesCOperationType	typedef
NesCParameter	typedef

Property	Value
Core	
interfaceMetaKind	mNesCCommandInterface
maximumMultiplexity	*
maximumMultiplicity	*
minimumMultiplexity	1
minimumMultiplicity	0

Scenario Detail View

Instances

Name	Kind	Type
ClockC	<Scenario>	clockC
HPLPowerManagemen	NesCComponent	HPLPo
PowerManagement	provides	Powerl
Disable	providesCommand	Disabl
Enable	providesCommand	Enable
NoLeds	NesCComponent	NoLeds
TimerM	NesCComponent	TimerM

Connections

Name	Kind	End Po
	NesCInterfaceConnect	
	NesCInterfaceConnect	
	NesCInterfaceConnect	

Open Ports

Name	Type	Parent
StdControl	StdControl	TimerM
Timer	Timer	TimerM

Open Properties

Name	Type	Parent

14 errors, 0 warnings, 0 infos

Description	Resource	Path	Location
Errors (14 items)			
port "Disable" of instance "HPLPow	timerC.scena	newNesC/specification/sce	foo
port "Enable" of instance "HPLPow	timerC.scena	newNesC/specification/sce	foo

port "Disable" of instance "HPLPowerManagementM" has too few connections (has 0, min is 1)

Figure 4.5: CADENA EMF-supported problem report

Figure 4.5 illustrates such a change-dependence situation (for better legibility the three artefacts of *fig.* 4.5 are displayed separately in *apx.* D.1, *fig.* D.1, D.2, and D.3). On the style tier, the multiplexity of the `providesCommand` port option of the `mNesCComponent` meta kind is changed from `[0..*]` to `[1..*]`, thereby requiring a minimum fan-out of 1 for each port declared within this port option (upper-left). Within the project there exist several scenarios which use instances of types within the kind `NesCComponent` which is created from `mNesCComponent`. One example is the `timerC` assembly (graph, upper right), which features the component `HPLPowerManagementM` of the `NesCComponent` type `HPLPowerManagement`. The component has two `providesCommand` ports, called `Disable` and `Enable`, which in this scenario are not connected. Therefore, the new fan-out requirement defined on the meta-kind which this instance ultimately depends on is not met. CADENA reports this problem immediately (marked with the error symbol "⊗"), as can be seen in the form-based scenario editor (lower left) at the component instance view, but also in the globally visible list of problems in the lower part of the CADENA window.

Next to propagating changes and checking consistency, the EMF automatic cross referencing framework used for engineering CADENA also simplifies the checking of the numerous completeness predicates introduced in the previous sections (*e. g.*, whether all port options of a component meta-kind are specified on a kind derived from that meta-kind, or if all multiplexity constraints are fulfilled in a scenario). It also serves for incremental type checking, where only those parts of a model are re-checked where an actual change happened.

4.5.3 CONCEPTUAL EXTENSIONS

Finally, CADENA is not only a frontend for CALM or the tool for practical testing of CALM, it also adds conceptual extensions to pure CALM. One way this is done is through the various plugin points offered by CADENA. Beyond modeling the interrelations of a component architecture (as pure CALM does), CADENA can be enhanced to do code generation for modeled architectures (implemented, *e. g.*, for nesC and CCM in end-to-end development projects), perform basic feasibility checks (data-flow and dependence analysis, implemented for PRiSM) or support developers by mechanically providing development heuristics (rate seeding in PRiSM).

Another capability of CADENA which has been discussed but is not fully implemented yet and which also reaches conceptually beyond CALM is automatic typing and kinding through delayed attribute valuation. The idea is to leave, for example, a type-level attribute unvaluated on the module tier of CADENA to create an incomplete or parametric type. This parametric type then can be extended into multiple different complete types depending on the valuation of that attribute given on instances created with the parametric type.

In summary, CADENA serves as an experimentation platform for CALM but also tries to investigate functionality which is not yet entirely formalized.

CHAPTER 5

CALM Models of Existing Component-frameworks

"The nice thing about standards is that you have so many to choose from."

— Andrew S. Tanenbaum. Computer Networks, 2nd ed, p.254

CALM and CADENA 2 have been used to capture various component frameworks, both industrial and experimental/educational. This chapter introduces three styles, `nesC`, `ccm`, and `Prism`, and gives examples of a module and an assembly within each style. All three of the selected styles have been developed and used in larger research/industry-cooperation projects: The `nesC` style is part of a current joined research effort at Kansas State University about sensor networks, where nesC and TinyOS-based devices are used in areas such as veterinary telemedicine or environmental/hydrological sensing. The `ccm` style was used in a cooperative effort with the Lockheed Martin Advanced Technology Laboratories towards robust support for model-driven development on the base of CADENA 2. Finally, the `Prism` style springs out of a cooperation with the Boeing Company on their Bold Stroke project in the context of the DARPA Program Composition for Embedded Systems (PCES), where CADENA has been used by Boeing to develop the avionics software flown on the Scan Eagle UAV platform for the PCES capstone demo (*i. e.*, the programs final life demonstration of achievements, see DARPA press release [12]).

Also, for all three styles CADENA plugins have been developed to support end-to-end development of systems within the given frameworks. For `nesC` and `ccm`, code generation facilities are integrated into CADENA 2, for `Prism` some analysis tools (slicing, dependency analysis) and development heuristics had already been implemented for CADENA 1.

The description of the `nesC` style and its artefacts is detailed and in-depth to supplement the CALM semantics described in Chapter 4 (*i. e.*, substantial parts of the CALM structures generated by the syntactical representation of `nesC` are presented for illustration). The `ccm` and `Prism` styles are presented in less detail with more focus on modeling decisions made to capture the respective frameworks.

5.1 The nesC ADL for highly distributed, embedded, systems

5.1.1 Architectural elements of nesC

The nesC component ADL [29, 28] features one kind of component, depending on its internals either called *module* (not to be confused with the module tier of CALM) or *configuration*. Components in nesC are black-boxes in as much as for the use of a component within a project its internals do not have to be known; nevertheless nesC terminology distinguishes between modules and configurations, the former being components which are implemented directly by code, the latter being components which internally consist of assemblies of smaller components, again either modules or configurations. This duality provides a flexible way of nesting, a whole system being one configuration, which on each level of nesting features similar structures. Modules and configurations exist for a number of small device-cores called *motes*. Often, the same module/configuration is implemented for multiple different motes. In these cases, the respective implementation for the modules/configurations is automatically selected at compile-time to fit to the target mote of the compilation. This bind-by-name strategy allows the nesC developer to assemble systems for multiple target motes at once. Unfortunately, the selection mechanism is based on complex conditionals within Unix *make*-files, and therefore error prone, a situation which at times leads to a failure to build a project occurring very late in the development process, but mostly works acceptably well in practice.

Each nesC module/configuration can have any number of ports drawn from three kinds of interfaces. The three interface kinds of nesC are called *command*, *event*, or *interface*. A nesC command is a single message-interface to the command provider (internally realized through a method call on the provider), a nesC event a message from the event-provider (usually a return value of a method of the provider). NesC interfaces are collections of commands and events, offering to structure commands and events into larger units. Since interfaces can contain a single command or a single event, it is possible without loss of expressiveness to omit the pure commands and events not contained inside interfaces to enforce more structure when modeling nesC.

Connectors in nesC are always binary. As there are three kinds of interfaces, there are also three kinds of connectors, which require strict type equality of the command/event/interface types of both of their roles. NesC terminology does not distinguish between command, event, or interface connectors since nesC itself does not entail the concept of kinds. In diagrams in nesC standard iconography, connectors are always double-arrow tipped lines, with solid tipped arrows for commands (pointing from user to provider), outlined tipped arrows for events (pointing from provider to user), and bundles of multiple solid and outlined arrows for interfaces corresponding to the events and commands an interface contains.

Being based on the C programming language and targeted for embedded systems, nesC features a subset of primitive C data-types, usually with reduced word lengths to accommodate small embedded processors. These types tend to have names which describe their domain, a typical nesC platform type being for example `uint8_t`, which is an unsigned 8-bit integer (and thus has the integer interval $\{0, \ldots, 255\}$ as its domain). Other nesC platform types are predefined enumeration types of domains which are commonly used in nesC projects such as the type `result_t`, which has the domain $\{\texttt{success}, \texttt{fail}\}$.

5.1.2 A nesC style

A possible CALM model of the (simplified) nesC architectural style is given in Listing 5.1. The CALM style `nesC` starts with the definition of a few of the nesC platform types in terms of CALM primitive data-types (*l.* 2–8). For example, Line 3

```
typedef uint8_t          = INT[0..255];
```

defines the CALM representation of the nesC type ***`uint8_t`*** (an 8-bit unsigned integer) as a CALM builtin **`INT`** which has been constrained to cover the limited interval from 0 to 255. Lines 11–19 define the CALM type `nesc_operation`, which is meant to capture the commands and events contained in the interface kind `nesCInterface`. Note that the type `nesc_operation` is not a platform type but a type used for modeling within CALM. The interpretation of all type definitions builds the attribute type set

$$\hat{\mathrm{T}} = \{\texttt{result\_t} : \hat{\tau}_0, \texttt{uint8\_t} : \hat{\tau}_1, \ldots\},$$

with $\hat{\tau}_0$ being the type defined by **`ENUM`**`{success, fail}`, $\hat{\tau}_1$ being the type defined by **`INT`**`[0..255]`, *etc.*. The interface meta-kind `mNesCInterface` is declared in Lines 21 and 22

```
   metainterface mNesCInterface {
22   attribute operations : MODULE nesc_operation list };
```

The interface meta-kind—meant to provide the structure for the nesC interface kind—features a single attribute `operations` which is a list with members of the type `nesc_operation`, which is set to be valuated on the type level through the binding time modifier **`MODULE`**. The whole meta-kind is recorded as

$$\texttt{mNesCInterface} : (\mathbb{i}\,\{\texttt{operations} : (\mathbb{a}\,\hat{\tau}_{\circ\text{-}\mathbf{list}})\}),$$

Listing 5.1: nesC-ADL style

```
   style nesC {
 2   typedef result_t       = ENUM { success, fail };
     typedef uint8_t        = INT[0..255];
 4   typedef uint16_t       = INT[0..65536];
     typedef uint32_t       = INT[0..4294967296];
 6   typedef int8_t         = INT[-128..127];
     typedef int16_t        = INT[-132768..32767];
 8   typedef int32_t        = INT[-2147483648..2147483647];
     typedef nesc_type      = union { result_t, uint8_t,
10     uint16_t, uint32_t, int8_t, int16_t, int32_t }
     typedef nesc_operation_type = ENUM { event, command };
12   typedef nesc_parameter = struct {
       name : STRING, type : nesc_type };
14   typedef nesc_operation = struct {
       async          : BOOLEAN,
16     name           : STRING,
       operation_type : nesc_operation_type,
18     parameters     : nesc_parameter list,
       return_type    : nesc_type };
20
     metainterface mNesCInterface {
22     attribute operations : MODULE nesc_operation list };
     interfacekind nesCInterface : mNesCInterface {};
24
     metacomponent mNesCModule {
26     provides [0..*] provides : mNesCInterface [0..*];
       uses     [0..*] uses     : mNesCInterface [0..*] };
28   componentkind nesCModule : mNesCModule {
       provides -> nesCInterface;
30     uses -> nesCInterface };

32   metaconnector mNesCWire {
       uses     [1] provider_side : mNesCInterface [1];
34     provides [1] user_side     : mNesCInterface [1] };
     connectorkind nesCWire : mNesCWire {
36     typevar a : nesCInterface;
       provider_side -> a;
38     user_side -> a }
   }
```

where $\hat{\tau}_{\texttt{o-list}}$ is the type defined for `operations` in Line 22. The meta-kind `mNesCInterface` is the only interface meta-kind defined within the simplified nesC style `nesC`, the "naked" commands or events have to be wrapped into interfaces in this style. This is a restriction compared to nesC itself, but it provides the same expressiveness while requiring more structure. Lines 25–27 define the component meta-kind `mNesCModule`.

```
     metacomponent mNesCModule {
26     provides [0..*] provides : mNesCInterface [0..*];
       uses     [0..*] uses     : mNesCInterface [0..*] };
```

Two port options (*l.* 26–27) specify the possibility to declare an arbitrary number (the multiplicity equals `[0..*]`) of provides- or uses-ports that feature interfaces from the structural realm defined by the interface meta-kind `mNesCInterface` and can be arbitrarily connected (the multiplexity equals `[0..*]`). The port options define the words "`provides`" and "`uses`" as keywords for provides- and uses-ports on the module tier respectively. These keywords coincide with the CALM internal keywords for port parities "**`provides`**" and "**`uses`**", but are not directly related. The style does not declare any attributes on the component meta-kind. The complete meta-kind is recorded with its empty attribute mapping and its port option mapping containing the two port options `provides` and `uses`:

$$\texttt{mNesCModule} : (\mathbb{c}\,(\emptyset, \{\ \ \texttt{provides} : (\mathbb{p}\,(\mathbf{\texttt{provides}}, \texttt{mNesCInterface}, \{0,\ldots,\star\}, \{0,\ldots,\star\})),\\ \texttt{uses} : (\mathbb{p}\,(\mathbf{\texttt{uses}}, \texttt{mNesCInterface}, \{0,\ldots,\star\}, \{0,\ldots,\star\}))\qquad \})).$$

Finally, Lines 32–34 define the sole connector meta-kind `mNesCWire`, which captures the structure of the nesC wire connection

```
     metaconnector mNesCWire {
33     uses     [1] provider_side : mNesCInterface [1];
       provides [1] user_side     : mNesCInterface [1] };
```

	$\texttt{nesC} : (\mathbf{s}\,(\hat{\mathrm{T}}, \gamma, \kappa, \Xi))$
CALM data types (platform types and types for modeling)	$\hat{\mathrm{T}} = \{\texttt{result\_t} : \hat{\tau}_0,\ \texttt{uint8\_t} : \hat{\tau}_1,\ \texttt{uint16\_t} : \hat{\tau}_2,\ \texttt{uint32\_t} : \hat{\tau}_3,\ \ldots\}$
Meta-kind set	$\gamma = \{\texttt{mNesCInterface} : (\mathbb{i}\,\hat{\alpha}_i),\ \texttt{mNesCModule} : (\mathbb{c}\,(\emptyset, \hat{\pi}_c)),$
	$\texttt{mNesCWire} : (\mathbb{l}\,(\emptyset, \hat{\rho}_l))\}$
Individual mappings	$\hat{\alpha}_i = \{\texttt{operations} : (\mathbb{a}\,\hat{\tau}_0)\}$
	$\hat{\pi}_c = \{\texttt{provides} : (\mathbb{p}\,(\textbf{provides}, \texttt{mNesCInterface}, \{0, \ldots, \star\}, \{0, \ldots, \star\})),$
	$\texttt{uses} : (\mathbb{p}\,(\textbf{uses}, \texttt{mNesCInterface}, \{0, \ldots, \star\}, \{0, \ldots, \star\}))$
	$\hat{\rho}_l = \{\texttt{provider\_side} : (\mathbb{r}\,(\textbf{uses}, \texttt{mNesCInterface}, \{1\}, \{1\})),$
	$\texttt{user\_side} : (\mathbb{r}\,(\textbf{provides}, \texttt{mNesCInterface}, \{1\}, \{1\}))$
Kind set	$\kappa = \{\texttt{nesCInterface} : (\mathbf{i}\,\bar{\alpha}_i),\ \texttt{nesCModule} : (\mathbf{c}\,(\emptyset, \bar{\pi}_c)),$
	$\texttt{nesCWire} : (\mathbf{l}\,(\emptyset, \bar{\rho}_l))\}$
Individual mappings	$\bar{\alpha}_i = \{\texttt{operations} : (\mathbf{a}\,\bar{\tau}_0)\}$
	$\bar{\pi}_c = \{\texttt{provides} : (\mathbf{p}\,(\textbf{provides}, \texttt{nesCInterface}, \{0, \ldots, \star\}, \{0, \ldots, \star\})),$
	$\texttt{uses} : (\mathbf{p}\,(\textbf{uses}, \texttt{nesCInterface}, \{0, \ldots, \star\}, \{0, \ldots, \star\}))$
	$\bar{\rho}_l = \{\texttt{provider\_side} : (\mathbf{r}\,(\textbf{uses}, \texttt{nesCInterface}, \{1\}, \{1\})),$
	$\texttt{user\_side} : (\mathbf{r}\,(\textbf{provides}, \texttt{nesCInterface}, \{1\}, \{1\}))$
Interface typing constraints	$\Xi = \{\mathrm{type}(\texttt{nesCWire.provider\_side}) = \mathrm{type}(\texttt{nesCWire.user\_side})\}$

Table 5.1: Formalization of nesC in CALM

The "wire" in nesC is a one-to-one connection between two ports. One is the provider of the interface, connected to the `provider_side` role, the other is the user, connected to the `user_side`. Both roles have multiplicity and multiplexity one (`[1]`). The whole meta-kind is captured as

$$\begin{aligned}\texttt{mNesCWire} : (\mathbb{l}\,(\emptyset, \{\ \ & \texttt{provider\_side} : (\mathbb{r}\,(\textbf{uses}, \texttt{mNesCInterface}, \{1\}, \{1\})),\\ & \texttt{user\_side} : (\mathbb{r}\,(\textbf{provides}, \texttt{mNesCInterface}, \{1\}, \{1\}))\quad \})).\end{aligned}$$

The kinds, essence of a style, are exported immediately after their respective meta-kinds. Line 23 defines the kind `nesCInterface` as having the structure of `mNesCInterface`

```
  interfacekind nesCInterface : mNesCInterface {};
```

The export is straightforward, the sole attribute on `mNesCInterface` (*i. e.*, `operations`) has no style-tier elements which would have to be valuated. The kind is recorded as

$$\texttt{nesCInterface} : (\mathbf{i}\,\{\texttt{operations} : (\mathbf{a}\,\bar{\tau}_{\texttt{o-list}})\}),$$

where (again) $\hat{\tau}_{\texttt{o-list}}$ is the type defined for `operations` in Line 22. Lines 28–30 define the component kind `nesCModule`

```
   componentkind nesCModule : mNesCModule {
29   provides -> nesCInterface;
     uses -> nesCInterface };
```

Both port options of the meta-kind, `provides` and `uses`, which on the meta-kind are defined with the interface meta-kind `mNesCInterface`, are specified to the kind `nesCInterface`. The resulting kind is

$$\begin{aligned}\texttt{nesCModule} : (\mathbf{c}\,(\emptyset, \{\ \ & \texttt{provides} : (\mathbf{p}\,(\textbf{provides}, \texttt{nesCInterface}, \{0, \ldots, \star\}, \{0, \ldots, \star\})),\\ & \texttt{uses} : (\mathbf{p}\,(\textbf{uses}, \texttt{nesCInterface}, \{0, \ldots, \star\}, \{0, \ldots, \star\}))\qquad \})).\end{aligned}$$

The export of the sole connector kind `nesCWire` in Lines 35–38

```
   connectorkind nesCWire : mNesCWire {
36   typevar a : nesCInterface;
     provider_side -> a;
38   user_side -> a }
```

differs from the export of `nesCModule` in the export of the roles `provider_side` and `user_side`. Instead of exporting the roles by giving the respective interface kind directly, Line 36 defines a type variable `a` within `nesCInterface`. The result of this is twofold. First, the connector kind is recorded with the interface kind in its role options

$$\texttt{nesCWire} : (\mathbf{l}\,(\emptyset, \{\ \ \texttt{provider\_side} : (\mathbf{r}\ (\texttt{uses}, \texttt{nesCInterface}, \{1\}, \{1\})), \\ \texttt{user\_side} : (\mathbf{r}\ (\texttt{provides}, \texttt{nesCInterface}, \{1\}, \{1\}))\ \ \})).$$

Second, CALM defines interface typing constraints

$$\xi_1 \;=\; \text{type}(\texttt{nesCWire.provider\_side}) = \texttt{a},\ \text{and}$$
$$\xi_2 \;=\; \text{type}(\texttt{nesCWire.user\_side}) = \texttt{a}.$$

The transitive hull of ξ_1 and ξ_2 yields ξ_3 with

$$\xi_3 \;=\; \text{type}(\texttt{nesCWire.provider\_side}) = \text{type}(\texttt{nesCWire.user\_side}),$$

resulting in an extended constraint set $\Xi = \{\xi_1, \xi_2, \xi_3\}$. Reduced by the constraints $\xi_i \in \Xi$ which contain local variables (*i. e.*, the variable `a`, which is local to the connector kind definition and is contained in ξ_1 and ξ_2), the set

$$\text{cls}_{\{\texttt{a}\}}(\Xi) = \{\xi_3\},$$

which is the new global set of type constraints. The complete formalization of the nesC ADL through the CALM `nesC` style is summarized in Table 5.1.

5.1.3 A NESC MODULE

Listing 5.9 defines a short module called `sensornet` within the ***nesC*** architectural style. Lines 2–37 declare constants, in this case all of them of the type ***nesc_operation***. For example, Lines 17–19 declare the constant `stop` by giving the respective structure as a literal.

```
   stop : nesc_operation = struct { async = false,
18   name = "stop", operation_type = command,
     parameters = [], return_type = result_t }
```

Recall that the type name ***nesc_operation*** denotes a kind level type $\bar{\tau}_0 \in \bar{\mathrm{T}}$. The literal therefore has to evaluate to the corresponding type level attribute type $\tau_0 \in \mathrm{T}$. Here this is trivially the case, since the original type ***nesc_operation*** : $\bar{\tau}_0$ does not contain any binding time specifiers.

Lines 40–46 declare ***nesCInterface*** types. For example, Line 43 declares a ***nesCInterface*** type called `Timer`.

```
nesCInterface Timer { operations = [start, stop, fire] };
```

nesCInterface is an interface kind defined by the `nesC` style (*lst.* 5.1) as

$$\textit{\textbf{nesCInterface}} : (\mathbf{i}\ \{\texttt{operations} : (\mathbf{a}\ \bar{\tau}_{\circ})\})$$

where $\bar{\tau}_{\circ}$ is defined by the style as

```
MODULE nesc_operation list
```

(*i. e.*, the attribute `operations` is a list with elements of the type ***nesc_operation*** which by the binding time specifier **MODULE** needs to be valuated at the type level). The body of `Timer` contains one attribute valuation, assigning a list of ***nesc_operation*** constants to the attribute `operations`. Altogether, Lines 40–46 define seven different ***nesCInterface*** types. Note that in the body of the type `RSend` in Line 42

```
nesCInterface RSend include Send { };
```

Listing 5.9: Module within the nesC-ADL

```
module sensornet of nesC {
  init : nesc_operation = struct { async = false,
    name = "init", operation_type = command,
    parameters = [], return_type = result_t };
  send : nesc_operation = struct { async = false,
    name = "send", operation_type = command,
    parameters = [struct { name = "payload",
      type = uint32_t }], return_type = result_t };
  sendDone : nesc_operation = struct { async = false,
    name = "sendDone", operation_type = event,
    parameters = [struct { name "result",
      type = result_t }], return_type = result_t };
  start : nesc_operation = struct { async = false,
    name = "start", operation_type = command,
    parameters = [struct { name = interval,
      type = uint32_t }], return_type = result_t };
  stop : nesc_operation = struct { async = false,
    name = "stop", operation_type = command,
    parameters = [], return_type = result_t }
  fire : nesc_operation = struct { async = false,
    name = "fire", operation_type = event, parameters =[],
    return_type = result_t };
  setRate : nesc_operation = struct { async = false,
    name = "setRate", operation_type = command,
    parameters = [struct { name = interval,
      type = uint32_t }], return_type = result_t };
  queue : nesc_operation = struct { async = false,
    name = "queue", operation_type = command,
    parameters = [struct { name = payload,
      type = uint32_t }], return_type = result_t };
  dequeue : nesc_operation = struct { async = false,
    name = "dequeue", operation_type = command,
    parameters = [], return_type = result_t };
  deliver : nesc_operation = struct { async = false,
    name = "deliver", operation_type = event,
    parameters = [struct { name = payload,
      type = uint32_t }], return_type = result_t };


  nesCInterface StdControl {operations = [init] };
  nesCInterface Send { operations = [send, sendDone] };
  nesCInterface RSend include Send { };
  nesCInterface Timer { operations = [start, stop, fire] };
  nesCInterface Clock { operations = [setRate, fire] };
  nesCInterface Queue { operations = [queue] };
  nesCInterface Dequeue { operations = [dequeue, deliver] };

  nesCModule LinkControl {
    provides init   : StdControl;
    provides input : Send;
    uses reset    : StdControl;
    uses clock    : Timer;
    uses queue    : Queue;
    uses dequeue : Dequeue;
    uses output   : Send }
  nesCModule Timer_ {
    provides init   : StdControl;
    provides timer : Timer }
  nesCModule Link {
    provides init   : StdControl;
    provides input : Send;
    uses output : RSend }
  nesCModule Clock_ {
    provides clock : Clock }
  nesCModule Queue {
    provides init     : StdControl;
    provides queue    : Queue;
    provides dequeue : Dequeue }
  nesCModule TimerControl include Timer {
    uses clock : Clock }
}
```

the attribute `operations` is not valuated. Instead, it inherits the valuation from the type `Send` through inclusion. The two ***nesCInterface*** types are structurally equal, but not interchangeable (see discussion in Section 4.3.4).

Finally, Lines 48–70 declare ***nesCModule*** types. Each ***nesCModule*** type declares some number of ports using the port option names ***provides*** and ***uses*** as keywords. For example, Lines 56–58 declare the ***nesCModule*** type `Timer_`.

```
nesCInterface RSend include Send { };
```

As with types of interface kinds, types of component kinds can include structure from already declared types. For example, the type `TimerControl` includes the two ***provides*** ports from `timer` and adds the ***uses*** port `clock`. Table 5.2 shows the complete structure of the ***nesCModule*** `Timer_`.

$\texttt{sensornet} : (\mathbf{m}\ (\mathrm{T}, \psi))$			
Types of the interface kind ***nesCInterface***	$\texttt{StdControl} :$	$(\textit{nesCInterface}\ \{\texttt{operations} : [\texttt{init}]\})$	$\in \psi$
	$\texttt{Timer} :$	$(\textit{nesCInterface}\ \{\texttt{operations} : [\texttt{start}, \texttt{stop}, \texttt{fire}]\})$	$\in \psi$
The port map of `Timer_`	$\{\texttt{timer} :$	$(\textit{provides}\ (\textbf{provides}, \texttt{Timer}, \texttt{[0..*]})),$	
	$\texttt{init} :$	$(\textit{provides}\ (\textbf{provides}, \texttt{StdControl}, \texttt{[0..*]}))\}$	$= \pi_0$
The type `Timer_` of the component kind ***nesCModule***	$\texttt{Timer\_} :$	$(\textit{nesCModule}\ (\emptyset, \pi_0))$	$\in \psi$

Table 5.2: Some elements of sensornet in nesC in CALM

5.1.4 A NESC ASSEMBLY

Listing 5.14: Simple assembly within the nesC-ADL

```
scenario NetworkLink includes sensornet {
  LinkControl linkControl { };
  Timer_      timer { };
  Link        hwRadioLink { };
  Queue       sendQueue { };

  nesCWire { user_side = linkControl.reset;
             provider_side = timer.init };
  nesCWire { user_side = linkControl.reset;
             provider_side = sendQueue.init };
  nesCWire { user_side = linkControl.reset;
             provider_side = hwRadioLink.init };
  nesCWire { user_side = linkControl.clock;
             provider_side = timer.timer };
  nesCWire { user_side = linkControl.queue;
             provider_side = sendQueue.queue };
  nesCWire { user_side = linkControl.dequeue;
             provider_side = sendQueue.dequeue };
  nesCWire { user_side = linkControl.output;
             provider_side = hwRadioLink.input };
}
```

Listing 5.14 shows a simple assembly which uses the ***sensornet*** module. For the purpose of presentation it will be assumed that the style ***nesC*** (*lst.* 5.1) is the only style which ***sensornet*** complies to, which means that for

$$\textit{sensornet} : (\mathbf{m}\ (\mathrm{T}, \Psi)) \in \mu$$

we have that

$$\mathcal{S}(\psi) = \{\textit{nesC}\}$$

which means that, by drawing types from the module ***sensornet***, the assembly is evaluated with the set of attribute types, kinds and interface typing constraints associated with the ***nesC*** style. This allows to create instances of the kind ***nesCWire***, which is a kind of ***nesC***.

Lines 2–5 instantiate various ***nesCModule*** types. For example, Line 2 creates the instance `linkControl` of the type ***LinkControl***.

```
LinkControl linkControl { };
```

Superficially, this line creates the instance

$$\texttt{linkControl} : (\textbf{\textit{LinkControl}}\ (\emptyset, \{\texttt{init}, \texttt{input}, \texttt{reset}, \texttt{clock}, \texttt{queue}, \texttt{dequeue}, \texttt{output}\}))$$

Note though, that two of the ports of the type ***LinkControl***, namely `init` and `input`, provide their interfaces. Therefore Line 2 implicitly also creates two interfaces according to the port types:

$$\texttt{linkControl.init} : (\textbf{\textit{StdControl}}\ \{\texttt{operations} : [\texttt{init}]\})$$

and

$$\texttt{linkControl.input} : (\textbf{\textit{Send}}\ \{\texttt{operations} : [\texttt{send}, \texttt{sendDone}]\})$$

Lines 7–20 create and instantiate ***nesCWire*** types. For example, Lines 11–12 define an implicitly named connector which links to the port `linkControl.reset` on its sole `user_side` role and to the port `hwRadioLink.init` on its sole `provider_side` role.

```
nesCWire { user_side = linkControl.reset;
           provider_side = hwRadioLink.init };
```

The effect of these lines is fourfold. First, a ***nesCWire*** type is created according to the kind definition and the types of the roles inferred from the interface instances they link to

$$\begin{aligned} \textit{type-name}_3 : (\textbf{\textit{nesCWire}}\ (\emptyset, \{\quad & \textit{role-name}_1 : (\texttt{user\_side}\ (\textbf{provides}, \textbf{\textit{StdControl}}, \texttt{[1]})) \\ & \textit{role-name}_2 : (\texttt{provider\_side}\ (\textbf{uses}, \textbf{\textit{StdControl}}, \texttt{[1]}))\quad \})). \end{aligned}$$

This type has to comply with multiplicity and interface typing constraints of the ***nesCWire*** kind. Namely it has to have exactly one `user_side` role and exactly one `provider_side` role which both feature the same ***nesCInterface*** type (in this case ***StdControl***). Second, an instance of the on-the-fly created ***nesCWire*** type is created

$$\textit{instance-name}_3 : (\textit{type-name}_3\ (\emptyset, \{\textit{role-name}_1, \textit{role-name}_2\}))$$

(note that CALM picks random names for almost every structural entity of connectors). Third, the provided interfaces are created. In this case, *role-name*$_1$ provides an interface of type ***StdControl***.

$$\textit{instance-name}_3\texttt{.}\textit{role-name}_1 : (\textbf{\textit{StdControl}}\ \{\texttt{operations} : [\texttt{init}]\})$$

Fourth and finally, the set of link clauses Λ receives two new clauses, one for each new role

$$\begin{aligned} \lambda_1 &= \textit{instance-name}_3\texttt{.}\textit{role-name}_1 : \{\texttt{linkControl.reset}\} \\ \lambda_2 &= \textit{instance-name}_3\texttt{.}\textit{role-name}_2 : \{\texttt{hwRadioLink.init}\} \end{aligned}$$

which reflect the connection defined by this instantiation. Table 5.3 illustrates the resulting assembly with its interrelations.

5.1.5 Semantic caveats

As described above, components in nesC fall into two categories, namely *modules* and *configurations*, which from the outside are indistinguishable (hence they are represented by only one component kind, ***nesCModule***, in the CALM ***nesC*** style). The difference between the two entities lies in their implementation, where modules are implemented by (nesC-specific) C code, while configurations are implemented as networks of other components, those in turn again being either modules or configurations. Therefore, the

NetworkLink : $(\mathbf{z}\,(\theta, \Lambda))$	
	The set of instances $\theta = \{$
linkControl :	(***LinkControl*** $(\emptyset, \{$init, input, reset, clock, queue, dequeue, output$\}))$,
timer :	(***Timer_*** $(\emptyset, \{$init, timer$\}))$,
hwRadioLink :	(***Link*** $(\emptyset, \{$init, input, output$\}))$,
sendQueue :	(***Queue*** $(\emptyset, \{$init, queue, dequeue$\}))\}$,
linkControl.init :	(***StdControl*** {operations : [init]}),
linkControl.input :	(***Send*** {operations : [send, sendDone]}),
timer.init :	(***StdControl*** {operations : [init]}),
timer.timer :	(***Timer*** {operations : [start, stop, fire]}),
hwRadioLink.init :	(***StdControl*** {operations : [init]}),
hwRadioLink.input :	(***Send*** {operations : [send, sendDone]}),
	...,
instance-name$_1$:	(*type-name*$_1$ $(\emptyset, \{$*role-name*$_1$, *role-name*$_2\}))$,
instance-name$_2$:	(*type-name*$_2$ $(\emptyset, \{$*role-name*$_1$, *role-name*$_2\}))$,
instance-name$_3$:	(*type-name*$_3$ $(\emptyset, \{$*role-name*$_1$, *role-name*$_2\}))$,
instance-name$_4$:	(*type-name*$_3$ $(\emptyset, \{$*role-name*$_1$, *role-name*$_2\}))$,
	...,
instance-name$_1$. *role-name*$_1$:	(***StdControl*** {operations : [init]}),
instance-name$_2$. *role-name*$_1$:	(***StdControl*** {operations : [init]}),
instance-name$_3$. *role-name*$_1$:	(***StdControl*** {operations : [init]}),
instance-name$_4$. *role-name*$_1$:	(***Clock*** {operations : [setRate, fire]}),
	..., }
	The set of link clauses $\Lambda = \{$
instance-name$_1$. *role-name*$_1$:	{linkControl.reset}
instance-name$_1$. *role-name*$_2$:	{timer.init}
instance-name$_2$. *role-name*$_1$:	{linkControl.reset}
instance-name$_2$. *role-name*$_2$:	{sendQueue.init}
	...,
instance-name$_7$. *role-name*$_1$:	{linkControl.output}
instance-name$_7$. *role-name*$_2$:	{hwRadioLink.input} }

Table 5.3: The NetworkLink assembly formalized in CALM

equivalent of a CALM assembly (*i. e.*, the arranging of instances of architectural elements into a cooperative topology) in nesC itself is the internals of a configuration.

While superficially a nesC configuration and a CALM assembly in ***nesC*** style look similar, there is no trivial correspondence between the two. Most importantly, nesC does not distinguish strictly between a module/configuration type and an instance thereof. Instead of creating possibly multiple instances of a component type, every component in nesC is introduced once into a system, multiple references always point to that same instance. For example, Listing 5.17 shows the top level configuration of the nesC Surge application.[1] On Lines 42 and 43 multiple components are introduced or allocated (*e. g.*, Main, SurgeM, TimerC, LedsC, ...), on Lines 45–64 connections between the ports of these components are defined (since nesC only knows one kind of connector which is restricted to one-to-one connections, explicit mentioning of the connectors is unnecessary). Among the components used is the configuration TimerC, which is defined in Listing 5.18.[2] Another component introduced in Surge application next to TimerC is GenericCommPromiscuous (*l.* 43) (which in the body of the configuration appears as Comm, *e. g.*, *l.* 51). The GenericCommPromiscuous configuration is shown in Listing 5.19.[3] Like Surge (*lst.* 5.17) it uses the component TimerC (*l.* 76, 85–86). Yet this is not a new instance, instead the connections defined in this configuration link to the same timer instance as the enclosing configuration Surge. Chapter 7 (particularly *sec.* 7.1.4, 7.1.5) discusses a CALM specific

[1] Listing 5.17 is an excerpt, for the full listing with copyright notice see Appendix C.1.
[2] Listing 5.18 is an excerpt, again, for the full listing with copyright notice see Appendix C.1.
[3] Listing 5.19 is an excerpt of GenericCommPromiscuous, for the full listing with copyright notice see Appendix C.1.

Listing 5.17: Excerpt of the nesC Surge application's main configuration

```
includes Surge;
includes SurgeCmd;
includes MultiHop;


configuration Surge {
}
implementation {
  components Main, SurgeM, TimerC, LedsC, NoLeds, Photo, RandomLFSR,
    GenericCommPromiscuous as Comm, Bcast, MultiHopRouter as multihopM, QueuedSend, Sounder;

  Main.StdControl -> SurgeM.StdControl;
  Main.StdControl -> Photo;
  Main.StdControl -> Bcast.StdControl;
  Main.StdControl -> multihopM.StdControl;
  Main.StdControl -> QueuedSend.StdControl;
  Main.StdControl -> TimerC;
  Main.StdControl -> Comm;
  //  multihopM.CommControl -> Comm;

  SurgeM.ADC   -> Photo;
  SurgeM.Timer -> TimerC.Timer[unique("Timer")];
  SurgeM.Leds  -> LedsC; // NoLeds;
  SurgeM.Sounder -> Sounder;

  SurgeM.Bcast -> Bcast.Receive[AM_SURGECMDMSG];
  Bcast.ReceiveMsg[AM_SURGECMDMSG] -> Comm.ReceiveMsg[AM_SURGECMDMSG];

  SurgeM.RouteControl -> multihopM;
  SurgeM.Send -> multihopM.Send[AM_SURGEMSG];
  multihopM.ReceiveMsg[AM_SURGEMSG] -> Comm.ReceiveMsg[AM_SURGEMSG];
  //multihopM.ReceiveMsg[AM_MULTIHOPMSG] -> Comm.ReceiveMsg[AM_MULTIHOPMSG];
}
```

Listing 5.18: Excerpt of TimerC configuration

```
configuration TimerC {
  provides interface Timer[uint8_t id];
  provides interface StdControl;
}

implementation {
  components TimerM, ClockC, NoLeds, HPLPowerManagementM;

  TimerM.Leds -> NoLeds;
  TimerM.Clock -> ClockC;
  TimerM.PowerManagement -> HPLPowerManagementM;

  StdControl = TimerM;
  Timer = TimerM;
}
```

approach to these semantics.

The identity of entities created through reference to the same name is a design choice in nesC which acknowledges the limited availability of resources in embedded systems. Nevertheless it does not mean that all facets of the concept of types are absent in nesC. Specifically for the timer configuration `TimerC`, but also for other configurations and modules, there are multiple implementations which reflect different platforms for which an executable can be compiled. Depending on an argument to the `make` program which drives the compilation of a nesC system, a platform specific implementation is chosen during the building process. `TimerC` can be seen as defining the type of each of its implementations.

5.2 The CORBA Component Model

5.2.1 Architectural elements of CCM

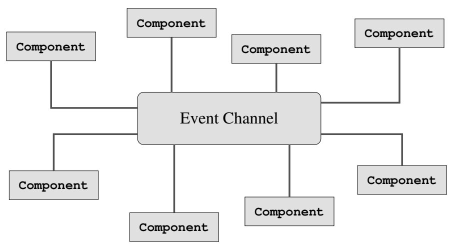
The CORBA Component Model (CCM) offers a single component kind. Two communication mechanisms are available, an asynchronous publish-subscribe event notification service and a synchronous remote method

Listing 5.19: Excerpt of GenericCommPromiscuous configuration

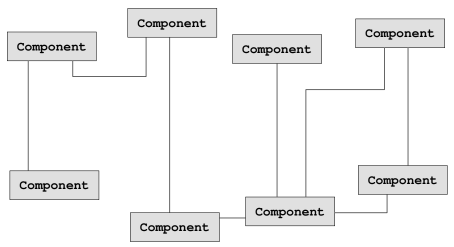
```
configuration GenericCommPromiscuous
{
  provides {
    interface StdControl as Control;
    interface CommControl;

    // The interface are as parameterised by the active message id
    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];

    // How many packets were received in the past second
    command uint16_t activity();

  }
  uses {
    // signaled after every send completion for components which wish to
    // retry failed sends
    event result_t sendDone();


  }
}
implementation
{
  // CRCPacket should be multiply instantiable. As it is, I have to use
  // RadioCRCPacket for the radio, and UARTNoCRCPacket for the UART to
  // avoid conflicting components of CRCPacket.
  components AMPromiscuous as AM,
    RadioCRCPacket as RadioPacket,
    UARTFramedPacket as UARTPacket,
    NoLeds as Leds,
    TimerC, HPLPowerManagementM;

  Control = AM.Control;
  CommControl = AM.CommControl;
  SendMsg = AM.SendMsg;
  ReceiveMsg = AM.ReceiveMsg;
  sendDone = AM.sendDone;

  activity = AM.activity;
  AM.TimerControl -> TimerC.StdControl;
  AM.ActivityTimer -> TimerC.Timer[unique("Timer")];

  AM.UARTControl -> UARTPacket.Control;
  AM.UARTSend -> UARTPacket.Send;
  AM.UARTReceive -> UARTPacket.Receive;

  AM.RadioControl -> RadioPacket.Control;
  AM.RadioSend -> RadioPacket.Send;
  AM.RadioReceive -> RadioPacket.Receive;
  AM.PowerManagement -> HPLPowerManagementM.PowerManagement;

  AM.Leds -> Leds;
}
```

call service. Ports which interface with the event notification service are called *event ports*, or depending on whether they send or receive notifications, event *sources* or event *sinks*. Ports which interface with the remote method call service are called *facet* if they provide methods (method execution site) or *receptacle* if they call methods (method call site). Interfaces of the remote method call service are expressed as bundles of method signatures and other attributes. In the CCM interface definition language version three (IDL3) they are declared with the keyword `interface`. Interfaces to the event notification service are method signatures of so called push-methods which carry the event data as their arguments. In the CCM IDL3 they are introduced as bundles of attributes that represent the arguments and hence the payload of the push method. As these bundles of arguments represent the data generated by an event about which the notification is sent, they are identified as the event structure and introduced in IDL with the keyword `eventtype`. Due to the choice of keywords it has become common to refer to the asynchronous notification service sloppily as *event connection* while the synchronous call service is referred to as *interface connection*.

The asynchronous event notification service is usually implemented through a (conceptually monolithic, centralized) functional unit called the *event channel* (EC). The EC in CCM is responsible for maintaining the connection between source and sink and for queueing of events (usually according to thread groups).



(a) CCM Event Channel (concrete)



(b) CCM Event Channel (abstract)

Figure 5.1: Abstracting a central communication unit in favor of intended connections

Nevertheless, instead of integrating the service unit EC into the architectural model as a concrete entity (*fig.* 5.1(a)), it is usually abstracted to show the intended connections instead (*fig.* 5.1(b)).

CCM is designed around object oriented languages for implementation. Therefore, a general object oriented inheritance subtyping is assumed for all entities of CCM. For example, for interfaces to the remote method call service the (explicitly declared) typing hierarchy is based on inclusion of the list of method signatures and fields analogous to classes or object signatures in an object oriented language. For reasons outlaid in Section 4.3.4, CALM does not implement this kind of structural subtyping in a generic way. To identify and respect subtyping relations in CADENA, the plug-in mechanism has to be used.

5.2.2 A CCM STYLE

Listing 5.20 shows a possible style for the CCM architecture framework. Line 3 includes CORBA data types from a separate file (shown in excerpts in Appendix C.2) which has a form similar to the first part of the ***nesC*** style (*lst.* 5.1, *l.* 2–19).[4] Based on the interface meta-kind `mCCMInterface` (*l.* 5–6) two specific interface meta-kinds are declared, `mCCMEventHandler` (*l.* 8) and `mCCMDataInterface` (*l.* 11–12). The kind `CCMEvent` (*l.* 9) is defined with the former, the kind `CCMInterface` with the latter (*l.* 13).

The sole component kind in CCM is captured with the meta-kind `mCCMComponent` (*l.* 15–21), which contains five different port options modeled after the respective keywords of the CCM IDL3, which are `provides` and `uses` for facets and receptacles, `consumes` for event sinks and `publishes` and `emits` for event sources. Ports introduced with the keyword `emits` differ from ports introduced with the keyword `publishes` through their multiplexity; while the former has a maximum fan-out of one, the fan-out is unbounded for the latter. The `mCCMComponent` meta-kind is used to define the `CCMComponent` kind (*l.* 22–28).

Two connector meta-kinds and kinds are declared/defined in the `ccm` style. The first one models the connection through the event service. Lines 30–32 declare the meta-kind `mCCMEventService` with two role options, `consumer` and `publisher`. When defining the kind `CCMEventConnector` (*l.* 33–37), two type variables (`a` and `b`) are used to express the property of the connector that the consumer-side event-service interface type (`a`) needs to be the same as or a subtype (in the object oriented sense) of that on the publisher side (`b`). This is due to the implementation of the event notification by a so called *push* method, which in this context means that the publisher calls a method on the consumer to notify about the occurrence of an event. Note that CALM can declare this property, yet the checking depends on tool support, specifically on a CADENA plugin.

The second connector models the data service connection. The meta-kind `mCCMDataService` is declared in Lines 39–41. It contains two role options (`facet` and `receptacle`). The meta-kind is used to define the kind `CCMInterfaceConnector` (*l.* 42–46). Again, the types for roles within the role options are constrained by two type variables (`a` and `b`) for which a subtype relation is declared. Here, the type of a facet interface has to be a subtype of a receptacle interface.

[4]Since most architectural models do not require any literals of platform types for their definition, it seems unnecessary to faithfully model the whole (rather extensive) CORBA type system. Listing C.4 is only a conceptual experiment.

Listing 5.20: CCM-ADL style

```
style ccm {

  include CorbaTypes;

  metainterface mCCMInterface {
    attribute fields  : MODULE field list };

  metainterface mCCMEventHandler extends mCCMInterface {};
  interfacekind CCMEvent : mCCMEventHandler {};

  metainterface mCCMDataInterface extends mCCMDataInterface {
    attribute methods : MODULE method_signature list };
  interfacekind CCMInterface : mCCMDataInterface {};

  metacomponent mCCMComponent {
    provides [0..*] provides  : mCCMDataInterface [0..*];
    uses     [0..*] uses      : mCCMDataInterface [0..*];

    provides [0..*] consumes  : mCCMEventHandler [0..*];
    uses     [0..*] publishes : mCCMEventHandler [0..*];
    uses     [0..*] emits     : mCCMEventHandler [0..1] };
  componentkind CCMComponent : mCCMComponent {
    provides -> CCMInterface;
    uses -> CCMInterface;

    consumes -> CCMEvent;
    publishes -> CCMEvent;
    emits -> CCMEvent };

  metaconnector mCCMEventService {
    uses     [1] consumer  : mCCMEventHandler [1];
    provides [1] publisher : mCCMEventHandler [1] };
  connectorkind CCMEventConnector : mCCMEventService {
    typevar a, b : CCMEvent;
    assert a <= b;
    consumer -> a;
    publisher -> b };

  metaconnector mCCMDataService {
    uses     [1] facet      : mCCMDataInterface;
    provides [1] receptacle : mCCMDataInterface };
  connectorkind CCMInterfaceConnector : mCCMDataService {
    typevar a, b : mCCMInterface;
    assert a <= b;
    facet -> a;
    receptacle -> b }
}
```

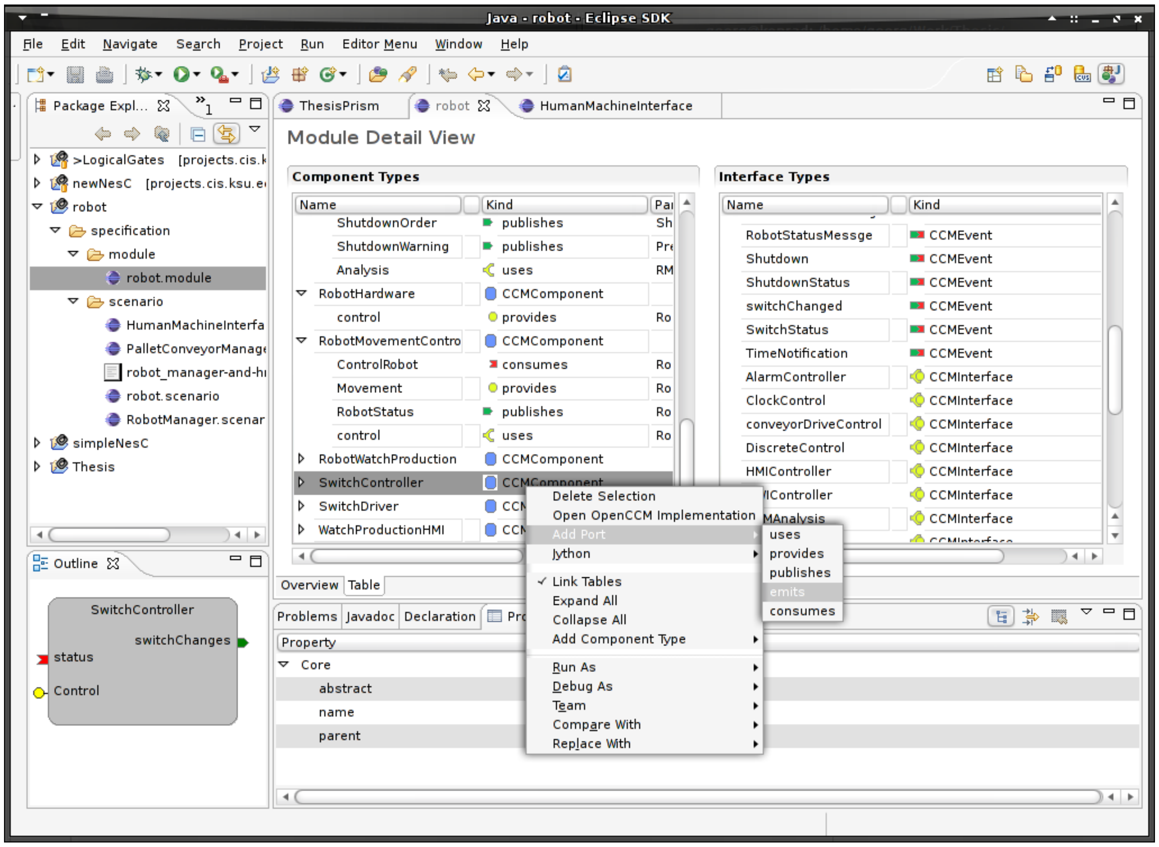
5.2.3 A CCM MODULE

Figure 5.2 shows a module of the ***ccm*** style within the CADENA module editor. This specific module is part of the "Robot Model", which served as a semi-realistic industrial example of CCM development used to communicate features, capabilities, requirements, and feedback about CADENA between Kansas State and Lockheed Martin.

The ***CCMComponent*** type `SwitchController` is highlighted in the Component Types area and thus a graphic representation is shown in the Outline view. To again illustrate the adaptiveness of CADENA's editors, the Add Port context menu is open in the screenshot, where the keywords introduced on the style tier (***uses***, ***provides***, ***publishes***, ***emits***, and ***consumes***) are visible as menu options.

A textual representation of the same module in CALM syntax is shown in Listing C.7 in Appendix C.2. The ***CCMComponent*** type `SwitchController`, which is highlighted in Figure 5.2, is defined declared in Lines 140–143 of this listing as

```
      CCMComponent SwitchController {
141     consumes  status : SwitchStatus;
        provides  Control : SwitchControl;
143     publishes switchChanges : switchChanged };
```



Module Detail View

Component Types

Name	Kind	Pa
ShutdownOrder	publishes	Sh
ShutdownWarning	publishes	Pr
Analysis	uses	RM
RobotHardware	CCMComponent	
control	provides	Ro
RobotMovementContro	CCMComponent	
ControlRobot	consumes	Ro
Movement	provides	Ro
RobotStatus	publishes	Ro
control	uses	Ro
RobotWatchProduction	CCMComponent	
SwitchController	CCMComponent	
SwitchDriver	CCM	
WatchProductionHMI	CCM	

Interface Types

Name	Kind
RobotStatusMessge	CCMEvent
Shutdown	CCMEvent
ShutdownStatus	CCMEvent
switchChanged	CCMEvent
SwitchStatus	CCMEvent
TimeNotification	CCMEvent
AlarmController	CCMInterface
ClockControl	CCMInterface
conveyorDriveControl	CCMInterface
DiscreteControl	CCMInterface
HMIController	CCMInterface
MIController	CCMInterface
MAnalysis	CCMInterface

Figure 5.2: A module within CCM in CADENA (Robot-model)

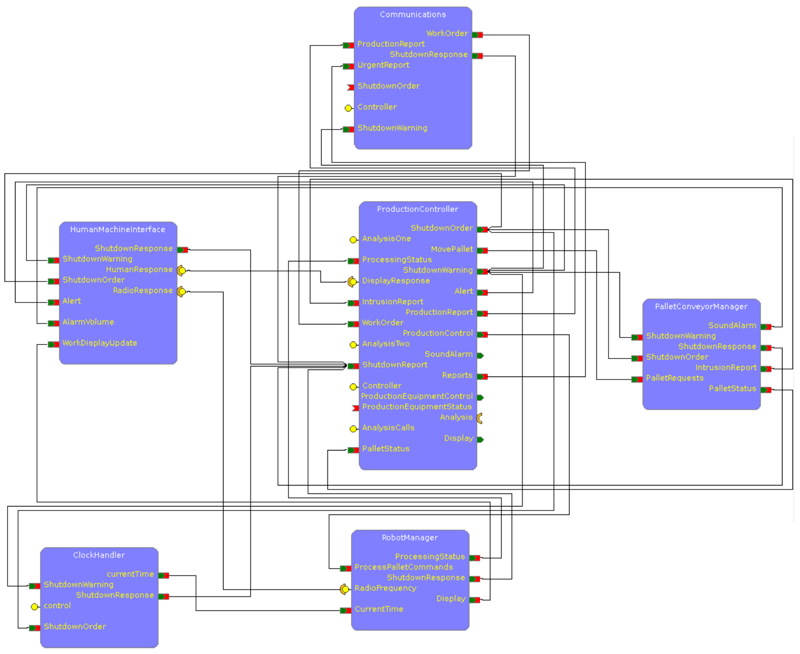


Figure 5.3: A CCM scenario (Robot Model) in the CADENA graphical scenario editor

5.2.4 A CCM scenario

Figure 5.3 shows the main assembly of the “Robot Model”. The top component `Communications` is an instance of the type ***`Communications`*** declared in the module ***`robot`*** (*lst.* C.7, *l.* 58–65).

```
   CCMComponent Communications {
59   consumes ProductionReport : ProductionStatus;
     consumes ShutdownOrder     : Shutdown;
61   consumes ShutdownWarning   : PrepareToShutdown;
     consumes UrgentReport      : IntrusionStatusReportManagement;
63   emits    ShutdownResponse : ShutdownStatus;
     emits    WorkOrder         : ProductionWorkOrder;
65   provides Controller : MWIController };
```

The provided interfaces are on the left, the used interfaces on the right side of the component (CADENA standard layout). Interfaces are displayed by pairs of matching icons, one of which represents the provider side (socket), the other the user side (plug). Table 5.4 outlines the ancestry for the `Communications`

$$\begin{array}{rrcc}
\gamma \vdash & \texttt{mCCMComponent} : (& \mathbb{c} & (\hat{\alpha}, \hat{\pi})) \\
 & \Downarrow & & \triangledown \\
\kappa \vdash & \textbf{\textit{CCMComponent}} : (& \mathbf{c} & (\bar{\alpha}, \bar{\pi})) \\
 & \searrow & & \triangledown \\
\psi \vdash & \textbf{\textit{Communications}} : (& \textbf{\textit{CCMComponent}} & (\alpha, \pi)) \\
 & \searrow & & \triangledown \\
\theta \vdash & \texttt{Communications} : (& \textbf{\textit{Communications}} & (\dot{\alpha}, \dot{\pi}))
\end{array}$$

Table 5.4: Ancestry of the Communications component

instance from the meta-kind level through the kind and type level to the instance level.

5.3 The Boeing Bold Stroke/PRiSMComponent Model

5.3.1 Architectural elements of Bold Stroke/PRiSM

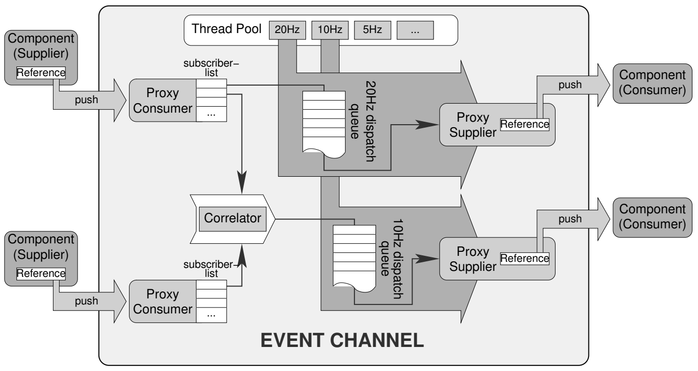


Figure 5.4: Schematics of the PRiSM event channel

The PRiSM component model was developed by the Boeing Company within the Bold Stroke project to implement aviation control software. It is conceptually based on CCM, but internally highly optimized towards real-time guarantees. PRiSM features one main component kind (PRiSM Components) as container for business logic (*i. e.*, device drivers, computation units, *etc.*) and an additional component kind for network-traffic optimization called *correlator*.[5] Similar to CCM, two communication services are offered,

[5]During our cooperation phase with Boeing correlators were in a conceptual stage. It is unknown to the author whether and in which form they are included in the actual current development at Boeing.

an asynchronous, purposefully simple, event notification service, and a synchronous data transfer service. As in the CORBA model, the asynchronous notification service is supported by a conceptually monolithic infrastructure unit called *event channel*. As opposed to CCM though, the PRiSM EC does not only receive, queue, and dispatch event notifications, it also maintains and controls the execution thread pool, dispatches events according to rate groups, generates timeout notifications, and houses the correlator components.

Figure 5.4 shows a schematic overview of the PRiSM EC. Event notifications (often shortly called *events*) are received by proxy consumers (*i. e.*, interfaces similar to the event sink interfaces on components), and queued into dispatch queues according to the connections stored in the proxy's subscriber list, optionally after being passed through correlators. The Thread Pool controls the dispatch of event notifications through proxy suppliers (*i. e.*, interfaces similar to the event source interfaces on components) to the receiving component. This mechanism ensures that each component can be assigned a specific run rate regardless of the run rates of other components it communicates with, since the buffering of the event notifications in the dispatch queues serves to bridge differing run rates without synchronization problems. Consequently, components are organized in thread groups according to their run rate, which in turn is determined by the main data/notification channels and the required real-time guarantees. To aide determining the suitable run-rate of any particular component, event notification messages carry the rate at which they occur as an attribute. In a complex setting, individual run-rates of components are then initially assigned (or *seeded*) by analyzing the propagation of event notifications starting from the initial timeout events through data accumulation and data processing components to display or actor components.

Because of the availability of an asynchronous notification service which can bridge thread groups, a *control-push—data-pull* strategy has been used frequently in PRiSM systems. When new data is generated by a (usually timeout-driven) device, the push-method based asynchronous event notification service is used to notify other components about the new data (control-push). Upon receiving the notification, the subscribing components then actively retrieve the data through the synchronous data service (data-pull). This strategy guarantees that processing components only become active when data is available and never block waiting for new input. It also means that many conceptual connections are formed by pairs of concrete connectors, one asynchronous notification connection and one synchronous data connection. Nevertheless, PRiSM lacks the necessary instruments of abstraction to merge these pairs into single connectors. CALM and CADENA's approach to add such an abstraction to a model is discussed in Section 6.3 and 7.2.2.

5.3.2 A PRiSM STYLE

Listing 5.23 shows a CALM style modeling the PRiSM architecture framework. It strives to provide an abstract view to the PRiSM infrastructure. As seen in the CCM style in Section 5.2, the event channel is not modeled as a single-block entity but instead as individual one-to-one connectors. For structural reasons, the style distinguishes between timeout event notifications and general event notifications which are indistinguishable in PRiSM's underlying implementation. Correlators and sources of timeout notifications are factored out of the event channel to allow the event notification communication itself to be abstracted into one-to-one connections easily.

In detail, the style defines three interface kinds, `PrismEvent` for interfaces to the event notification service, `PrismInterface` for interfaces to the data transfer service, and `PrismTimeout` for interfaces communicating timeout events (*l.* 4–9).

The main component kind, `PrismComponent` (*l.* 11–16, 17–22), contains port options, both used and provided, for all three interface kinds. Besides `PrismComponent` the `Prism` style defines two additional component kinds which model functions of the event channel that cannot easily be captured as infrastructure abstractions.

The `PrismCorrelator` kind (*l.* 23–25, 26–28) models correlators, a specialized kind of components which resides inside the event channel and filters event notifications according to defined patterns.[6] Through correlators the network traffic is optimized and, more importantly, components which subscribe to correlated event notifications instead of every single event notification which participates in a correlation pattern can be leaner in their implementation and are activated less often in a concrete environment. The `PrismCorrelator` is defined with two port options, one of them, `consumer` (*l.* 24), allowing an arbi-

[6]It has been proposed to use correlators or a similar semi-business–semi-infrastructure entity to perform standard communication tasks which go beyond pattern recognition but are still feasible for automatization, such as source selection or data-consistency control.

Listing 5.23: PRiSM-ADL style

```
    style Prism {
 2    typedef PrismRunRate = ENUM { 1, 5, 10, 20, 40 };

 4    metainterface mPrismEvent {};
      interfacekind PrismEvent {};
 6    metainterface mPrismInterface {};
      interfacekind PrismInterface {};
 8    metainterface mPrismTimeout {};
      interfacekind PrismTimeout {};
10
      metacomponent mPrismComponent {
12      provides [0..*] provides  mPrismInterface [0..*];
        uses     [0..*] uses      mPrismInterface [0..*];
14      provides [0..*] consumes  mPrismEvent     [0..*];
        uses     [0..*] publishes mPrismEvent     [0..*];
16      provides [0..*] timeout   mPrismTimeout   [0..*] }
      componentkind PrismComponent {
18      provides -> PrismInterface;
        uses -> PrismInterface;
20      consumes -> PrismEvent;
        publishes -> PrismEvent;
22      timeout -> PrismTimeout };
      metacomponent mPrismCorrelator {
24      provides [0..*] consumer  mPrismEvent [0..*];
        uses     [1]    publisher mPrismEvent [0..*] };
26    componentkind PrismCorrelator {
        consumer -> PrismEvent;
28      publisher -> PrismEvent };
      metacomponent mPrismTimer {
30      uses [0..1] timeout mPrismTimeout [0..*] };
      componentkind PrismTimer {
32      timeout -> PrismTimeout };

34    metaconnector mPrismEventConnector {
        attribute run_rate : SCENARIO PrismRunRate;
36      provides [1] source : mPrismEvent [1];
        uses     [1] sink   : mPrismEvent [1] };
38    connectorkind PrismEventConnector {
        typevar a : PrismEvent;
40      source -> a;
        sink -> a };
42    metaconnector mPrismInterfaceConnector {
        provides [1] user_side : mPrismInterface [1];
44      uses     [1] provider_side : mPrismInterface [1] };
      connectorkind PrismInterfaceConnector {
46      typevar a : PrismInterface
        user_side -> a;
48      provider_side -> a };
      metaconnector mPrismTimeoutConnector {
50      provides [1] source : mPrismTimeout [1];
        uses     [1] sink   : mPrismTimeout [1] };
52    connectorkind PrismTimeoutConnector {
        typevar a : PrismTimeout;
54      source -> a;
        sink -> a }
56  }
```

trary number of event sink interfaces (*i. e.*, provided interfaces to the event notification service), models the correlator input, the other one, `publisher` (*l.* 25), requiring exactly one event source interface (*i. e.*, used interface to the event notification service), models the correlator output.

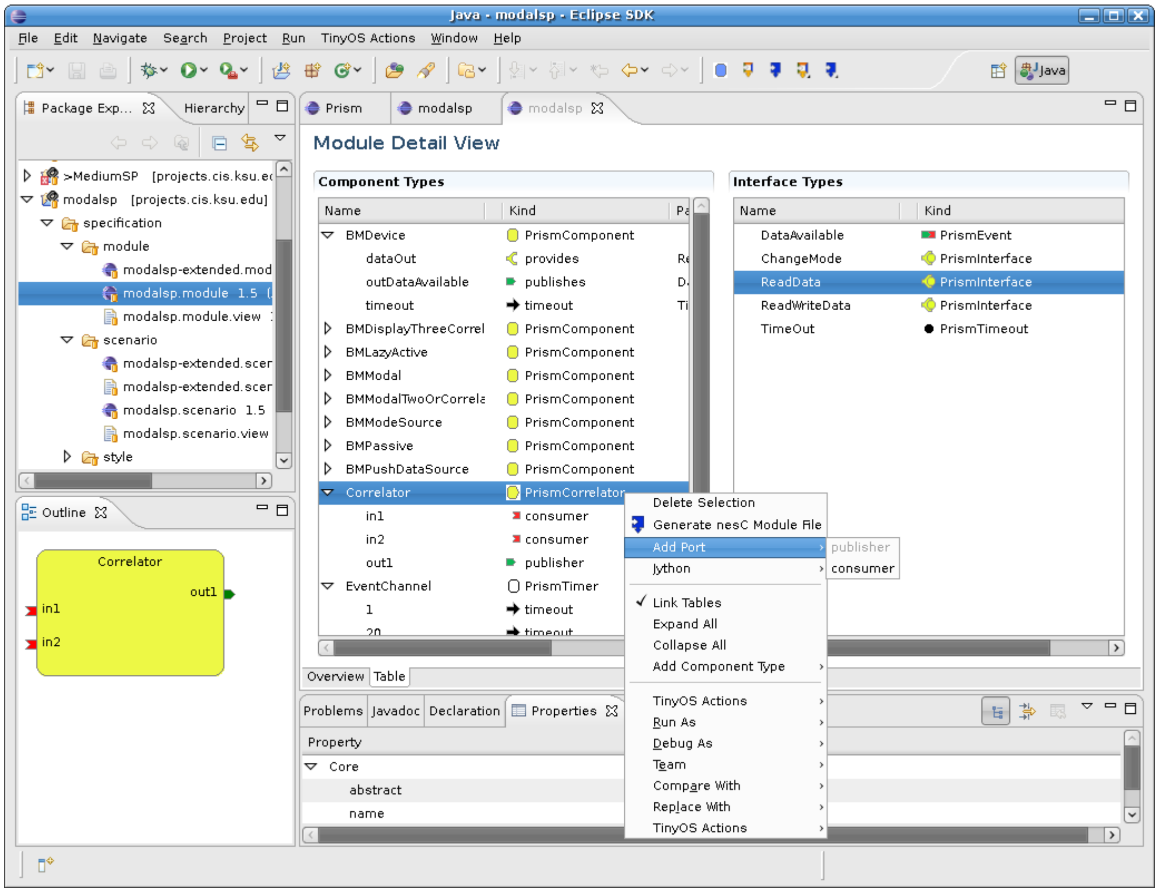
The `PrismTimer` (*l.* 29–39, 31–32) models the source of timeout events within the event channel's thread pool. Its port options allow at most one used interface to the timeout notification service. Canonically, this service should be modelled in CALM as a connector with one single role. Problems with the graph layout in CADENA at the time of the development of this style led to the modeling as a component kind.

Three connector kinds are defined in the `Prism` style. The `PrismEventConnector` (*l.* 34–37, 38–41) abstracts the communication mechanism of the event channel as one-to-one connections. As it is the case with similar connector kinds of other styles described above, `PrismEventConnector` requires the interfaces associated with its two roles to be of equal type (assured through the type variable `a`, *l.* 39). But instead of just modeling and infrastructure restriction of PRiSM, this typing requirement is a CALM abstraction. The type of an event notification in the actual PRiSM framework is a vague concept since each event notification has the same structure (*i. e.*, it is essentially a record of numerals which encode information about the event, such as its time of occurrence, its source, *etc.*). Instead of denoting an actual structural compatibility between different event notifications, the CALM type is used to add a nominal structure to the event notifications which is not present in PRiSM itself. In other words, while PRiSM only features generic event notifications, the CALM model of PRiSM adds the possibility to nominally distinguish the nature of the event about which a notification is given, and thereby helps to prevent false connections.

The `PrismInterfaceConnector` (*l.* 42–44, 45–48) connects data transfer service interfaces. As opposed to the event notifications in PRiSM, interfaces to the data transfer service do have an underlying type, that is manifested in the signature of the data retrieval method. Again, the interface type on both roles of the connector has to be equal.

Finally, `PrismTimeoutConnector` (*l.* 49–51, 52–55) serves to connect timeout notification ports. In addition to the distinction between timeout notifications and general event notifications, which is introduced on top of PRiSM itself by the CALM model, timeout event notifications can also be separated into nominal types. To support the use of these nominal types for structuring assemblies, the `PrismTimeoutConnector` also requires type equality for its two roles.

5.3.3 A PRiSM MODULE



Component Types

Name	Kind	Pa
BMDevice	PrismComponent	
dataOut	provides	Re
outDataAvailable	publishes	D.
timeout	timeout	Ti
BMDisplayThreeCorrel	PrismComponent	
BMLazyActive	PrismComponent	
BMModal	PrismComponent	
BMModalTwoOrCorrela	PrismComponent	
BMModeSource	PrismComponent	
BMPassive	PrismComponent	
BMPushDataSource	PrismComponent	
Correlator	PrismCorrelator	
in1	consumer	
in2	consumer	
out1	publisher	
EventChannel	PrismTimer	
1	timeout	
20	timeout	

Interface Types

Name	Kind
DataAvailable	PrismEvent
ChangeMode	PrismInterface
ReadData	PrismInterface
ReadWriteData	PrismInterface
TimeOut	PrismTimeout

Figure 5.5: CADENA module editor within PRiSM style

Figure 5.5 shows the `modalsp` module of the ***`Prism`*** style in CADENA. The PRiSM makes a very high-level use of component types. For example, the ***`PrismComponent`*** type `BMDevice` (opened in *fig.* 5.5) entails all possible sorts of sensor device drivers. They all have in common that they provide one timeout notification port which drives their data accumulation, they use an event notification port to publish a notification whenever new data has been produced, and they provide a data transfer interface for other components to retrieve the data. This single type is used for devices like GPS, accelerometer, altitude meter, *etc.*.

5.3.4 A PRiSM assembly

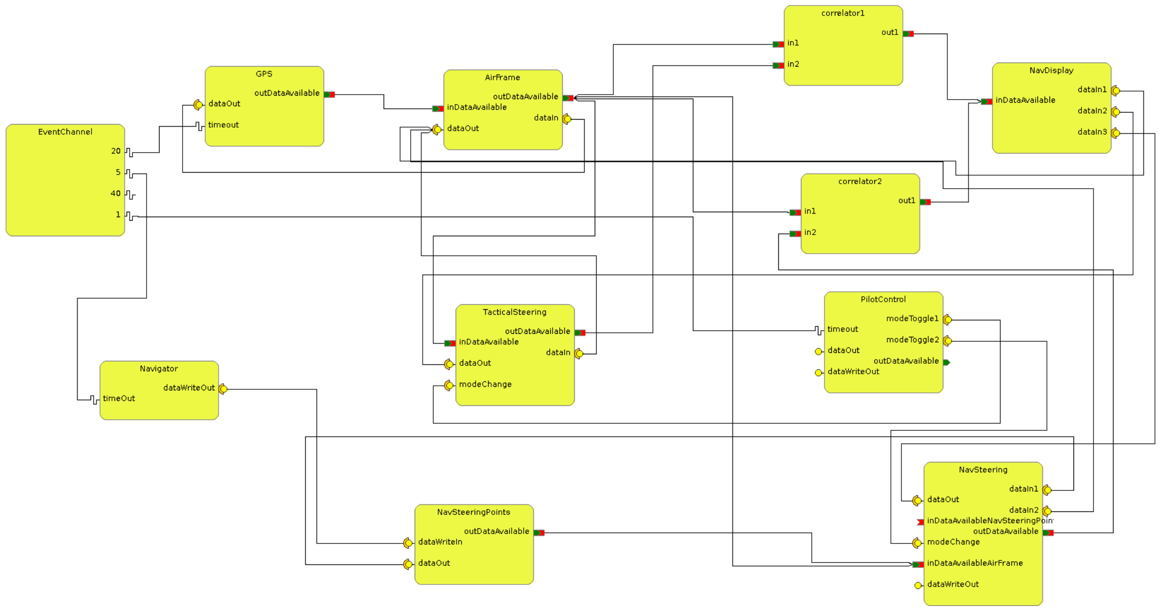


Figure 5.6: The ModalSP PRiSM assembly in CADENA

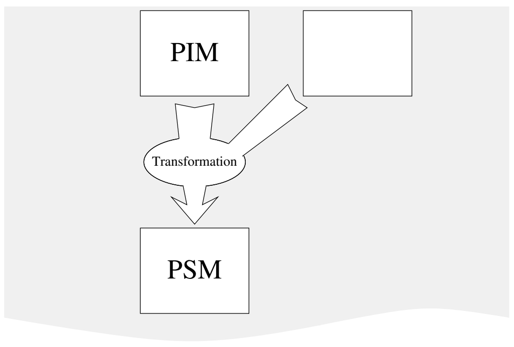
Figure 5.6 shows the **Prism** modalsp assembly in CADENA. The event channel is displayed in the scenario as a single timeout notification source with four timeout ports. An example for the control-push–data-pull strategy is the double connection between the **PrismComponent** instances GPS and Airframe. The GPS is a timeout driven device. Whenever it produces new data, it sends a notification on the outDataAvailable port (control-push). The Airframe receives this notification on its inDataAvailable port and retrieves the data through its dataIn port, which is connected back to the dataOut port on the GPS through the synchronous data transfer service (data-pull). Similar double connections exists for example between the Airframe and the TacticalSteering, between the Airframe and the NavSteering, or between the NavSteeringPoints and the and the NavSteering. These double connections sometimes cannot be identified without additional knowledge about the intended semantics of an assembly (see also *sec.* 6.3).

CHAPTER 6

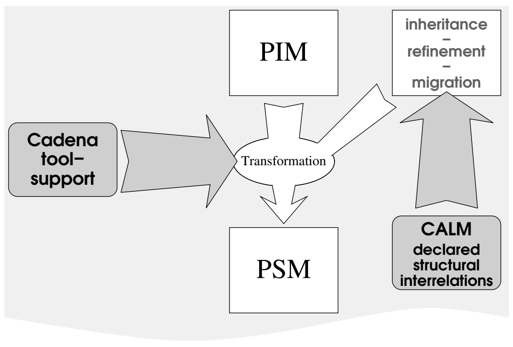
STYLE INTERRELATIONS AND STYLE REFINEMENT

6.1 IDEAS OF MODEL DRIVEN DEVELOPMENT

In 2001, the Object Management Group (OMG, www.omg.org) started the Model Driven Architecture (MDA) initiative to contribute to the ideas of model driven development. Core of the initiative is the attempt to make software in general more integrated (*i. e.*, less effort in composing software from different sources) and more easily maintainable (*i. e.*, less effort with debugging, decomposing, re-organizing, or repairing) by providing abstract models of the software architecture. These abstract models have to be capable of being mechanically translated into concrete, platform specific, systems. Therefore, a key factor of the abstract models is that they are provided in a standardized, machine manipulable/employable, form. In [53], the authors explain the process of leveraging an abstract model to obtain a concrete system as a transition from what they call the Platform Independent Model (PIM) to what they call the Platform Specific Model (PSM) (*fig.* 6.1(a)). The description of the proposed process is purposefully vague, the OMG suggests that both the transformation (oval) as well as any input to the transformation (empty rectangle) can be accomplished through various different mechanisms. Nevertheless, the OMG is working on a particular mechanism, namely on mappings of abstract modeling elements to services on all sorts of concrete platform specifications and implementations. The OMG explicitly states that multiple PIMs on succeeding levels of abstraction are intended in their process before a PSM is reached.



(a) OMG MDA Refinement (from [53])



(b) CALM/CADENA support for MDD transition

Figure 6.1: The CALM/CADENA support for a model driven architecture approach

CALM takes a similar approach of moving successively through models of different levels of abstraction, but it proposes a more specific methodology (*fig.* 6.1(b)). Abstractions and concretizations of architectural models are accomplished in CALM by moving the models through styles which are conceptually related through sharing kinds. Key factor in this approach is that CALM styles are a concrete, tangible (*i. e.*, manipulable), part of the modeling framework. The interconnectedness of all CALM artefacts allows to easily identify operations that can be accomplished mechanically *vs.* operations which intrinsically require interaction. Further, CALM/CADENA can offer extensive support for those operations which cannot be automatized.

6.2 REFINEMENT STRATEGIES IN CALM

6.2.1 MODEL MIGRATION

The process of transferring CALM architectural models (*i. e.*, CALM assemblies and their associated modules) from one style to another, related, style is called *model migration*. Roughly speaking, the relation between the source style and the receiving style is given through shared or even just related kinds. While the relation of sharing kinds can be seen from the point of various interpretations, it serves the intuition in most cases to think of one of the styles as a more abstract style and the other as the more concrete style to understand how this approach correlates with the OMG MDA proposal. Other (and not necessarily hierarchic) interpretations of a shared kind relation among styles include styles modeling different versions of the same platform, different implementations of the same infrastructure, the same platform in different computing environments, the same platform enhanced with various features, *etc.*. In summary, whereas the ideas of model migration are explained in terms of refinement, other uses are possible and intended in CALM.

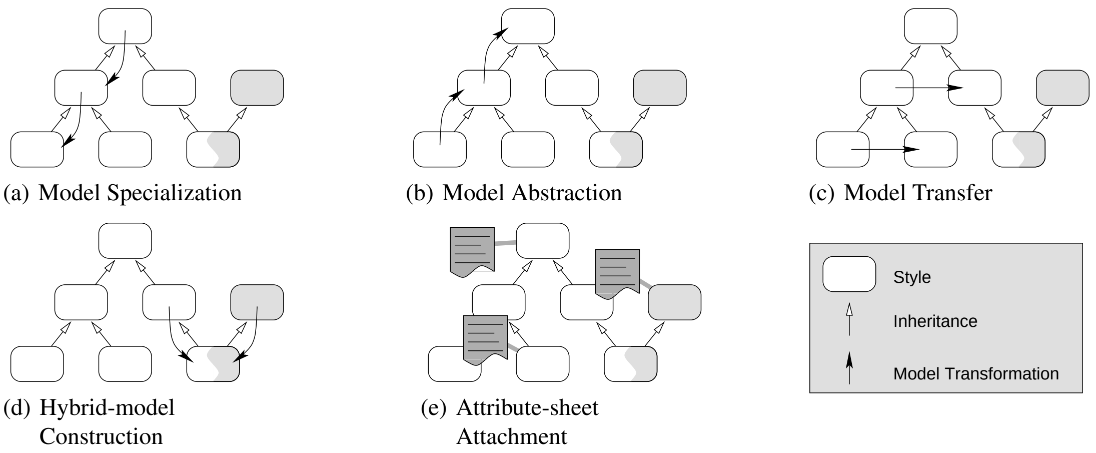
The concepts of style refinement and model migration first of all build on CALM's notion of *compliance*. A set of types of architectural entities is said to comply to a style if all kinds of the types are defined in that style. The formalization of CALM captures compliance through the relation

$$\mathrel{+\!\!\!\models} \subseteq \mathsf{Identifier} \times \Psi$$

(*sec.* 4.3.3, 4.4.7) which relates a style $id : (\hat{\mathrm{T}}, \gamma, \kappa, \Xi)$ given through its name id to a type map ψ of architectural elements iff all types in ψ live in kinds of κ, or formally speaking

$$id \mathrel{+\!\!\!\models} \psi \quad \Leftrightarrow \quad id : (\hat{\mathrm{T}}, \gamma, \kappa, \Xi) \in \sigma,\ \text{and} \begin{cases} \forall id_t^i : (id_k^i\ \alpha) \in \psi. \quad id_k^i : (\mathbf{i}\ \bar{\alpha}) \in \kappa,\ \text{and} \\ \forall id_t^c : (id_k^c\ (\alpha, \pi)) \in \psi. \quad id_k^c : (\mathbf{c}\ (\bar{\alpha}, \bar{\pi})) \in \kappa,\ \text{and} \\ \forall id_t^l : (id_k^l\ (\alpha, \rho)) \in \psi. \quad id_k^l : (\mathbf{l}\ (\bar{\alpha}, \bar{\rho})) \in \kappa. \end{cases}$$

Recall that not only from modules, but also from assemblies, maps of types can be extracted to verify compliance to a style. The discussion in this section will nevertheless focus on types and modules, since the methodology for assemblies and instances is subsumed in the handling of types.



(a) Model Specialization (b) Model Abstraction (c) Model Transfer
(d) Hybrid-model Construction (e) Attribute-sheet Attachment

Figure 6.2: Examples for using the style hierarchy as a tool for model driven design

Figure 6.2 illustrates the use of (hierarchical) style interrelations to accomplish various operations in MDD through model migration which will be explained in the following sections. A style A' in this illustration is said to *inherit* from a style A if it includes kinds from A through its header declaration (see *sec.* 4.2.16).

6.2.2 Specialization

Figure 6.2(a) illustrates the migration from more general to more specific styles in CALM. In this section, the migration of an architectural model from a style A to a A' which inherits from A will be referred to as *specialization*. Nevertheless, for reasons discussed in Section 6.3, CALM generally assumes a broader view and refers to all of the transfers illustrated in Figure 6.2 simply as *migration*.

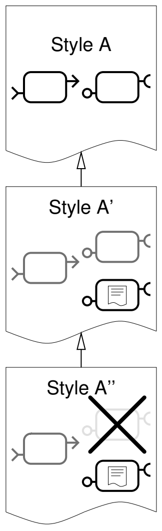


Figure 6.3: Refinement among styles related through shared or related kinds

Figure 6.3 exemplifies the relations between a style A and two successors A' and A'' which would be interpreted as successive levels of abstraction with A being more abstract than A''. Style A contains two component kinds for illustration (depicted as rounded rectangles with different port-icons standing for differing port options). Both kinds are inherited by style A' (turned grey). Therefore, any module or scenario created in A which makes use of these kinds of A (all other aspects neglected) also complies to A'. A migration from style A to style A' is therefore trivial, no change has to be made to the architectural models.

Nevertheless, style A' is not identical to style A. Instead it defines one original component kind which is a more specific version of one of the kinds of A (depicted through the "attribute sheet" in the new component kind). Therefore, architectural models which have been taken over from A can be gradually moved towards more specific versions by translating types of the inherited kind into types of the original kind. This process of transferring types from one kind to another is what CALM specifically refers to as migration, it is the core of translating whole modules and scenarios.

Migrating types can be trivial if the additional information is optional, or it can be done mechanically, if mappings, rules or heuristics for adding the new information exist, or it might intrinsically require interaction if the information has to be supplied from outside. The key for any mechanization or tool support is to exactly identify the information which needs to be added. Let $id : (\mathbf{c}\ (\bar{\alpha}, \bar{\pi}))$ be the inherited kind and $id' : (\mathbf{c}\ (\bar{\alpha}', \bar{\pi}'))$ the original kind. The migration of a type $id_t : (id\ (\alpha, \pi))$ into a type $id_t : (id'\ (\alpha', \pi'))$ then depends on the differences between $\bar{\alpha}$ and $\bar{\alpha}'$ and $\bar{\pi}$ and $\bar{\pi}'$.

$$\begin{array}{ccc} id : (\mathbf{c}\ (\bar{\alpha}, \bar{\pi})) & & id' : (\mathbf{c}\ (\bar{\alpha}', \bar{\pi}')) \\ \triangledown & & \triangledown \\ id_t : (id\ (\alpha, \pi)) & \rightsquigarrow & id_t : (id'\ (\alpha', \pi')) \end{array}$$

Any relationship between the meta-kinds from which inherited kind and original kind are produced might help in this step, nevertheless CALM does not require such a relationship to exist.

Listing 6.2: Migration example: A transitional style

```
style ccm_T include ccm {

  metacomponent mCCMComponent_M extends mCCMComponent {
    provides [1] monitor : mCCMDataInterface [0..1] }

  componentkind CCMComponent_M : mCCMComponent_M {
    provides -> CCMInterface;
    uses -> CCMInterface;

    consumes -> CCMEvent;
    publishes -> CCMEvent;
    emits -> CCMEvent;

    monitor -> CCMInterface };
}
```

Finally, style A'' inherits from A'. In A'', the original component kind, which stems from style A, is not present any more (depicted by its crossed-out shape); style A'' either did not include the kind while inheriting from style A' or explicitly purged the kind from its list of kinds by *elision* through the **elide** command. Elision allows to conveniently reduce inherited kinds from a style if, for example, the infrastructure does not provide for it anymore or, as in this case, a more specific version of the same kind has been defined.

The reason for removing the inherited kind from A'' is to use the compliance check of CALM to determine the completeness of migrations. Any architectural model from A' which complies to A'' can be considered completely migrated, since it does not contain types and instances of the old, abstract, inherited kind from style A anymore. Instead, all types which previously lived in the inherited kind have been transferred into the new kind of A'. In other words, style A' can be considered a *transitional style* in the gradual migration of architectural models from style A to style A''. Naturally, this same process can be repeated multiple times with a several transitional styles which form "check points" along the way of specializing architectural models.

As an example please consider the component kind ***CCMComponent*** of the ccm style, as defined in Listing 5.20 (*pp.* 75).

```
metacomponent mCCMComponent {
  provides [0..*] provides  : mCCMDataInterface [0..*];
  uses     [0..*] uses      : mCCMDataInterface [0..*];

  provides [0..*] consumes  : mCCMEventHandler [0..*];
  uses     [0..*] publishes : mCCMEventHandler [0..*];
  uses     [0..*] emits     : mCCMEventHandler [0..1] };
componentkind CCMComponent : mCCMComponent {
  provides -> CCMInterface;
  uses -> CCMInterface;

  consumes -> CCMEvent;
  publishes -> CCMEvent;
  emits -> CCMEvent };
```

Assume that there is a testing platform for CCM, which requires every component to have an additional port for monitoring its activity. Assemblies of the ccm style have to be migrated to a style called ccm_M which models the testing platform. A transitional style ccm_T can be defined as in Listing 6.2. The style ccm_T defines a new kind ***CCMComponent_M*** which differs from ***CCMComponent*** though one port option monitor, which requires exactly one port (multiplicity [1]) to a ***CCMDataInterface***. Therefore, for any ***CCMComponent*** type to fit into the new kind, a single monitor port has to be added.

Finally, ccm_M (*lst.* 6.3) represents the finished transition. It contains all the kinds of ccm_T minus the inherited kind ***CCMComponent***, which contains components without the monitor port.

Another example for the specialization of architectural models is a differentiation of the component kind ***nesCModule*** of the style nesC (*lst.* 5.1, *pp.* 65). Listing 6.4 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.

Listing 6.3: Migration example: Monitored CCM

```
style ccm_M include ccm_T {
  elide CCMComponent
}
```

Listing 6.4: More specific components in nesC

```
style nesC_refined extends nesC {
  elide nesCModule;

  componentkind nesCSoftModule : mNesCModule {
    provides -> nesCInterface;
    uses -> nesCInterface };
  componentkind nesCHWModule : mNesCModule {
    provides -> nesCInterface;
    uses -> nesCInterface };
  componentkind nesCNWModule : mNesCModule {
    provides -> nesCInterface;
    uses -> nesCInterface };
}
```

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; instead, the example shows how CALM allows to hand-tailor an existing component model such as nesC to a specific development context, in this case by introducing a distinction on structurally equal entities to enforce additional structure within architectural models. Note that no transitional style is needed in this case since the migration of types from ***nesCModule*** into any of the new kinds is trivial.

6.2.3 Abstraction and Transition

The abstraction (*fig.* 6.2(b)) of an architectural model is the reverse operation of the specialization as described in Section 6.2.2. Horizontal transfer of models as illustrated in Figure 6.2(c) can be seen as either a single operation or as successive steps of abstraction and specialization.

6.2.4 Hybrid-model construction

Hybrid styles (*fig.* 6.2(d)) model a situation of cooperating component infrastructures, which means they 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).

Figure 6.4 illustrates the underlying concept of hybrid styles. Two styles A and B model different component platforms. A hybrid style can be created by first forming a union of the two styles $A + B$, and then introducing “bridging elements” into the style. Bridging elements are components or connectors which can associate interfaces of both styles and therefore can close the gap between assemblies of the different styles.

Figure 6.5 shows a hybrid between CCM and PRiSM in Cadena. The CCM and PRiSM elements are inherited (not visible in this view, the style editor only displays original meta-kinds and kinds) and a component meta-kind (`mPrismCCMBridge`) and component kind (`PrismCCMBridge`) are added. `PrismCCMBridge` types can be used as “translators” between CCM and PRiSM elements.

A hybrid scenario is shown in Figure 6.6. In this figure, PRiSM components are yellow, CCM components blue, and hybrid original components white. Two data-acquisition components and a correlator from PRiSM communicate their results through one bridge component to the CCM data consumer.

6.2.5 Attribute sheets

A refinement/specialization situation which can be seen as orthogonal to the style inheritance hierarchy is illustrated in 6.2(e). The idea is to separate parts of the attribute declarations from the declared-name-based specification of structural interrelations. Instead of within the styles themselves, the separated attribute declarations are then captured in so called *attribute sheets* which can be attached to existing styles.

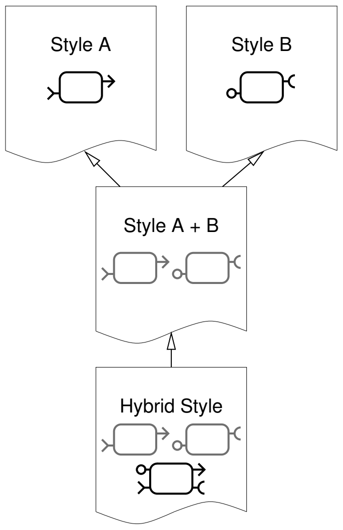
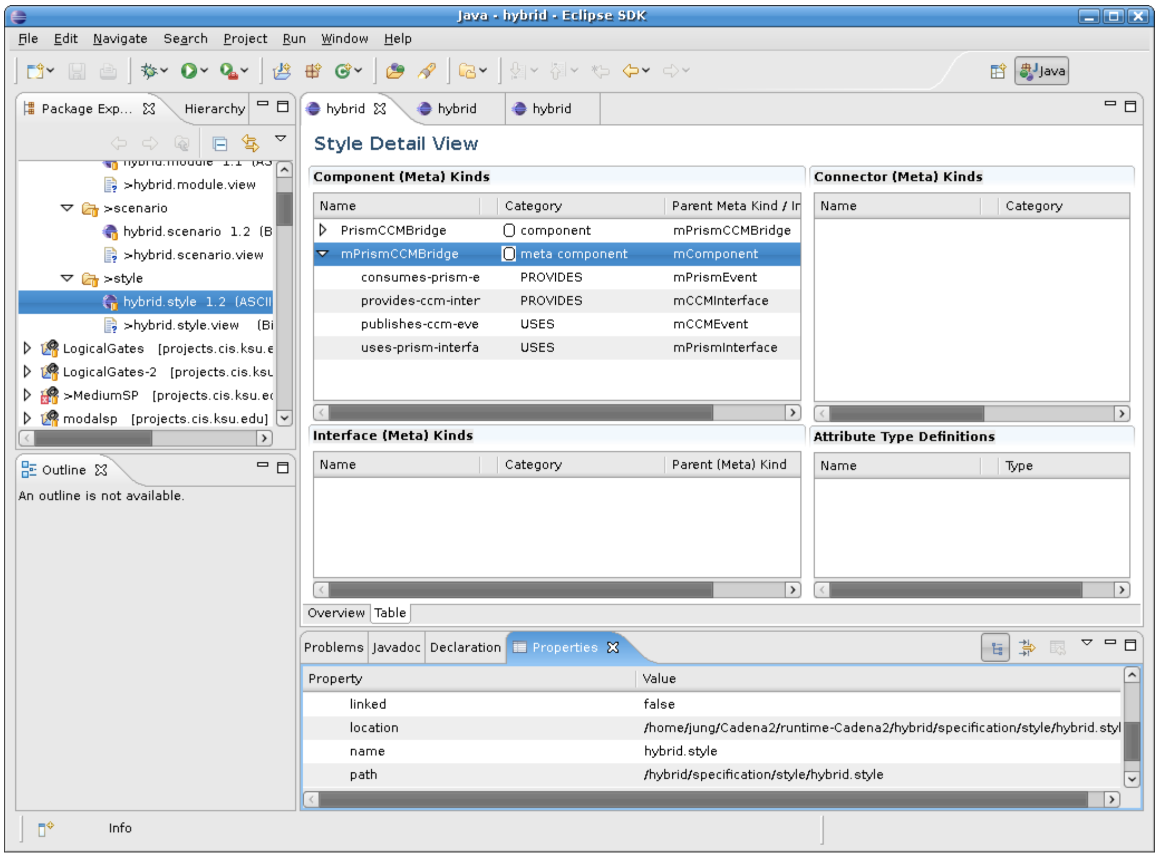


Figure 6.4: Construction of a hybrid style through inheritance


Java - hybrid - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Package Exp... Hierarchy
hybrid hybrid hybrid
Outline
Overview Table
Problems Javadoc Declaration Properties
Info

hybrid.module 1.1 (AS
>hybrid.module.view
>scenario
hybrid.scenario 1.2 (B
>hybrid.scenario.view
>style
hybrid.style 1.2 (ASCII
>hybrid.style.view (Bi
LogicalGates [projects.cis.ksu.e
LogicalGates-2 [projects.cis.ksu
>MediumSP [projects.cis.ksu.ec
modalsp [projects.cis.ksu.edu]

An outline is not available.

Style Detail View

Component (Meta) Kinds

Name	Category	Parent Meta Kind / Ir
PrismCCMBridge	component	mPrismCCMBridge
mPrismCCMBridge	meta component	mComponent
consumes-prism-e	PROVIDES	mPrismEvent
provides-ccm-inter	PROVIDES	mCCMInterface
publishes-ccm-eve	USES	mCCMEvent
uses-prism-interfa	USES	mPrismInterface

Connector (Meta) Kinds

Name	Category

Interface (Meta) Kinds

Name	Category	Parent (Meta) Kind

Attribute Type Definitions

Name	Type

Property	Value
linked	false
location	/home/jung/Cadena2/runtime-Cadena2/hybrid/specification/style/hybrid.styl
name	hybrid.style
path	/hybrid/specification/style/hybrid.style

Figure 6.5: Defining a hybrid style in CADENA

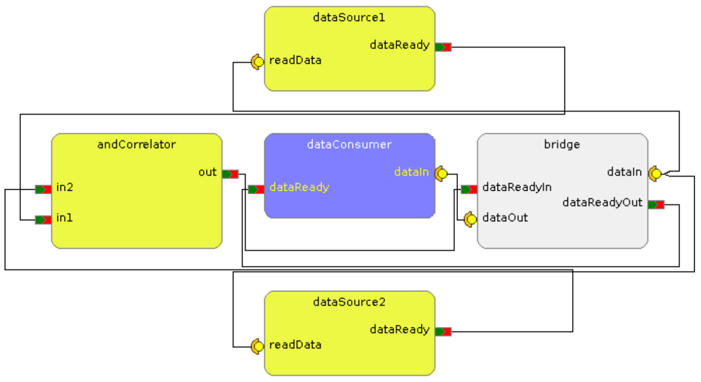


Figure 6.6: An assembly of the hybrid style

In this approach, the presence or absence of an attached attribute sheet distinguishes a more specific *vs.* a more abstract version of a style. The transfer of types from the abstract version of a style without attributes to the specific version of that style with attributes attached and *vice versa* is a migration operation corresponding to the transfer between different styles.

Separate attribute sheets are not discussed formally in Chapter 4, but they are available as a functionality of CADENA.

6.3 Discussion

In the previous sections, a style which only features a subset of the kinds of another is considered more general and abstract, while the style with more kinds is assumed to be more specific and concrete. Also, a style in which more attributes are declared is regarded to be more specific, a style with less attributes more general. Nevertheless, the generalized assumption that a style is more specific if it contains more information (more kinds, more attributes) falls short when applied to complex development contexts.

As an example consider the Boeing Bold Stroke/PRiSM development as described in Section 5.3. As described, PRiSM architectures frequently use a combination of one asynchronous notification connection and one synchronous data transfer connection to implement a single conceptual connection which spans different thread groups.

CALM allows to easily define a new kind of connector which does not model a single connection service provided by the infrastructure, but instead abstracts a bundle, which concretely is a pair out of two services and therefore manifests what in the Bold Stroke/PRiSM context is merely a suggested assembling strategy as a modeling element. Figure 6.7 for example shows a different version of the ModalSP assembly in a PRiSM style, called `prism-extended`, revised from the `prism` style (*sec.* 5.3) to contain the connector kind ***`PrismPushPullConnector`*** which models the paired connection. In the `prism-extended` style, the main component kind ***`PrismComponent`*** is elided in favor of a new, original, kind ***`PrismExtendedComponent`***, which features a port option to interface with the new, abstract, ***`PrismPushPullConnector`*** service.

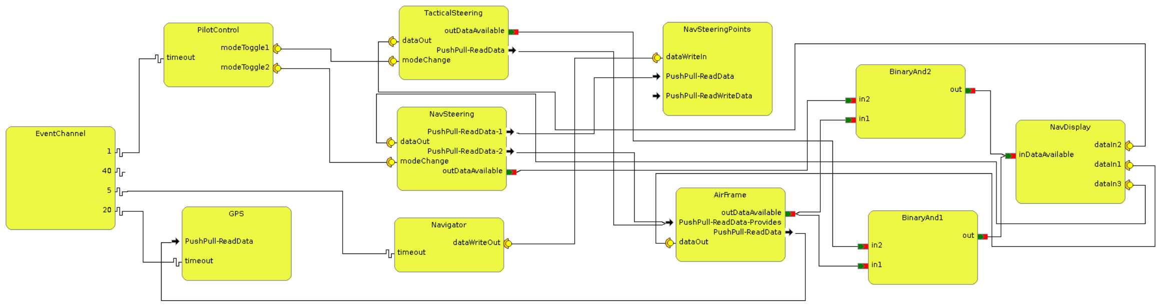


Figure 6.7: The ModalSP PRiSM assembly with abstract connections (compare to *fig.* 5.6, *pp.* 81)

The benefit of introducing this abstraction is not substantial in the reduction of the number of connections (the ModalSP example in its form in Figure 6.7 only has four connections less than in its form in Figure 5.6), instead it is mainly found in the fact that the abstraction helps to prevent “wrong wiring” when assembling scenarios in the Bold Stroke context.

Nevertheless, the introduction of the new connector kind and the refined component kind (and also of a new interface kind describing the interaction between the new connector and the new ports) is not necessarily a step towards a more platform specific model. Instead, the architecture model in the `prism-extended` style has to be seen as more abstract than the one in the `prism` style with regards to faithful modeling of the infrastructure or to the number of steps necessary to translate the model into concrete, deployable, code.

On the other hand it can be argued that, seeing PRiSM as a general platform for component systems and Bold Stroke as a specific development context using PRiSM, the style `prism-extended` is more specific than `prism`, since it supports the particular development of Bold Stroke more faithfully than the style `prism`.

Realizing that a transition from a PIM to a PSM can mean both, either adding or reducing information (the conceptual link between a notification connection and a data connection given through the push-pull strategy is not visible in the finished implementation, *i. e.*, this particular piece of information is reduced), CALM abstains from generally calling a migration from a parent style to a child style specialization. Instead, CALM offers to specialize by adding implementation information in form of related CALM models to architectural elements. Chapter 7 describes how abstractions, such as the one described in this section, can be introduced on multiple levels into a model without losing the tie to concrete platform implementations.

CHAPTER 7

ADVANCED TYPING OPERATIONS OF CALM

"*the psychological profiling* [of a programmer] *is mostly the ability to shift levels of abstraction, from low level to high level.*"

— Donald Knuth. in Jack Woehr. "An Interview with Donald Knuth"
(Dr. Dobb's Journal, April 1996)

Various existing component frameworks include some notion of *nesting*, which means that (sub-) architectures can reside within elements of enclosing architectures. From the perspective of a component-oriented paradigm, nesting is an apparent and trivial strategy; the very idea of components is to abstract functionality by (speaking in CALM terminology) hiding it inside a shell, it therefore naturally suggests itself as a tool to structure assemblies by hiding sub-assemblies in a similar shell as is done with base functionality.

Usually, the idea of nesting is realized by some way of "making ports available to the outside", which means that some existing assembly can be integrated into another, and the ports of the inner assembly, as far as they are available in some way (*e. g.*, if they still have potential for additional connections, or if they are explicitly assigned to be available), can be connected to fitting ports of the outer assembly. In this approach, two strategies can be distinguished. First, ports can be available to the outside of an assembly by default, in which case any assembly can be naturally considered a sub-assembly of larger architectures. Second, ports might only be available through explicit exposure, in which case an assembly needs additional specification to be usable in a larger context, but also is more protected from unintended use. One example for this second approach is nesC, where ports of inner components of an assembly can be identified with declared ports of an enclosing type through the "->" operator. Another example is J-Sim (www.j-sim.org) where ports of inner components (in J-Sim called "child components") have to be connected to ports of the shell component (in J-Sim called "parent component") through a "shadow" connection.

CALM tries to put this approach on a formal basis. To do so, CALM can build on its foundation of clearly specified interrelations of nominally declared entities. This basis allows to present a methodology of architecture nesting which is more general, more flexible, and at the same time more precise than previous approaches known to the author.

7.1 TYPING ASSEMBLIES

Obviously it is possible in CALM to integrate existing assemblies into others by some *copy-and-paste* mechanism. Given any adequate scoping method (not realized in CALM), the integrity of the integrated assemblies can also easily be maintained. In fact, various combinations of copy-and-paste with aliasing for ports, and a scoping which maintains pasted assemblies as recognizable entities, have been proposed for CADENA. Currently, CADENA features an "Add Scenario Instance" menu item on the scenario tier which allows to integrate a previously defined assembly as a single unit into a new one. Ports of the integrated assembly which are to be available to the new one have to be *exposed* explicitly.

Nevertheless, besides just integrating assemblies, CALM features a more elegant approach to the idea of nesting. To treat existing CALM assemblies as architectural elements of other CALM assemblies in a sound way, they have to be embedded into the CALM type system. Nesting is then based on inferring a type for an assembly and wrapping the assembly into a shell which represents the inferred type. The shell can then be used as an abstraction in different assemblies like any other (also abstract) architectural modeling element. Naturally, every shell can be associated with multiple assemblies as possible alternatives, analogous to the concept of normal components being defined by their shell, but free in the way they are implemented.

7.1.1 The CALM structure of an assembly

For the discussion of the typing of assemblies, recall the structure of the types of elements of the three basic categories in CALM, as discussed informally in Chapter 3 and formally in Chapter 4. Next to the kind name, interfaces are specified through their name and their map of attributes, components are specified through their name, their map of attributes, and their map of ports, and connectors are specified through their name, their map of attributes, and their map of roles.

$$\begin{array}{rcl} \text{interface type} & : & id_t^i : (id_k^i\ \alpha) \\ \text{component type} & : & id_t^c : (id_k^c\ (\alpha, \pi)) \\ \text{connector type} & : & id_t^l : (id_k^l\ (\alpha, \rho)) \end{array}$$

The same structure is given for instances of the three categories

$$\begin{array}{rcl} \text{interface instance} & : & id_v^i : (id_t^i\ \dot\alpha) \\ \text{component instance} & : & id_v^c : (id_t^c\ (\dot\alpha, \dot\pi)) \\ \text{connector instance} & : & id_v^l : (id_t^l\ (\dot\alpha, \dot\rho)) \end{array}$$

An assembly is a collection of instances of these three categories together with a link list Λ. Specifically, an assembly contains a collection of attributes given by the union of all attribute maps of its elements. Further, it contains ports, again as a collection of all ports of its elements, and similarly it contains roles. Summarizing all these elements into single maps yields a structure for assemblies

$$(\dot\alpha, \dot\pi, \dot\rho),$$

with $\dot\alpha$ being an instance level attribute mapping which is the union of all attribute mappings of the individual instances, $\dot\pi$ being, likewise, the union of all port mappings of the individual (component) instances, and $\dot\rho$ finally being the union of all role mappings of the (connector) instances of the assembly. Given this structure, it is obviously possible to infer the type level equivalent

$$(\alpha, \pi, \rho),$$

with α, π, and ρ being the type level mappings corresponding to $\dot\alpha$, $\dot\pi$, and $\dot\rho$. This would suggest to generally type an assembly with the name id according to the form

$$\text{assembly type} \quad : \quad id : (\textit{assembly-kind-name}\ (\alpha, \pi, \rho))$$

with its kind being inferred and named in some standard way. Nevertheless, this is not a qualitative step beyond a copy-and-paste approach, instead the main problem, that the typing is ad-hoc and somewhat random, persists. This inferred type does not characterize the assembly as a unit rather then a group of individual building blocks for two reasons. First, it seems much effort with little benefit to create named kinds for assemblies, for once since an assembly can comply to multiple styles (which makes it hard to determine the most sensible style to house the kind) and further since there is no unique way to turn ports into port options or roles into role options, and finally, related to the ambiguities when creating port/role options, it is unclear when two different assemblies should be given the same kind and when not. Second, and more importantly, it is not even unambiguous how the basic structure on the instance level $(\dot\alpha, \dot\pi, \dot\rho)$ is collected, even before it is turned into a type level structure (α, π, ρ).

7.1.2 The type spectrum

The problem of finding a unique type for an assembly springs from the existing links between elements of the assembly. As opposed to architectural elements from the three base categories, whose connectivity options are constrained at the style level, an assembly might have additional constraints introduced or existing constraints relaxed through the standing connections on the scenario tier. On the style tier, connectivity constraints are given mainly through the port and role options' multiplexity, which defines minimum and maximum fan-out for the respective ports. Existing links on the scenario tier shift the multiplexities and therefore change the minimum and maximum fan-out of ports and roles which is visible from the outside (see examples in Table 7.1).

	Given multiplexity	Number of connections	Outside-visible
narrower constraint	`[0..5]`	3	`[0..2]`
relaxed constraint	`[3..*]`	4	`[0..*]`
invariant constraint	`[0..*]`	1	`[0..*]`
closed off	`[0..5]`	5	`[0]`

Table 7.1: Examples for shifting multiplexities (see definition in *sec.* 4.2.1)

Naturally, ports or roles whose connectivity options are completely used internally in an assembly (such as the last example in *tbl.* 7.1) are of little interest when determining a type for an assembly. In fact, including such ports/roles into the type of an assembly would mean to encode information about the internals of the assembly which is irrelevant to the assembly's environment. At the same time, ports/roles whose minimum connectivity requirements are fulfilled internally (*i. e.*, their shifted multiplexity being `[0..`n`]` for some number $n > 0$), do not necessarily have to be included into the type of an assembly, since the concept of a shell only demands context requirements to be stated explicitly, context options can be omitted.

CALM defines two concepts to capture the significance of ports and roles for inferring the type of an assembly. Both concepts are based on the term *adjusted multiplexity* which is the multiplexity of a port/role shifted by the number of existing connections.

Definition 7.1 (Open—Closed) *A port (role) with adjusted multiplexity* `[`n`..`m`]` *(*$0 \leq n \leq m$*) is called open iff* $m > 0$*, otherwise it is called closed.*

Definition 7.2 (Complete—Incomplete) *A port (role) with adjusted multiplexity* `[`n`..`m`]` *(*$0 \leq n \leq m$*) is called complete if* $n = 0$*, otherwise it is called incomplete.*

An open port/role represents connectivity potential of an assembly and therefore *may* be included in the type, an incomplete port/role represents connectivity requirements and therefore *must* be included in the type. The overall impact of the open—closed and complete—incomplete properties on whether a port or role has to be included into the type of an assembly is displayed in Table 7.2.

	Open	Closed
Complete	*may* be included	*cannot* be included
Incomplete	*must* be included	*error*

Table 7.2: Open—closed, complete—incomplete, and the significance for the type spectrum

In summary, an assembly in CALM does not have a single type, but rather a *type spectrum*. This type spectrum contains as a minimum all incomplete ports (π_{min}) and roles (ρ_{min}) and as a maximum all open ports (π_{max}) and roles (ρ_{max}). Any mappings π, ρ with

$$\begin{array}{ccccc} \pi_{min} & \subseteq & \pi & \subseteq & \pi_{max} \quad \text{and} \\ \rho_{min} & \subseteq & \rho & \subseteq & \rho_{max} \end{array}$$

can be used for the type of the assembly. The exact form of the type spectrum of an assembly and how a type can be chosen within that spectrum is discussed in the following.

7.1.3 A shell from another category

In the previous sections, assemblies were implicitly assumed to be in a category of their own. Nevertheless, it was also discussed that a kind chosen for given assembly types, which would then in turn be of category "assembly", would be contingent or random, since no general unique and justifiable way of finding the most reasonable kinds for a collection of assembly types has been presented. It is however possible under circumstances to unify the type of an assembly with some declared type of one of the existing categories.

If for example the minimum port map π_{min} is empty (*i. e.*, all ports are complete), the port map can be omitted from the assembly type. The type then only contains an attribute map α and a role map ρ and therefore resembles the category connector. In summary, depending on completeness properties, the relations are as follows

$$\begin{array}{lcll} \pi_{min} = \emptyset & \rightsquigarrow & (\alpha, \rho) & \sim \text{category connector} \\ \rho_{min} = \emptyset & \rightsquigarrow & (\alpha, \pi) & \sim \text{category component} \\ \left.\begin{array}{l}\pi_{min} = \emptyset \\ \rho_{min} = \emptyset\end{array}\right\} & \rightsquigarrow & \alpha & \sim \text{category interface} \end{array}$$

Note though, that, however appealing in theory, the idea of reducing an assembly into a single interface type has not ben sufficiently explored within the CALM project as of yet to determine its practical value. Eventually the plan is to encode the protocol represented by an interface using an assembly (*e. g.*, in form of components as nodes and connectors as edges encoding a state transition system). This part falls into future work and is only mentioned here for completeness. Also note that outfitting assemblies with the shell of a component is a concept which (using different terminology, and often a less formal approach) is already present in previous work. The novelty which is introduced by CALM so far is that infrastructure abstractions can be created by outfitting an assembly with a connector shell. Examples for this approach and a discussion of its benefits are given below.

7.1.4 A wrapping example

One of the most often used library components in nesC is a timer component called ***TimerC***, which was already shown in Section 5.1.5 (*lst.* 5.18, *pp.* 72). In its original form, it contains a sub-assembly of four inner components, which are ***TimerM*** (the main control of the timer), ***ClockC*** (the actual hardware clock), ***NoLeds*** (a dummy replacing visual feedback LEDs), and ***HPLPowerManagementM*** (the power socket). Within the CALM nesC style ***nesC***, a simplified version (without LEDs and power management) was modeled by the assembly in Listing 7.1.[1]

Listing 7.1: The timer assembly in CALM (***sensornet*** module in *lst.* 5.9, *pp.* 68)

```
scenario timer_assembly includes sensornet {
  TimerControl control { };
  Clock_       hwClock { };

  nesCWire { user_side = control.clock;
             provider_side = hwClock.clock }
}
```

This CALM assembly features two ***nesCModule*** instances, which are `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***, which means the assembly has four *ports*. Also, it contains one unnamed instance of a ***nesCWire***-kind connector, through which it features two *roles*. A graphical representation of the assembly in nesC standard iconography is found in Figure 7.1(a).[2]

The multiplexity of all ports (before being connected) of types of the ***nesCModule*** kind is `[0..*]`. This multiplexity stays invariant also for the two ports `hwClock.clock` and `control.clock` which are connected inside the assembly.

$$\texttt{[0..*]} - 1 = \texttt{[0..*]}$$

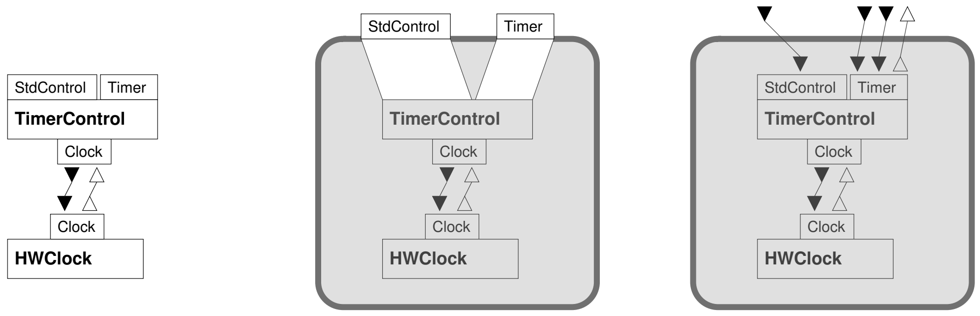
[1]Components such as LEDs or power are often omitted in nesC tutorials.

[2]Note that nesC iconography displays single connectors as "bundles" of lines if the connected interfaces contain multiple commands or events.

The situation is different for the two roles *unnamed*.`user_side` and *unnamed*.`provider_side`. Their declared multiplexity is `[1]`, and each of them has one connection, their adjusted multiplexity hence being `[0]`.

$$\texttt{[1]} - 1 = \texttt{[0]}$$

Therefore, the four ports are open, π_{max} contains all four of them; also they are complete, π_{min} is empty. Further, the two roles are closed, so $\rho_{min} = \rho_{max} = \emptyset$.



(a) the timer assembly (b) as component (c) as connector (service)

Figure 7.1: Wrapping the timer-assembly

Given these constraints, it is for example possible to choose a shell of category component. If in this specific case the ports `control.init` of type ***`StdControl`*** and `control.timer` of type ***`Timer`*** are chosen for π (and considering that α happens to be empty), the resulting (component) type corresponds exactly to the ***`nesCModule`*** type `Timer_` (compare to module in *lst.* 5.9, *pp.* 68).

```
  nesCModule Timer_ {
    provides init  : StdControl;
    provides timer : Timer }
```

This example provides the intuition for a definition of the type spectrum of an assembly which does not assume a category "assembly":

Definition 7.3 (Type Spectrum) *The type spectrum of an assembly is the set of all component, connector, and interface types which can be chosen as the type of the assembly.*

To associate an assembly with a declared type which is element of the type spectrum of the assembly, CALM uses the concept of *wrapping*. A wrap exposes ports (or, in the case of connectors, roles) and identifies them with ports (roles) of an existing component/connector type (*i. e.*, a shell), thereby naming the assembly as a possible implementation of the shell. In the case of the discussed timer assembly the respective wrap which identifies it as an implementation of the ***`nesCModule`*** type `Timer_` from the module `sensornet` is

```
implementation HWTimer :
  wrap timer_assembly into sensornet.Timer_ {
    expose init  = control.init;
    expose timer = control.timer };
```

with `HWTimer` being the name of the wrap. The wrap is also illustrated in Figure 7.1(b), a view of the same wrap in CADENAis offered in Figure 7.2.

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.* 7.1(c)). As a result, the two unconnected roles become minimum members of every type in the type spectrum, the assembly now has to be typed as a connector by exposing only the two open roles, and hiding the (complete) ports.

7.1.5 EXTENDING THE SERVICE LAYER

Unary connectors or better *service* connectors such as the timer described above (which has two roles, but does not "connect" different components) have turned out to be an elegant means to abstract services provided

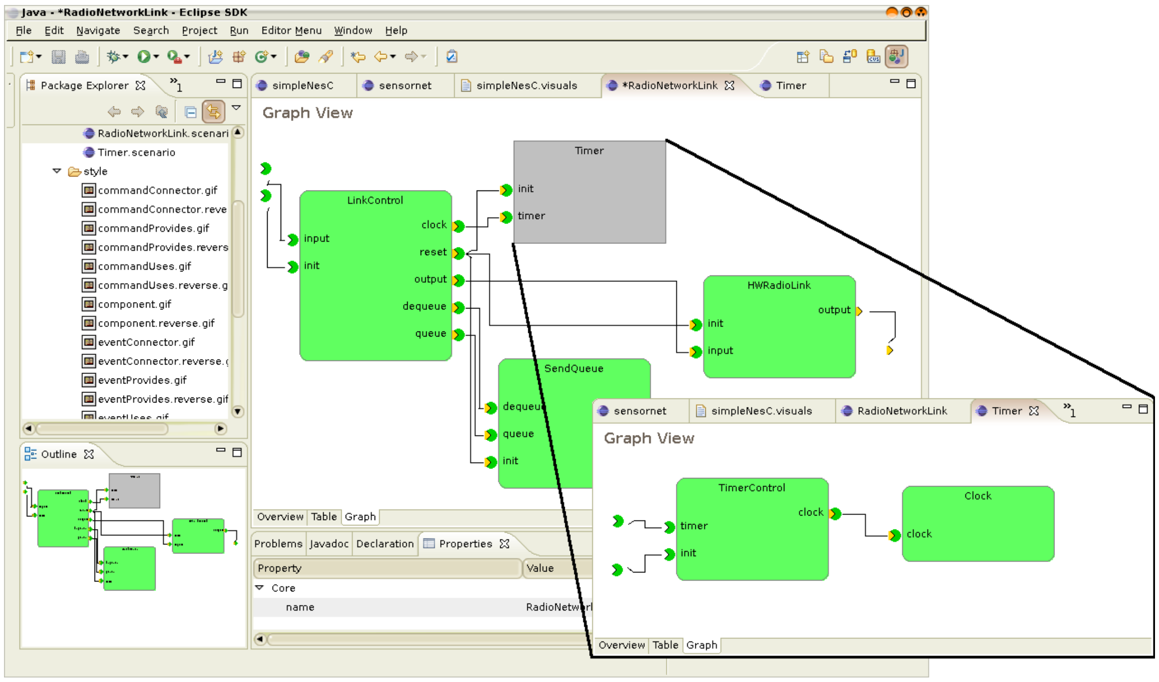


Figure 7.2: The Timer assembly modeled as component of Radio-Link in CADENA

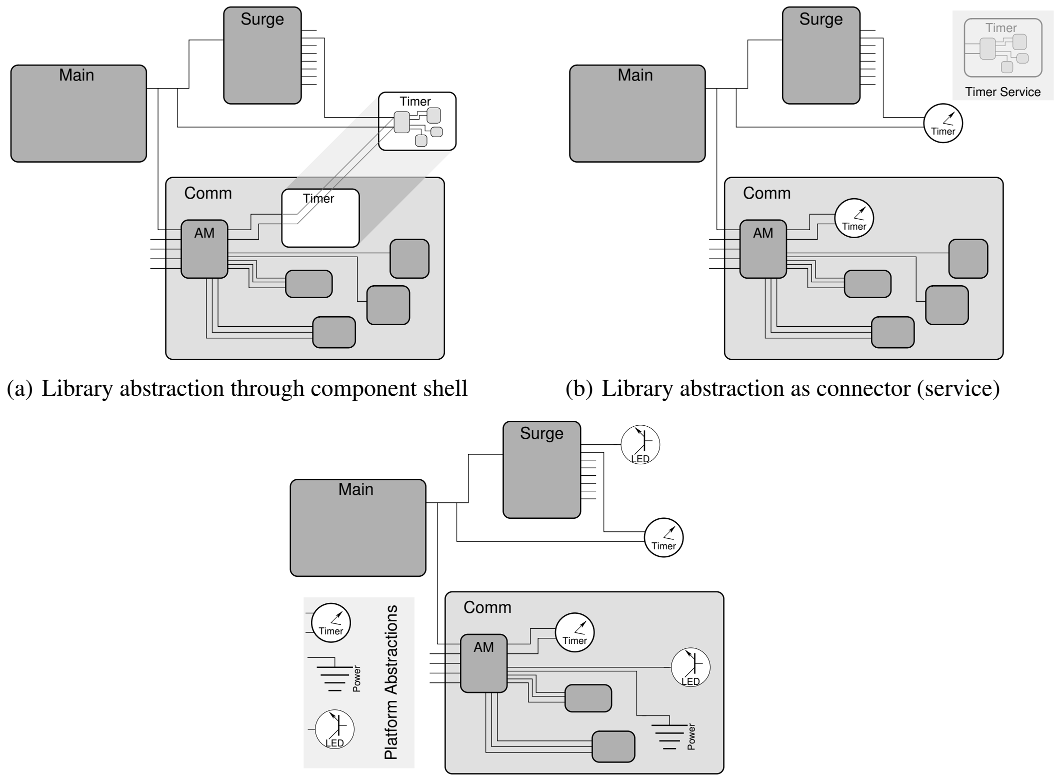
by library functions such as the timer. They also serve to structure assemblies conceptually and to narrow the focus even more towards functional aspects and away from low level infrastructure.

As an example of the potential of service abstractions in a component oriented environment, recall the discrepancy between standard CALM semantics and the approach of nesC which is exposed in Section 5.1.5. The approach to resolving this discrepancy as proposed by CALM is exemplified in Figure 7.3, which shows an essential part of the nesC Surge application (see also listings in *apx.* C.1) with emphasis on the `Timer` component.

Figure 7.3(a) illustrates how a connection to the `Timer` component is seen internally in nesC. The shell of the timer is present in multiple places, but its functionality is really instantiated only once. The `Timer` inside the `Comm` component therefore maps to the same instance as the one in the global Surge application. Two ports are offered by the `Timer` component, one featuring the `StdControl` interface, the other the `Timer` interface. The `Timer` interface is implemented in nesC in a way that it distinguishes the different connections, which means that commands coming in from two different connections do not influence each other, events sent on these connections reflect only the respective commands received on that same connector. In summary, the observable behavior of the `Timer` port with multiple connections is equivalent to multiple `Timer` components. For the `StdControl` port the opposite is true. `StdControl` is essentially a reset button, it is usually only invoked at the start or re-start of an application. In the case of the Surge application, the observable behavior of a single *vs.* multiple instantiations of `Timer` is still equivalent, but only by coincidence, since the reset signal arriving at the `StdControl` port of the global `Timer` originates from the `Main` component, and so does the one on the `Timer` inside the `Comm` component (routed through the `AM` component), so essentially the sole instance of `Timer` is connected twice to the same reset source. Nevertheless there is no guarantee that the observable behavior is always indistinguishable as to the number of instantiations. In the case of the `Timer`, developers instead must keep in mind that the `StdControl` port is "global", while the `Timer` port is local.[3]

The alternative to having multiple component shells which might or might not influence each other (a breach of the component oriented paradigm!) is defining the timer as a connector, which, as described above, means as a service of the infrastructure (7.3(b)). The advantage lies (besides allowing the assembly

[3]To be precise, the `Timer` port is not local to the instance of the `Timer` shell, rather its locality is decided by numeric identifiers. To have a "local" connection to the `Timer` port, a unique ID has to be generated.



(a) Library abstraction through component shell (b) Library abstraction as connector (service)

(c) Multiple library abstractions

Figure 7.3: Hidden (abstracted) semantics of library services

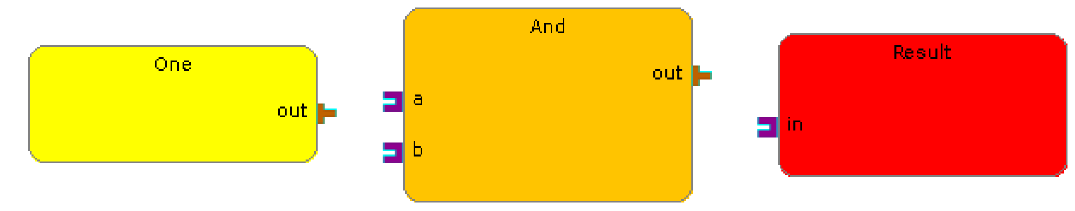
to be clearer) mainly in the intuition. The `StdControl` role on such a service can be declared as a reset for the service, while using the service can be local. This approach can be applied to all sorts of often used functionality and services (*fig.* 7.3(c) also abstracts the power connector and the LED visual feedback). Another advantage (specifically in nesC) is that, combined with the possibilities of style refinement and specialization of models as described in Chapter 6, the abstraction of (library) functionality as services allows to even more precisely hand tailor styles to platforms and product lines (*e. g.*, by abstracting exactly those library functions into services which are implemented for specific device cores or motes and summarizing them in a device-core-specific architectural style, to and from which models can be migrated).

Finally, note that the difficulty in fitting the introduction of the push-pull connector in the PRiSM style `prism-extended` into the OMG proposed PIM to PSM structure (discussed in *sec.* 6.3) can be resolved easily, if the newly introduced connector is considered as a shell for an "assembly" which consists of two parallel connectors, one synchronous and one asynchronous. This way, the information about how to realize the push-pull connector is still contained in the architectural model, which then can be clearly considered more platform specific (and *product specific*) then the style `prism`.

7.1.6 Structural pitfalls

Unfortunately, not always can a shell from a base category be found for a given assembly. An example where an assembly which warrants abstraction (potential for frequent re-use) cannot b outfitted with a shell because no appropriate kind is present is the `LogicalGates` style which models abstract semiconductor circuits.

`LogicalGates` features three kinds of components which are exemplified in Figure 7.4. The component kind ***`source`*** must have exactly one provided port (*fig.* 7.4(a)), ***`gate`*** can have any number of used ports and must have a single provided port (*fig.* 7.4(b)), and finally ***`sink`*** must have a single used port (*fig.* 7.4(c)). The single interface kind ***`Plug`*** represents the contacts of the semiconductors. Multiplexity constraints model the requirements of a common semiconductor circuit, for example the multiplexity of used (input-) ports is



(a) A source component (b) A gate component (c) A sink component

Figure 7.4: Examples of the component kinds in the LogicalGates style

[1], which means a minimum of one since unconnected input contacts tend to produce constant high-values because of the missing mass and a maximum of one because multiply connected inputs can lead to contingent behavior of the circuit. Further, the multiplexity of the provided (output-) contacts is [1..*], which means at least one since every signal has to be terminated. Multiple connections are not a problem for output ports.[4]

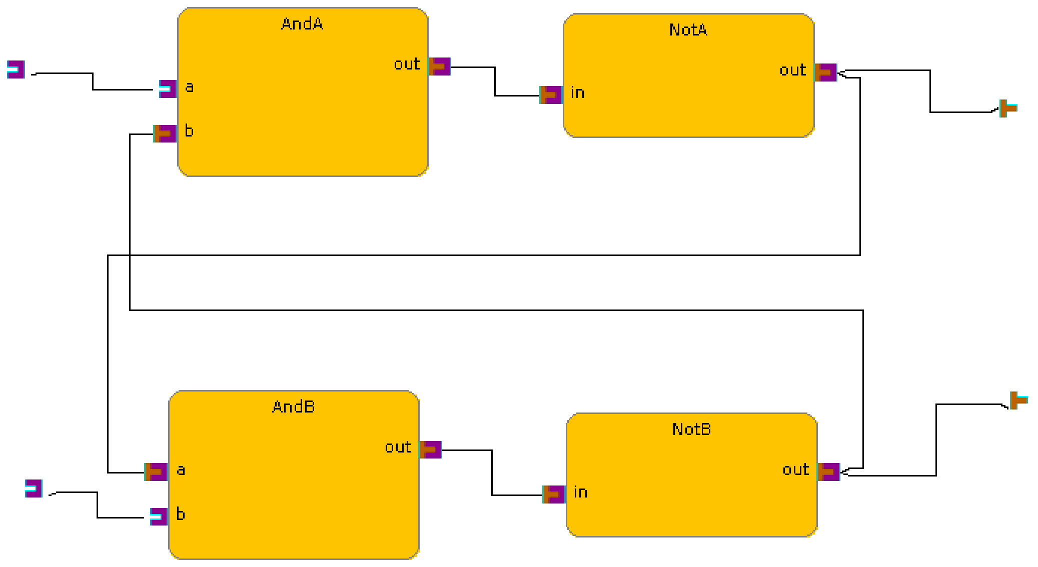


Figure 7.5: A flip-flop circuit realized in LogicalGates

Figure 7.5 puts four ***gate*** components together. Two And gates and two Not gates are assembled to form the standard flip-flop (or "alternating solid state") circuit. The contacts which are supposed to be visible to the outside of this assembly are shown as "dangling" ports. Figure 7.6 embeds the flip-flop circuit into a simple environment, and aliases the ports (Set, UnSet, Q, and NotQ). Nevertheless, this abstraction cannot be typed with a component shell: It has two input ports and therefore cannot be a ***source***, and two output ports and hence can neither be a ***gate*** nor a ***sink***. Still, the assembly can be used as a typed *component architectural pattern*. In this context, recall the "Add Scenario Instance" menu item of CADENA mentioned in the introduction to this section.

7.1.7 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

[4]For realistic semiconductors, the output would actually only support connections up to some certain maximum which can range between about four to up to a hundred depending on the technology used. Of course, the output multiplexity can be adjusted to given maxima.

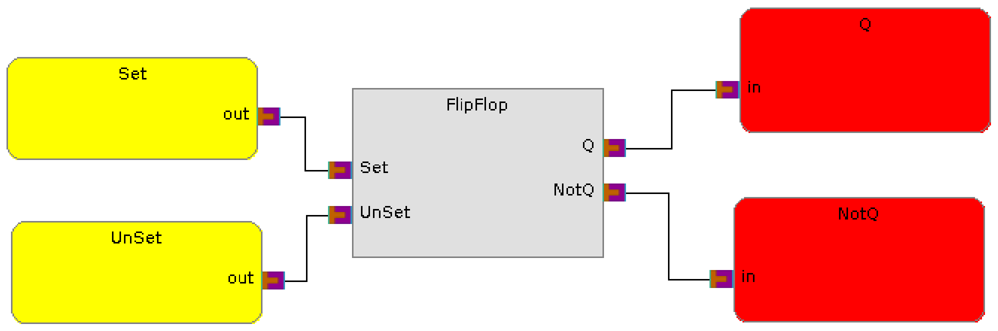


Figure 7.6: The flip-flop assembly used in realizing a one-bit register

associated with the architectural elements have to be easily *exchangeable*. Therefore, CALM separates the topology from the specification of the components' and connectors' internals.

Terms and Names

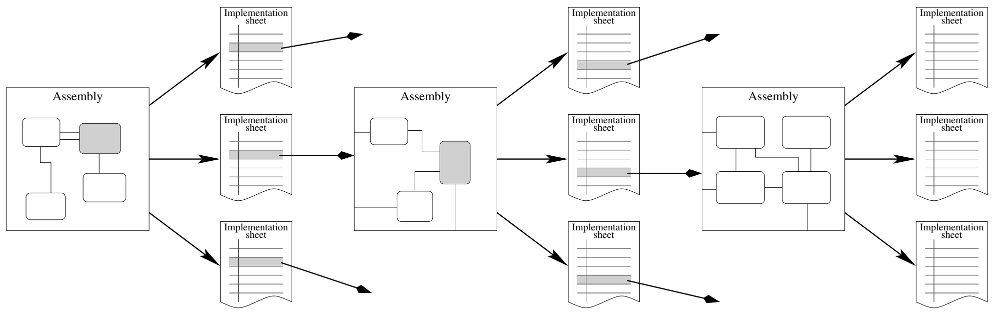
In Section 3.3.1, a distinction is introduced between the term *assembly* and *scenario*. The term assembly is used for the topology, whereas the scenario is described as an assembly with additional (meta-) data. Since data is added when moving from an assembly to a scenario, there can be multiple scenarios for every assembly. The connection between the scenario and the assembly is the *implementation table*.

An assembly is a collection of interconnected shells. The implementation table maps every element of an architectural assembly to an implementation. Possible implementations can be other (sub-) assemblies which, by means of wraps, are identified to have the type of the respective shell, or *atomic* implementations, which are references to source code implementations which are outside of the scope of CALM (but can be handled through the plugin system of CADENA).

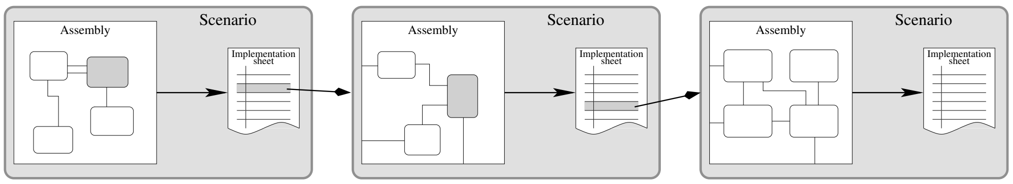
To support incremental supply of the meta-data, CALM distinguishes different stages of completeness of a scenario. A scenario is said to describe a *system*, iff all multiplexity constraints of the underlying assembly are met (*i. e.*, no connections have to be added), otherwise it is said to be a *subsystem*. Further, a scenario is said to be *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 *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 *abstract*, which means that they might be still incomplete at some level of nesting, or *deployable*, which is the equivalent of recursive completeness.

Top-down and bottom-up specification

CALM suggests two strategies of assigning sub-scenarios (as opposed to atomic implementations) as implementations to architectural elements (*fig.* 7.7). The first strategy is to record *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 *top-down* approach since first the global assembly is outfitted with implementation data, and then recursively the nested assemblies are handled. Other intuitions for this strategy are *branching*, *call-by-name*, or *lazy* specification. Figure 7.7(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.



(a) Top-down



(b) Bottom-up

Figure 7.7: Two strategies to define scenarios

The second strategy is to record *scenarios* in the implementation table. This approach can be thought of as *bottom-up* specification, since when choosing an implementation for an element of a scenario, all recursive implementations are already given. Other intuitions are *linear*, *call-by-value*, or *eager* specification. This strategy is illustrated in Figure 7.7(b). Instead of a tree of implementation sheets, the single implementation sheet on the top level describes the deployable (sub-) scenario.

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 to concisely specify (incomplete) multi-level topologies which manifest the concept of *reference architectures*.

FORM AND CONTENT

An implementation table has to contain one entry for each architectural element of the assembly it describes. In practice, interfaces are naturally included with the element which provides them. Connectors often have a standard or automatic implementation and therefore their entries might be supplied by the supporting tool (CADENA).

An entry for a component (or connector) so far can contain one out of two entities. First, it can contain a reference to some atomic implementation. The term atomic does not necessarily describe a monolithic or indivisible unit, it merely means an entity outside the scope of the CALM model itself and therefore indivisible from the perspective of CALM (*e. g.*, source code, hardware, *etc.*). Therefore the atomic implementations are the leave nodes of a nested implementation specification.[5]

Second, the entry can contain a reference to an artefact within the scope of CALM. In this second case, the entry has to specify an assembly and a mapping between ports or roles of that assembly to ports and roles of the shell (component or connector) to which the entry belongs. To do so, it references a wrap (*sec.* 7.1.3, 7.1.4) which provides exactly that information. Further, if using the bottom-up approach, the entry has to specify the next level implementation sheet for the referenced assembly to turn it into a scenario (in the

[5]As mentioned before, although atomic implementations are outside the scope of CALM, the CADENA plugin system can offer extensive support such as source code generation, deployment support, *etc.*.

top-down approach the implementation sheets are “factored out”). The possible entries are summarized in Table 7.3.

element-name	$\mapsto$	*atomic implementation*	A leave node entry
element-name	$\mapsto$	*wrap*	A top-down CALM artefact implementation (specifying an assembly)
element-name	$\mapsto$	*wrap + implementation-sheet*	A bottom-up CALM artefact implementation (specifying a scenario)

Table 7.3: Implementation table entries

7.2 Inter-style wrapping

So far, nesting of architecture models has been discussed as a means of organizing assemblies within a single architectural style, and as an elegant way to abstract recurring functionality into complex platform extensions. Further, it has been shown how CALM separates an architectural topology (assembly) from the intended implementations (implementation table) to fully exploit the flexibility of the component oriented paradigm within CALM scenarios.

This section tries to extend the notion of sub-architectures beyond hierarchical organization of assemblies and their implementations towards the concept of domain specific layers.

7.2.1 Components and layering

The concept of *layering* is very common in system development. For example, computing systems are often structured by more or less the following layers: hardware, basic input-output system (BIOS), operating system (OS), application layer, graphical user interface (GUI). Another well known and widely used example is the Open Systems Interconnection (OSI) seven layer reference model of the International Organization for Standardization (ISO) [34], which divides network communication implementations into physical layer, data link layer, network layer, transport layer, session layer, presentation layer, and application layer.

A defining characteristic of a layered system are the layering constraints, which normally express the basic requirement that programs of one layer can only access functionality within the same layer or of the layer immediately below (*e. g.*, in ISO/OSI the physical layer can only be accessed by the data link layer, the data link layer only by the network layer, *etc.*). Nevertheless, it is often advantageous or necessary to shortcut through layers to reduce overhead or to access functionality which would otherwise be covered up by intermediate layers. For example, most personal computing platforms offer some way to cut through GUI, application, and OS layer to directly access hardware (*e. g.*, directX or simple direct media SDL, which both access hardware acceleration features of graphic hardware circumventing the GUI layer). Also, layers are not always stacked perfectly on top of each other, sometimes they rather exist in parallel with overlapping functionality (such as the BIOS and the OS layer, which exchange responsibilities depending on the stage of the boot or shutdown sequence). Also, layers are not always stacked in a linear way, instead they are most often multi-dimensional, For example, a network application on a personal computer resides on top of both, the network stack and the hardware-BIOS-OS-GUI stack, with various interdependencies.

While layer constraints are often treated in a lax way to enable various optimizations, the main reason divide systems in layers is the fact that this organization often closely reflects distinguishable fields of expertise. For example, the physical layer in the communication stack focusses on signal processing and hardware (pins, voltages, cable specifications, adapters, *etc.*), whereas the data-link layer is concerned with flow control, sequencing, error correction, an so on. In short, each layer represents a specific *domain knowledge*. The abstractions presented to higher layers reflect the domain specific functionality.

CALM approaches system layering through nested architectures. The basic idea to this approach is to provide functionality of lower layers as services or components and thereby offer hand-tailored architectural styles and infrastructure abstractions for each layer which encode access capabilities and layering constraints in a component oriented way. To do so, CALM has to present the option of wrapping artefacts which represent different domain knowledge into a (component or connector) shell. Together with the fact that domain specific modeling contexts are expressed in CALM through architectural styles, this means that wrapping in CALM needs to be extended to transition architectural styles.

7.2.2 Coercion

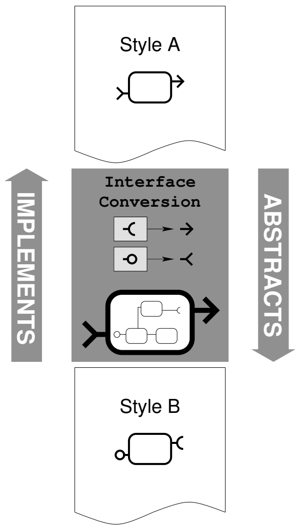


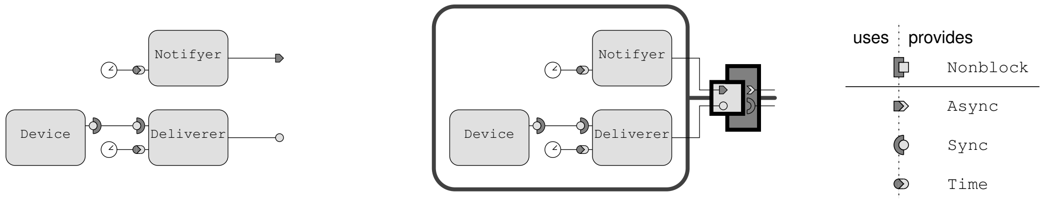
Figure 7.8: Layering: implementation/abstraction relation between styles

Wrapping an assembly into a shell of the same style as the assembly is straightforward: ports or roles of the assembly have to be identified with ports or roles of the shell. The problem which arises when crossing style boundaries is that ports and roles of shell and assembly do not feature interfaces from the same architectural style anymore. Therefore, a conversion has to be defined which specifies how ports/roles of the assembly *implement* ports/roles of the shell, while ports/roles of the shell abstract ports/roles of the assembly (*fig.* 7.8).

As a quick example recall the push-pull connector introduced in the style `prism-extended` as an extension of the style `prism` in Section 6.3 and further discussed at the end of Section 7.1.5. Figure 7.9(a) illustrates a small, simplified, `prism` assembly with a device component, a notifier component, and a data-delivery component. The used asynchronous ***`PrismEvent`*** port of the notifier (which is also connected to the ***`PrismTimeout`***, which is shown as a service in the figure) and the provided synchronous ***`PrismInterface`*** port of the data-delivery component (which also connects to the device component and the ***`PrismTimeout`*** service) are conceptually coupled to form a push-pull interface. Figure 7.9(b) shows the same assembly outfitted with a shell from the style `prism-extended`. The new, provided, non-blocking, push-pull interface has to be implemented by the used asynchronous port of the notifier component and the provided synchronous port of the data-delivery component.

The example exposes multiple important aspects of a port (role) conversion. First, a single port of the shell can be implemented by more than one port of the assembly. Further, the multiplexities of each port of the assembly involved in the implementation all have to contain the multiplexity of the implemented port on the shell. Also, ports can participate with *covariant* parities (*i. e.*, a used port participates in implementing a used port, a provided port participates in implementing an provided port) as well as with *contravariant* parities (*i. e.*, a used port participates in implementing a provided port and *vice versa*).

CALM requires conversions (*i. e.*, the actual translation of a group of implementing ports/roles into an abstracting port/role and back) to be enabled by a respective *coercion* on the style tier. A coercion is a style-level (kind-level) declaration which specifies how interfaces can be translated over different styles, which



(a) A base assembly (b) Wrapped with interface conversion

Figure 7.9: Inter-style wrapping through interface conversion

Listing 7.4: PRiSM to PRiSM-extended coercion

```
coercion prism_to_prism-extended :
  from prism build prism-extended.PrismPushPullInterface {
    uses      [1] notify : PrismEvent;
    provides [1] data   : PrismInterface }
```

means they capture knowledge about the abstractions which the (nominally declared) interfaces are supposed to stand for.

Because the concepts of coercions and conversions defined by them and wraps using the conversions are very similar to the way kinds, types, and instances relate in the CALM core language, the explanation is kept informal and centered around the `prism` to `prism-extended` example. The coercion for the example is given in Listing 7.4. In the listing, name of the coercion (`prism_to_prism-extended`) is given in Line 1. Line 2 specifies the interface kind (in this case the ***PrismPushPullInterface*** of the style ***prism-extended***) whose build is defined by this coercion, and the style (***prism***) from which the constituents of the interface can be drawn. Lines 3 and 4 contain the body of the coercion. Generally, the form of a coercion is

$$\texttt{\textbf{coercion}}\ id\ \texttt{:}\ \texttt{\textbf{from}}\ id_0\ \texttt{\textbf{build}}\ id_1\,\texttt{.}\,id_2\ \texttt{\{}\ b\ \texttt{\}},$$

with id being the name of the coercion, id_0 the implementing style, id_1 the style which holds the interface to be implemented, and id_2 the interface kind name. The body b of a coercion contains a semicolon-separated list of options of how interfaces from style id_0 (given as kinds) can participate in a conversion. Each such option generally has the form (closely resembling port/role options)

$$p\ q\ id\ \texttt{:}\ id_0,$$

with p being a parity, meant for provided ports (*i. e.*, **`provides`** stands for covariant parity, **`uses`** for contravariant parity), $q \in \mathcal{Q}$ being a range specifying the multiplicity, which (similar to port/role options) specifies minimum and maximum number of respective interfaces to be involved, id declaring a module level keyword to specify the actual interfaces, and id_0 specifying the interface kind from which they are drawn.

Listing 7.5: PRiSM to PRiSM-extended conversion

```
prism_to_prism-extended PushPullConversion :
  render modalsp-extended.PushPull-ReadData in modalsp {
    notify available : DataAvailable;
    data    deliver    : ReadData }
```

Much like with the elements of CALM's core framework, the coercion defines a language of coercions. Listing 7.5 shows an application of the coercion from Listing 7.4. The conversion is called with the name of the coercion which enables it. In its body it uses the keywords introduced by the coercion. Generally, the form of a conversion is

$$id_c\ id\ \texttt{:}\ \texttt{\textbf{render}}\ id_0\,\texttt{.}\,id_1\ \texttt{\textbf{in}}\ id_2\ \texttt{\{}\ b\ \texttt{\}},$$

with id_c being the name of a coercion and id being the name of the conversion. The qualified name $id_0\,\texttt{.}\,id_1$ gives module (id_0) and interface type (id_1) to be implemented, and id_2 is the module holding the interfaces used for the implementation. The body b contains a semicolon-separated list of element interface specifications of the form

$$id_0\ id_1 : id_2,$$

where id_0 is a keyword for the option defined by the coercion, id_1 is the name of the interface within the conversion, and id_2 gives the interface type, which has to be of the kind defined by the coercion for this respective option.

Listing 7.6: PRiSM to PRiSM-extended conversion

```
implementation PushPullConnector :
  wrap push_pull_assembly into modalsp-extended.PushPull-ReadData {
    expose provides_side = PushPullConversion {
      available = unnamed_PrismEventConnector_001.usesSide;
      deliver   = unnamed_PrismInterfaceConnector_001.providesSide };
    expose uses_side = PushPullConversion {
      available = unnamed_PrismEventConnector_001.providesSide;
      deliver   = unnamed_PrismInterfaceConnector_001.usesSide }
  }
```

Finally, Listing 7.6 shows an application of the ***PushPullConversion***. The `push_pull_assembly` (not shown, it contains one ***PrismEventConnector*** and one parallel ***PrismInterfaceConnector***) is wrapped into a single connector called `PushPullConnector`, which features two role (`provides_side` and `uses_side`). The roles are identified with pairs of roles from the assembly converted by using the ***PushPullConversion***.

7.2.3 A LAYERING EXAMPLE

Using the main example laid out in Chapter 2, this section illustrates how a CALM assembly can be layered for example to correspond to the three lowest tiers (called "media layers") of the ISO/OSI model. Recall that a conceptual architecture was presented without any specific component framework associated to it. Listing 7.7 shows a possible style called `global_a` which allows to turn the abstract architecture into a concrete CALM artefact. It defines three simple kinds, the interface kind `Plug`, the component kind `Box`, and the connector kind `Link`. Obviously, this style allows to draw the simple box-diagram given in Figure 2.1 (*pp.* 7).

Listing 7.7: A conceptual architecture style

```
style global_a {
  metainterface mPlug { };
  interfacekind Plug : mPlug { };

  metacomponent mBox {
    attribute name : SCENARIO STRING;
    provides [0..*] in  : mPlug [0..*];
    uses     [0..*] out : mPlug [0..*] };
  componentkind Box : mBox {
    in -> Plug;
    out -> Plug };

  metaconnector mLink {
    uses     [1] source : mPlug [1];
    provides [1] sink   : mPlug [1] };
  connectorkind Link : mLink {
    typevar a : Plug;
    source -> a;
    sink -> a }
}
```

As a refinement step, Listing 7.8 introduces a distinction between local connections and remote connections by defining the connector kinds `LocalLink` and `RemoteLink`.

`RemoteLink` connectors describe communication channels between parts of the system which are not collocated. This functionality is generally associated with the network layer of the ISO/OSI model (connectivity and routing services).

Listing 7.8: A conceptual architecture style, refined

```
style global_b extends global_a {
  elide Link;
  metaconnector LocalLink : mLink {
    typevar a : Plug;
    source -> a;
    sink   -> a }
  metaconnector RemoteLink : mLink {
    typevar a : Plug;
    source -> a;
    sink   -> a }
}
```

The network layer relies on the data-link layer which, along the routed path, is responsible for secure transfer and error correction. The example in Chapter 2 offers a straightforward assembly for a buffered send with re-send capability, consisting of a controller, a send queue for buffering of data, a timer which allows to determine re-send timeouts, and a communication component being capable of sending data and receiving acknowledgements over a wireless connection. The assembly is realized in nested nesC (see *fig.* 2.2, *pp.* 8, which shows the main assembly and the nested timer assembly, also compare *sec.* 7.1.4 *ff.*)

Finally, the data-link layer relies on a concrete infrastructure (the actual hardware) provided by the physical layer. The nesC assembly hides the functionality of the physical layer within the `HWRadioLink` component. While it is technically possible to realize the functionality of this layer in nesC as well, the example delegates the physical layer to a more hardware specific model (*fig.* 2.3, *pp.* 10). Listing 7.9 presents a style `radioComm` which represents the radio hardware model in CALM.

Overall, the layering of the styles in CALM and their correspondence to the media layers of the ISO/OSI model is displayed in Figure 7.10. The relations between the styles are given through respective coercions, conversions, and wraps. For example between `global_b` and `nesC` the coercion

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

specifies that interfaces from `global_b` (***`Plug`***) can be expressed in terms of an arbitrary number of elements from ***`nesCInterface`***. The conversion

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

is an example for the use of the coercion. This conversion packs one ***`sensornet.Send`*** interface type and one ***`sensornet.StdControl`*** interface type into the interface type ***`sensorbank.Send`***, with ***`sensorbank`*** being a module in ***`global_b`***. The conversion allows to wrap the data-link layer assembly into a remote connector of the global style as illustrated in Figure 7.11.

By relating styles to each other through implementation/abstraction relationships which enable wrapping across architectural domains and manifest layering requirements in a component oriented way, CALM offers an elegant way to attach concrete realizations to otherwise abstract modeling artefacts such as the box-and-line diagram which captures the global architectural layout of the sensor bank system. With the style `global_a`, its refined version `global_b`, and the connection to concrete component platforms (`nesC`, `radioComm`), the box-and-line diagram turns from a picture into a development artefact which is traceably linked to the actual system implementation.

Listing 7.9: A network hardware style

```
style radioComm {
  typedef Volt       = INT[0..*];
  typedef Watt       = INT[0..*];
  typedef Modulation = ENUM { AM, FM, PK };
  typedef Freq_Unit  = ENUM { KHz, MHz };
  typedef Frequency  = struct {
    class : Freq_Unit;
    value : INT[0..*] };

  metainterface mDigital {
    attribute max_rate : MODULE INT[0..*] };
  interfacekind Digital : mDigital { };
  metainterface mSignal {
    attribute Max_Amp : MODULE Volt;
    attribute Class   : MODULE Modulation;
    attribute Band    : MODULE Frequency };
  interfacekind Signal : mSignal { };
  metainterface mWave extends mSignal {
    attribute Power : Watt };
  interfacekind Wave : mWave { };
  metainterface mPower {
    attribute Voltage : Volt };
  interfacekind Power : mPower { };

  metacomponent mController {
    provides [0..*] data_in_port  : mDigital [0..1];
    uses     [0..*] data_out_port : mDigital [0..1] };
  componentkind Controller : mController {
    data_in_port  -> Digital;
    data_out_port -> Digital };
  metacomponent mDevice extends mController {
    provides [0..*] signal_in  : mSignal [1];
    uses     [0..*] signal_out : mSignal [1] };
  componentkind Device : mDevice {
    data_in_port  -> Digital;
    data_out_port -> Digital;
    signal_in     -> Signal;
    signal_out    -> Signal };
  metacomponent mActiveDevice extends mDevice {
    uses [1] power : mPower [1] };
  componentkind ActiveDevice : mActiveDevice {
    data_in_port  -> Digital;
    data_out_port -> Digital;
    signal_in     -> Signal;
    signal_out    -> Signal;
    power:        -> Power };
  metacomponent mSupply extends mController {
    provides [1..*] power : mPower [0..*] };
  componentkind Supply : mSupply {
    data_in_port  -> Digital;
    data_out_port -> Digital;
    power         -> Power };

  metaconnector mBus {
    provides [1] source : mDigital [1];
    uses     [1] sink   : mDigital [1] };
  connectorkind Bus : mBus {
    typevar a : Digital;
    source -> a;
    sink -> a };
}
```

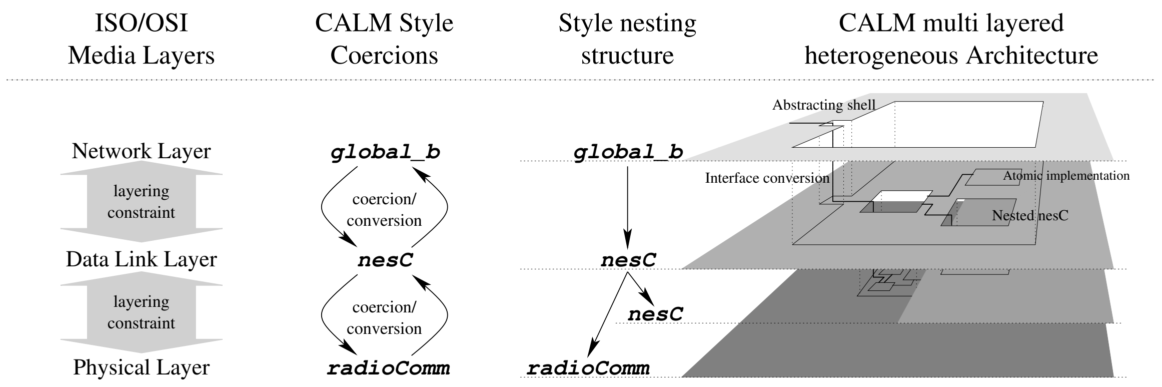


Figure 7.10: Constraining inter-style nesting

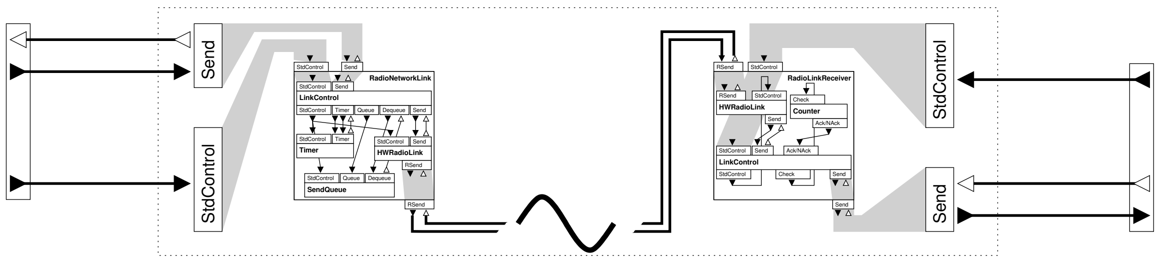


Figure 7.11: Network-assembly as connector

Chapter 8

Related Work

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”

— Charles Antony Richard Hoare. 1980 Turing Award Lecture.

8.1 Industrial tools

In current industrial practice, initial design (and commercial tool-support for design) is almost exclusively based on UML (www.uml.org). Over one hundred tools from academia, industry, and open-source projects support subsets of UML.

UML is an extremely extensive set of modeling languages mixing graphic elements with keywords and symbols. UML draws its generality and wide applicability (particularly in the context of object oriented programming) from this complexity, having a specific modeling construct for almost any situation. Unfortunately, the complexity of UML (a now already obsolete summary of the different notations from 1997 alone is a 148-page document [63], more recent summaries on notation were not found) make it difficult to identify clear concepts or determine applicable methodologies.

UML 2.0, the most recent main version, defines thirteen types of diagrams, divided into three categories. *Structure Diagrams* include the Class Diagram, which might be the most widely used UML specification, the Object Diagram, the Component Diagram, the Composite Structure Diagram, the Package Diagram, and the Deployment Diagram. *Behavior Diagrams* include the Use Case Diagram, used to gather requirements, the State Machine Diagram, which defines state transition systems, and the Activity Diagram, which is a specialization of the State Machine Diagram, expressing actions as states and activities which follow on completion of actions as edges. *Interaction Diagrams*, all derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram. Modeling elements of any type of diagram can appear in almost any other type of diagram, again adding to the complexity of UML modeling (UML 2.0 introduces some restrictions).

Most UML diagrams and specifications were designed directly for the task of modeling object oriented organizational patterns and techniques (Class Diagram, Object Diagram, Package Diagram, to some extend the Composite Structure Diagram, *etc.*), but a shift is made (in UML 2.0 more than in previous versions) towards capturing component oriented design principles. UML features modeling elements called components and named interfaces (primarily as elements of the Component Diagram). Components (in UML 2.0) are rectangles marked by the *stereotype* “⊐”, or alternatively by the word “component”. Interfaces are given by the symbol “−○”, if they are provided, or (new in UML 2.0) by “−(” for used, or (in UML terminology) *required*, interfaces.

Although UML is recommended by the OMG for the OMG MDA approach (see also 6.1) to specify platform independent models (PIM), no notion of connectors is given (*i. e.*, interfaces on component ports simply plug into each other). Neither components nor interfaces are kinded, and typing is very basic, which both makes it difficult to concisely capture platform requirements and restrictions. Higher-level architectural subsystem and layering constraints might be possible, especially since with the object constraint language (OCL) [64] a first order logic constraint language is included in many UML settings, but a direct support which only requires feasible effort is lacking. To move from the PIM to the platform specific model (PSM) OMG's MDA guide [53] recommends using one of the following modeling languages: custom meta-object facility (MOF) meta-models (www.omg.org/mof/), UML extended with profiles, or custom MOF meta-models extended with UML profiles [23]. CALM by no means claims to subsume capabilities of UML and MOF, because the focus of CALM only lies on modeling component-based architectures (platforms, libraries of types, and system scenarios). Nevertheless, the generality of the UML/MOF approach makes it hard to assert a system's internal consistency, the burden for this lying with the user. CALM in contrast is able to leverage the more limited context and provide a structured, incremental specification approach to platform model refinement through tailored language constructs for style refinement, omitting modeling concepts which are object oriented and problematic in a component oriented setting. For example, beyond concepts that can be easily captured in CALM, the UML Component Diagram also exhibits *inheritance* relations which capture a structural subtyping on components, but it is unclear how the problem of the insufficiency of structural subtyping in a component oriented setting (discussed in *sec.* 4.3.4) is addressed.

Further, the most widely used tools, such as Rhapsody and Rational Modeler only provide limited aspects of even the most basic typing capabilities (*e. g.*, that interface instances have to conform to types, typed interfaces on ports, or type-correct connections). Instead, tool support often focuses on lower-level class architectures, restricting the tools' applicability to object oriented system organization.

8.2 AADL

AADL [20, 22, 21] (www.aadl.info) is an architectural modeling and analysis framework based on MetaH [4]. It was initially developed under the name Avionics Architecture Description Language but was later renamed to Architecture Analysis & Design Language to account for the focus on analysis and for the applicability beyond the avionics domain. AADL strives to capture real-time embedded systems, complex systems of systems, and specialized performance capability systems. Similar with CALM, the overall aim of AADL is to serve for effective model-based design and analysis of architectures, but instead of having a manipulable architecture model like CALM, AADL uses a rich but fixed set of highly specialized modeling entities. The emphasis of AADL lies less in the modeling than in the built-in, fine-tuned, support for analysis capabilities.

The focus of AADL lies on the real-time aspects of complex systems. As a strategy to make the real-time behavior explicit within the modeling framework, AADL offers separate primitives for hardware entities and software abstractions and features a fixed set of options on how to associate both.

Being highly successful in the domain of avionics and generally in complex real-time systems, AADL was adopted and standardized by the Society of Automotive Engineers (SAE, www.sae.org) in 2004.

8.2.1 Abstractions and Primitives

AADL does not include an architectural meta-layer comparable to the CALM style. Nevertheless it features, for example, a rich set of predefined component kinds which are called *categories* in AADL terminology. The ten categories of components are organized into three groups: Application software components are abstractions for functional aspects and data and include *thread*, *thread group*, *process*, *data*, and *subprogram*. Execution platform components describe hardware and include *processor*, *memory*, *device*, and *bus*. The *system* is the only member of the last group called composite components.

A general notion of connectors as services in the sense of CALM is absent in AADL. Instead, exchange and control of data is modeled through a set of primitives for various communication mechanisms, while resource-use is captured by the interrelations between application software components and execution platform components within a system. The primitives for data and control exchange are *message passing*, *event passing*, and *synchronized access to shared* (resource-) *components*, further there are *thread scheduling protocols*, *timing requirements*, and *remote procedure calls*. Being restricted to these fixed choices, connectors cannot be considered first-class modeling entities in AADL.

The term “interface” in AADL is used for collections of “ports”, which in turn are again drawn from a fixed set of primitives. Port primitives exist for five different directed flows of either data or control (*unqueued state data*, *queued message data*, *asynchronous events*, *synchronous subprogram calls*, and *explicit access to data components*). Hence, compared to the declarative interface specification of CALM, AADL follows a synthetic approach. Similarly, the semantics of an interface specification in AADL are a function of the fixed semantics of the ports composing the interface.

8.2.2 Dynamic behavior

Components in AADL feature operational modes and mode transition to model dynamic behavior in a way akin to the component property specification (CPS) in Cadena 1 [15]. Unlike the CPS, the modes and mode transitions of AADL are not only used to describe dynamic internal control states of the components, but also capture run-time reconfigurations of the interconnections, a feature which the fixed architecture-model of Cadena 1 (*i. e.*, CCM) did not allow as an explicit option. CALM on the other hand focuses on structural modeling and relies on model-interpreters for any operational semantics, hence a dynamic state-transition specification like in AADL is absent.

Additional support for analysis of the dynamic behavior of AADL models springs from the explicit mapping of behavioral and data components to specific execution platforms, whose real-time capabilities are part of the fixed semantics of the respective execution platform components.

8.2.3 Extensibility

Similar to CALM, the AADL core-language only describes architecture topologies. In AADL, *annexes* are used to introduce further specification. Annexes are distinguished according to whether they are *official* annexes (*annex documents*) which are approved by an SAE committee, or local *annex libraries*. Annex subclauses can be used within component type and implementation declarations if the annex library containing them is included. Existing annex documents include, for example, behavioral specification (*sec.* 8.2.2), programming language bindings, and real-time property specifications.

Annexes are comparable to the Cadena plugin mechanisms (*i. e.*, they do not describe new architectural elements but instead add interpretation and specification to existing ones, namely to components). Nevertheless, they are essential to the real-time, behavioral, and QoS analysis offered by the AADL framework, and therefore an integral part of the discussion in AADL. Consequently, the most important annexes are very powerful and go through an extensive approval process to be standardized in annex documents by the SAE.

While annexes extend the AADL core language, they have only limited power for introducing new architectural abstractions (see *sec.* 6.3). Nevertheless, the concrete specification they can add to modeling elements in effect adds new new or more specialized notions of components to the concepts of AADL, a feature which is extremely important for the ambition of AADL to provide flexible real-time architecture analysis.

8.2.4 Typing

Similar to CALM, AADL defines component types within a kind (category) by specifying the component type’s interfaces (ports). Also, a component type can be instantiated multiple times within a given assembly. Since the interfaces are constructed as a synthesis of primitives, there is no explicit typing of them. Nevertheless the options of attaching connectors to the interfaces are intrinsically determined by the fixed semantics of each primitive. Further, the synthetic interfaces determine a directionality of control and data-flow which (although not explicitly mentioned) necessarily also enforces at least semantically some sort of parity on the resulting ports, but it is unclear whether this parity is structurally enforced.

Next to the type, a component is determined by its implementation and visible attributes. The implementation can be another assembly within AADL with intricate rules governing how components can be nested based on their category. These rules are set because the concept of the implementation in AADL not only serves to achieve a hierarchical, nested, organization of assemblies, but also as the mechanism to unambiguously associate application software components with the execution platform components they use as resources.[1] This strategy is the basis of the real-time analysis capabilities of AADL.

[1]To emulate this concept in CALM, multiple styles are needed, one for each different group of component categories. The coercion/conversion constraints in CALM then have to be used to implement the nesting rules of AADL.

Judging from its graphical syntax, AADL would relatively easily allow to associate multiple implementations to the same component type, comparable to the concept of the implementation table in CALM. Nevertheless this does not seem to be regarded as a key feature by the developers of AADL, at least there is no explicit support for such libraries.

The component types allow a structural subtyping akin to the classes in object oriented programming languages (*i. e.*, a component subtype contains all the defining elements of its ancestor and can add new elements). Since on one hand there is no mention of minimum or maximum fan-in/fan-out constraints comparable to CALM's multiplexity, and on the other hand there is an implicit parity on the synthetic, directional, interfaces, it is unclear how this subtyping strategy addresses the problems of structural subtyping in a component oriented setting (see *sec.* 4.3.4). Other notions of subtyping (*e. g.*, weak or strong behavioral subtyping), or the use of typing to directly enforce connectivity constraints are not mentioned in the documentation and do not seem to be an emphasis in AADL.

8.3 Architectural Styles

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.

Abowd, Allen, and Garlan [1, 2] first 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 (see also Garlan and Shaw, [61]). Nevertheless, existing ADLs vary in their ability to specify and enforce styles. Di Nitto and Rosenblum investigate ADL suitability for modeling component systems, noting the need for support of architectural styles and style refinement [16]. Of ADLs evaluated they found only Acme/Armani [26, 51] satisfactorily supporting style refinement for modeling middleware compliant software through Acme *family* extensions.

Nevertheless, there are important distinctions and tradeoffs between CALM styles and Acme's approach to representing architectural styles via families. 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 or kinding mechanism 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)." [38] 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 constraint language. While Acme does support the typing of component instances (*i. e.*, instances conform to component types), even basic typing capabilities relevant for constraining assemblies (such as the restriction of port interfaces to specific types or the requirement for type-correct port/interface/role connections) must be specified in first order logic.

Further, Acme does not provide strict separation of type declarations and the modeling of type instances. Organizing type declaration separate from type instantiation is allowed, but separation is neither required, nor enforced by the Acme language [27]. Acme type declarations are collected within the families, but types may be extended and created within Acme *systems*, resulting in a mixed model. Without constraints over type creation, model interpreters (even if associated with an Acme family) are expected to handle all types expressible in the language.

In contrast, CALM goes beyond the notion of families by introducing a separate modeling tier which captures the architectural style in a mechanically leverageable way, thereby defining precisely what can appear within a style and what cannot (*i. e.*, CALM styles appear at a meta-level above Acme families). 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. Further, the complexity of first order constraint languages arguably 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. Therefore, while first order constraint languages are necessary, 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 (*chp.* 6, 7.1). These capabilities are not supported in Acme and supporting them in any constraint framework based on first order logic seems difficult.

Like Acme, xADL 2.0 has been designed as *architecture exchange language*, but seeks greater robustness and the ability to adjust by using a flexible framework of XML schemas [13]. xADL provides basic notions of component and interface types but type checking, as well as constraint specification and enforcement mechanisms are not directly integrated and not intrinsic to the supported process of xADL, but rather rely on external tools (*e. g.*, OCL or Armani constraints).

xADL 2.0 does distinguish design-time types and run-time instances. Using xADL XSD schemas [65], all type definitions must reside within element *xArchTypes*, and instances of these types model a run-time system under element *xArchInstance* (CALM provides a similar separation of design-time types within the module tier, and run-time system description within scenario tier). 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. Also, using such an extension to bridge the gulf between platform specification and library types forces the elements of the specification and design-time types to be considered the same type category. This may be contrasted to CALM, where the platform definition happens on the meta-modeling level (style), and type libraries are grouped naturally through their conformance to the style specification (kinds). While there is value in a tool engineering approach that enables separate tools to provide constraint enforcement, the CALM/CADENA project pursues 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.

Both xADL 2.0 and CALM support product-line modeling. As laid out in Section 3.2.1, the CALM modeling tiers are designed to align with established product-line organizational models. To capture variabilities, xADL and CALM offer very different support. xADL 2.0 uses first class constructs to express variants and options of the run-time architectural model, a concept not present in current CALM/CADENA. CALM instead integrates the notion of architecture variabilities into the relations between separate styles and the idea of inter-style migration. Therefore, while xADL 2.0 captures multiple assemblies in one model (containing variabilities), thereby expressing a product family without having to strictly specify the platform, CALM models product lines from the perspective of their equal platform infrastructure, eliminating the need to summarize multiple assemblies in one model (*i. e.*, the approaches can be seen as dual to each other). Nevertheless, interaction with industrial collaborators during the development of CALM indicate that the main concern is with appropriately modeling and refining infrastructure models for product-line development as captured by styles and style refinement constraints, supporting a platform-oriented approach.

8.4 META-MODELING

The Generic Modeling Environment (GME) is a powerful framework supporting the graphic definition of domain-specific modeling languages and the capability to generate domain-specific graphic modeling environments [39, 37]. GME allows users to define a meta-model using a notation based on the UML class diagram and first order logic constraints written in OCL. Each GME meta-model specifies a *paradigm* (roughly corresponding to a style in CALM), which can be used to mechanically generate domain specific architecture modeling environments. Within the modeling environment derived from a paradigm, architectural models can be graphically defined (with entities defined in the paradigm) and stored either in an internal database or serialized into XML format. These models then serve as a base for analysis or code generation (similar as in core CALM), which in GME terminology is called *model-integrated program synthesis* (MIPS). The modeling environment of a paradigm is consequently termed the *MIPS environment*, whereas the set of models defined within the same MIPS environment (and therefore adhering to the same paradigm) is called a *family* (which is conceptually close to a product line). GME offers some support for composing paradigms from elements defined in existing paradigm definitions [40]. Nevertheless, in contrast to CALM, GME focusses on refining separate modeling elements, mechanisms to relate paradigms to describe concepts corresponding to CALM's style refinement (*chp.* 6) are lacking in GME.

Similar to CALM/CADENA, GME focusses on entity-interrelations. A dynamic semantics has to be added to the GME models through a model interpretation process.

An example of a GME paradigm is the Component Synthesis using Model-Integrated Computing (CoSMIC) project which defines a graphic architecture language called Platform-Independent Component Modeling Language (PICML) [30]. CoSMIC captures platform component, connector, and interface templates, but such fundamental notions of type-checking as type correct connections (*i. e.*, port/interface/role connections based on correct typing) must be accomplished through OCL constraints. Further, among other projects, GME has been used to generate a graphic modeling tool for OMG's MOF or a development environment for nesC called *Gratis* (www.isis.vanderbilt.edu/projects/nest/gratis/).

8.5 OTHER FRAMEWORKS

8.5.1 ARCHITECTURES AND TYPING

In [47], Medvidovic, Rosenblum, and Taylor present a type theory for architectures based on notions from programming languages (specifically object oriented languages). Focus is the development of a subtyping relationship for architecture elements 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 (see *sec.* 7.1). 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 (compare *sec.* 4.3.4). 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.

8.5.2 ARCHITECTURE DESCRIPTION LANGUAGES

Multiple ADLs have been developed either for specific platforms (*e. g.*, the interface description language IDL for CCM) or general purpose for academia and industry. While CALM focusses on interrelation models and nominally typed structures, other ADLs and meta modeling frameworks emphasize functional aspects directly (as opposed to handling these through a plugin/model-interpretation mechanisms as in CALM). For example, various ADLs provide tool support for analysis, for implementation generation, or they directly capture dynamic reconfiguration of assemblies.

In [49], Medvidovic and Taylor provide an extensive overview over existing approaches to software architectures, compare methods and clarify the terminology. Their work moreover identifies the main modeling concepts and compares their realization in existing approaches. Among the ADLs they describe are Acme as an architecture exchange language (see *sec.* 8.3), Aesop [24, 25] which, like Acme, features a notion of architectural styles, and SADL [52] which is designed for formal refinement of architectures with increasing level of detail.

As more specific ADLs (*i. e.*, rather for specific architectural styles rather then for exchange) they present the following. C2 [46, 48] emphasizes notions from object-oriented type systems to describe interesting concepts of component type refinement (*sec.* 8.5.1). C2 supports a particular class of architectures (layered message passing systems, also described as highly distributed systems) and confines type descriptions to a modeling tier analogous to CALM's module tier. Darwin [43, 44] also focusses on highly distributed systems and features a basic dynamism within the described architectures. MetaH [4] is designed for architectures within the guidance, navigation and control (GN&C) domain, featuring a specific programming language for algorithms in GN&C. Rapide [42, 41] addresses the dynamic behavior and dynamic reconfiguration of a described architecture. Finally, UniCon [60] synthesizes glue-code (*i. e.*, communication and run-time infrastructure implementation form components) in C.

CALM/CADENA seeks 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 weaved into CADENA (*e. g.*, through plugins). Behavioral descriptions [3] and dynamic reconfiguration mechanisms [44] 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) could also be incorporated. Nevertheless, CALM focusses on enabling variation through providing a well structured environment in which alternatives can be constrained rather than through enumerating some alternatives by explicit encoding. The rationale behind CALM is that the strong typing capabilities are the key 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. Similarly, behavioral aspects or dynamism in CALM have to be provided as extensions/interpretations based on the well structured nominal interrelation systems defined within CALM.

8.6 SUMMARY

In previous work, important constraint specification and enforcement mechanisms are often not directly integrated into an ADL and not intrinsic to the supported process. While existing ADLs sometimes feature very expressive, complex constraint languages based on first-order logic like Armani and OCL as means of complementing specification, many forms of architecture constraints, basic notions of compatibility between components, or style/platform constraints, can be captured through much simpler notions of type schemas and type checking that are much more familiar to both architects and developers. The complexity of first-order constraint languages arguably limits accessibility to engineers, makes constraints more difficult specify, maintain, and evolve in the context of large-scale systems, and negatively impacts the scalability of checking. Therefore, although first order logic is more expressive than the (nominal, massively interrelated) typing of CALM, it should only be used to complement the more intuitive integration of constraints into the very structure of the modeling language.

Often, ADLs go beyond the specification of interrelation models and include various orthogonal concepts such as behavior, dynamism, code generation, analysis, *etc.*. CALM and CADENA are designed to support such extensions on a case-to-case basis using plugins. This way, CALM can encompass and benefit from previous solutions, potentially combining different orthogonal aspects.

While previous frameworks offer many innovative ideas which inspired various aspects and concepts of CALM, they seem insufficient 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 architectural styles rather than simply specifying a closed enumeration of available types, (b) incorporate multiple architectural styles within a single system (as 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 architectural styles.

CHAPTER 9

CONCLUSIONS

"I have had my results for a long time: but I do not yet know how I am to arrive at them."

— Carl Friedrich Gauß. in A. Arber. "The Mind and the Eye" (1954)

While CALM and CADENA have been outlined in various previous publications, this thesis gives the first in-depth, comprehensive, presentation of the CALM core syntax and interrelation semantics (*chp.* 4). Further, (with less rigor) it describes in detail the extensions which broaden the view from intra-style interrelations towards complex, multi-style, models which express refinement, abstraction, specialization, and other relations of architectural styles and artefacts relevant for model driven development (*chp.* 6, 7).

By introducing a style tier able to model component frameworks in a uniform way, and by integrating the artefacts of the style tier (*i. e.*, styles) into the overall modeling concept to be manipulable, first-class, entities, CALM allows to capture interdependencies of architectural styles on differing levels of abstraction or precision. CALM therefore moves beyond merely capturing existing, widely used, component frameworks or abstract, conceptual component models or even highly specific, domain-tailored modular styles and architectures realized within any of them. Instead, the whole development from abstract relations of planned entities towards concrete architectures of fixed building blocks with completed implementations can be captured in CALM. This allows to re-visit and change abstractions and re-factor their concrete manifestations at any stage of the development process, preventing decay of the architectural integrity.

As a side effect of the uniform capturing of architectural styles, conceptual architectures which are not grounded in a concrete middleware setting and previously would have been exchanged in form of informal graphs with attached natural language descriptions (*e. g.*, as PowerPoint slides) receive the same status in CALM as architectures which are anchored in available platform implementations. Such conceptual styles therefore become formally well defined entities which in CALM can be connected with more concrete artefacts until a level is reached which satisfiably allows code development or even mechanic code generation.

With CADENA, CALM has an experimentation platform to apply the otherwise complex notions in a convenient way to practical development tasks. CADENA itself is highly modular and allows substantial extensions, such as code generation, analysis, and verification tools.

9.1 SHORTCOMINGS OF CALM

Even though CALM was evaluated against many common, middleware-based, component frameworks, various details of CALM are not "hardened" enough by experience. Also, few elements of the core calculus seem verbose and could maybe be tightened without loss of expressiveness or generality.

Moreover, the already introduced notions of CALM and the experiments with modeling systems using these notions uncover some aspects of the basic modeling elements which open doors to a diverse set of questions, insights, and directions for future work.

9.1.1 The attribute system

Currently, the category of CALM artefacts is centered around the trinity of attributes, ports, and roles. An artefact with attributes only can be typed as interface, attributes and ports are defining for a component, attributes and roles classify a connector, while all three can appear in an assembly.

Even under a shallow assessment, attributes do not fit elegantly enough into this pattern. Also, most architectures developed for demonstration and experimentation do not make use of attributes at all, as all interfaces/component/connector kinds, types, and instances are nominally identifiable, and their interrelations are independent from attributes anyways. Consequently, factoring out attributes into separate attribute sheets is a feature which had been introduced early into the CALM development and the Cadena tool-set. These attribute sheets (shortly mentioned in *sec.* 6.2.5) can be added to a style to create an unnamed, ad-hoc, refined style with more precision (a refinement step independent from kind elision/addition based style refinement). Finally, along through the inception of core CALM the decision was taken to separate out development of the attribute type system from the rest of CALM.

All these considerations suggest that a CALM core completely without attributes could provide a more focused, less cluttered, formal capture of a system's entities' interrelations. A separate "attribute style" can still be used to complement interrelation models with meta-data. Note that one strength of CALM is that every modeling element is uniquely identifiable and easy to reference, which greatly simplifies linking information to it from outside of CALM.

With the attributes factored out, the duality of ports and roles is left to distinguish four different classes of modeling elements (interface, component, connector, assembly) through their presence or absence, which seems more justifiable. It should be mentioned that the importance of attributes in a model depends mostly on the presence of extensions available for that model such as code generation (which depends on attributes to provide meta-data) or static analysis (which needs attributes for operational semantics).

On the other hand, a qualitatively improved attribute system might integrate other entities which stand aside of the CALM core, such as the implementation table, including coercions and conversions. Finally, it might serve to add operational semantics to the interrelation skeleton provided by core CALM.

9.1.2 Interfaces and parities

The parities `provides` and `uses` on ports and roles have been subject of much discussion. For abstract, conceptual, styles they seem unnecessary or even inhibiting a certain level of generality (the problem is discussed from a different perspective in *sec.* 3.3.3). For concrete styles on the other hand, two distinguishable sides of an interface, which models an interaction point and therefore links with at least two agents, seem absolutely necessary.

An important direction of future research would be the concept of a variable number of parities defined per interface (meta-) kind. Background to this idea (which has not been published in this form as of yet) is the observation that, specifically since an interface might represent an interaction point with a complex protocol, any number of agents might participate in different roles in the interaction. The task of introducing this idea into CALM in non-trivial because the parities of current CALM, beyond distinguishing actors in an interaction, also take part in the instantiation of interfaces in an assembly (*i. e.*, an interface is currently instantiated together with the element which provides it).

9.1.3 Flexible component types

Often, architectural styles allow component types with a variable number of ports or at least with specific ports which can be replicated. An example is again the nesC timer component (*e. g.*, *sec.* 2.2, 5.1.5, 7.1.4) whose timer-port can be replicated with each individual copy carrying a numeric identifier (*apx.* C.1.3, *lst.* 5.18, *l.* 52: the `uint8_t`-typed variable `id` holds the identifier)

```
provides interface Timer[uint8_t id];
```

Port options (or role options) in CALM do not offer the possibility to define replicable ports. Moreover, CALM type-level component artefacts are declared with the number of ports as part of the type. The current workaround to capture replicable ports in CALM and CADENA is to create types "on-the-fly" with tool support, nevertheless the distinction between the type of a timer instance with for example three timer ports *vs.* four timer ports persists.

Also, CALM could potentially benefit from the introduction of *parametric* types where not only the number of ports but also the valuation of some of the attributes on a component could be supplied as parameters to a component type-template. Again, currently such constructs can be approximated in CADENA through tool support. Future research has to decide which of these features (flexible port numbers, parametric types, *etc.*) adds a fundamental or qualitative benefit to the modeling concepts of CALM.

9.1.4 Standard operational semantics

CALM aims to provide a comprehensive interrelation semantics capturing architectures within component oriented frameworks. An obvious step for extension is to add uniform operational specifications to CALM to enable static analysis of system behavior. An interesting approach to accomplish uniform operational specification in CALM (besides using the attribute system, see *sec.* 9.1.1) would be a CALM style modeling some automaton-based input/output process description (Mealy/Moore finite state machines, petri nets) with components being nodes and connectors being edges, carrying an intrinsic (and well defined) operational semantics. Assemblies in that style can then be connected to modeled component architectures in the same way that inter-style sub-assemblies are handled now (*sec.* 7.2.2).

Additional benefit of this approach would be that the abstraction-implementation relation between styles which in current CALM is based on declaration (*i. e.*, *coercion* and *conversion*, *sec.* 7.2.2), can then be grounded on the relations of every style to the common operational sub-style. To illustrate, consider a style A whose elements can (due to respective coercions) be implemented in terms of two different styles B_1 and B_2.[1]

(a)
$$\begin{tikzcd} & A & \\ B_1 \arrow[ur] & & B_2 \arrow[ul] \\ \vdots & & \vdots \end{tikzcd}$$

(b)
$$\begin{tikzcd} & A & \\ B_1 \arrow[ur] & & B_2 \arrow[ul] \\ & C \arrow[ul] \arrow[ur] & \end{tikzcd}$$

In the current, declarative approach (a), the relation between B_1 and B_2 is not clear, nevertheless assuming a uniform operational style C (b) the inter-style relations can be formally scrutinized.

9.1.5 Subtyping on module-level artefacts

Section 4.3.4 discusses the fact that the common notions of subtyping which are drawn from object oriented programming do not transfer easily into the world of component oriented development.

Because object oriented programming is the most widely used paradigm in current practice, most working developers nowadays are familiar with the concepts of *inheritance*, where a subtype extends or overwrites functionality inherited from a supertype in a way that subsequently instances of the subtype can be used in lieu of instances of the supertype. Although the component oriented paradigm offers a different approach to module—environment relations (*i. e.*, the exchangeability is based on stated context dependencies as opposed to declarative sub-typing), some component oriented frameworks (*e. g.*, CCM) offer the possibility to create component types by inheriting from existing types, a functionality which working programmers often would expect out of habit despite the differences between the object oriented and the component oriented paradigm. CALM allows to extend component types as a convenient way to create new types (see *sec.* 4.3.2), but for reasons mentioned above (*sec.* 4.3.4) does not consider the new types to be related to the existing type.

Future research would have to find a way to capture covariance and contravariance properties of ports to turn this structural extension mechanism into real subtyping.

[1] It is irrelevant for this illustration whether the styles B_1 and B_2 are mutually exclusive alternatives or options that can be used simultaneously on different elements of the same assembly.

9.2 Cadena development

9.2.1 Synchronizing of CALM and Cadena

The first version of Cadena has been developed before the start of the CALM project, while the development of Cadena 2 happens simultaneously with the work on CALM. Naturally, some concepts developed inside CALM did not find their way into the complex Cadena tool-set yet, some deprecated ideas of CALM which in CALM itself have been revised or even abandoned are still manifested in Cadena, and finally some Cadena original concepts are not formalized thoroughly in CALM.

Extensive work needs to be done on Cadena to more faithfully reflect CALM concepts. Nevertheless, the aim with this work is not to turn Cadena into a one-to-one manifestation of CALM, since a tool offers different opportunities than a calculus. Instead the current system of mutual inspiration between theory and tool can persist, if most core concepts of CALM find their way into the implementation.

Object oriented programming notions

The first drafts of CALM were based on analogies to object oriented concepts. The three categories would be expressed in terms of three basic meta-kinds (***`mComponent`***, ***`mConnector`***, and ***`mInterface`***), with every new meta-kind being derived directly or indirectly from one of these three using one-inheritance (*i. e.*, every meta-kind would have exactly one ancestor). The category of a meta-kind would not be given by a keyword (in current CALM "**`metacomponent`**", *etc.*) but rather through the root of the inheritance tree in which they are created. For the three basic meta-kinds to be available, every style would have to directly or indirectly inherit from a core style.

This system was designed to ensure that categories can easily be added to the concept (by adding a new base meta-kind). Nevertheless practical realization in CALM as well as (independently) in Cadena proved to be overly complicated, so that a clear distinction of the categories based on keywords was introduced in CALM. Cadena still features the core style and the three base meta-kinds ***`mComponent`***, ***`mConnector`***, but only on the surface. Data structures holding meta-kinds instead are tailored to individual categories.

Therefore, one important improvement for Cadena would be to eliminate the core style and the three base meta-kinds from the input wizards and dialogs.

Model migration facilities

Another issue where Cadena lags behind CALM is due to the concept of migration. Data structures of CALM module and scenario artefacts hard-link to the style in which they are created. It is therefore extremely difficult to transfer architectural models from one style to the next within Cadena, even if compliance is already established (*i. e.*, no alterations to the models would be necessary).

Extensive tool support for migration is crucial for the concept to be accepted. Therefore, after adapting the main data structures to be more independent from the style they were created in (and instead reference more clearly individual kinds), various user interaction and feedback features can be added to Cadena to support migration in a convenient way.

Implementation tables

The last major block of improvements needed for Cadena is a more thorough support for nesting and flexible attachment of implementation information to the elements of an architectural assembly (Chapter 7). Currently, Cadena only supports the integration of existing assemblies into new ones, linking assemblies to declared types is still under construction. These features will be crucial for Cadena to move beyond homogeneous architectural styles and development support through simple code generation. With these planned capabilities Cadena will not only be able to capture a component framework and to support (end-to-end) development therein but also to enhance the modeled frameworks capabilities, introduce useful abstractions which the modeled framework itself cannot offer, and connect the models with other domain expertise on different levels of abstraction.

9.2.2 Engineering

While some of the core data-structures of CADENA need to be revisited (see, *e. g.*, *sec.* 9.2.1), the overall design of CADENA is extremely robust and stable. Being a tool for software engineering, CADENA's own engineering problems are worth looking at.

Idiosyncrasies of CALM and EMF

As described, the representation of the massively interconnected CALM models in CADENA relies on the automatic referencing service of EMF (*sec.* 4.5.2). The reference chains of artefacts within CADENA can be as complex as in CALM itself.

To allow efficient work with CADENA most of the hidden functionality of update propagation is forked into separate concurrent threads. The programming of CADENA is very robust with respect to the end-results, which means that once the threads converge, the model will be consistent. Unfortunately the use of concurrency gives rise to various race conditions which EMF itself does not always handle. This leads to thrown exceptions in CADENA. An example of such an error (which, due to the underlying race conditions, cannot always be reproduced) occurs when a complex hybrid style is loaded and immediately after some model elements (modules, scenarios) in that style are opened. The first opening of the model elements might lead to a long list of exceptions (which CADENA catches and displays), if the same elements are closed and subsequently opened, they do not yield any errors.

While from the view of CADENA this is a technical problem which can be fixed with defensive programming, it also raises research questions about the impact of timing and thread dependencies in interrelation models.

Modularity and exchange

To enable industrial size development on the base of CADENA, CADENA artefacts have to be exchangeable between groups of developers through various mechanisms (*e. g.*, version/revision control systems such as CVS, PRCS, or subversion, web repositories, or simply e-mail attachments). The problem is, that what is captured by CALM is only a part of the data which CADENA manipulates. Next to the core concepts (which are inside of CALM), meta information such as form data (opened or closed input forms), visualizations (interface icons, component/connector visual styles), or layout information (position of components in graphical view, routing of connectors, size, hidden/visible parts, *etc.*) is also crucial for a working communication inside and in between different development teams.

CADENA has to solve the problem that on the one hand, the essential parts of a model (the actual CALM model) has to be recognizable and uncluttered by visual meta-data, while on the other hand, visual meta-data needs to be available to accompany the core model in a convenient way to enable conversation between developers. Considering that the core models (containing styles, modules, assemblies, coercion and conversion data, attribute sheets, implementation tables, *etc.*) are already complex sets of data, separating out visual and development state meta-information is non trivial.

9.3 Summary

To eventually arrive at a general, widely applicable, theory of interrelation models of modular systems, the concepts of CALM have to be constantly questioned and re-visited with continuing careful evaluation against real-life component-based development. Nevertheless, just generalizing from currently used industry standards and procedures is not likely to add fundamental improvements to CALM, instead, new concepts which are invented inside the realm of CALM's theoretical background need to be carried into the development practice to prove their applicability in and improve working development methods and abstractions. It therefore seems important to maintain the current duality between CALM and CADENA to work on both, concept and practical application.

BIBLIOGRAPHY

[1] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 9–20. ACM Press, 1993. 8.3

[2] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):319–364, 1995. 8.3

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997. 3.1.3, 8.5.2

[4] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, 1996. 8.2, 8.5.2

[5] Bogor software model checking framework. http://bogor.projects.cis.ksu.edu. 1.2.1

[6] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003. 4.5.2

[7] CADENA web site. http://cadena.projects.cis.ksu.edu/. 1.1, 4.5

[8] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. Calm and Cadena: Metamodeling for component-based product-line development. *IEEE Computer*, 39(2):42–50, February 2006. 1.1, 1.3, 4.5

[9] A. Childs, J. Greenwald, V. P. Ranganath, X. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, P. S. Kumar, and G. Singh. Cadena: An integrated development environment for analysis, synthesis, and verification of component-based systems. In M. Wermelinger and T. Margaria, editors, *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 160–164. Springer, 2004. 1.2.1

[10] CIAO – component-integrated ACE ORB. http://www.cse.wustl.edu/~nanbor/projects/CIAO/. 3.2.1

[11] P. Clements and L. Northrop. *Software Product Lines*. Addison Wesley, 2002. 1.1, 3.2.1

[12] DARPA demonstrates network centric technologies. Press-release. http://www.darpa.gov/body/news/2005/pces_demo.pdf, April 2005. 1.1, 5

[13] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 266–276. ACM Press, 2002. 1.1, 8.3

[14] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes In Computer Science*, pages 261–276. Springer, September 1999. 1.3

[15] X. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, and Robby. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 2852/2003 of *Lecture Notes in Computer Science*, pages 154–181. Springer, November 2002. 1.2.1, 8.2.2

[16] E. Di Nitto and D. S. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 13–22. IEEE Computer Society Press, 1999. 8.3

[17] M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the 3rd International Conference on Embedded Software*, volume 2855/2003 of *Lecture Notes in Computer Science*, pages 173–189. Springer, October 2003. 1.2.1

[18] Enterprise java CORBA component model. http://www.cpi.com/ejccm/. 3.2.1

[19] Eclipse modeling framework project (EMF). http://www.eclipse.org/modeling/emf/. 1.3, 4.5.2

[20] P. H. Feiler. Modeling of system families. Technical Report CMU/SEI-2007-TN-047, Software Engineering Institute, Carnegie Mellon University, 2007. 8.2

[21] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis &amp; design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2004. 8.2

[22] P. H. Feiler, B. Lewis, S. Vestal, and E. Colbert. An overview of the SAE architecture analysis and design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *Workshop on Architecture Description Languages (WADL)*, volume 176 of *IFIP International Federation for Information Processing*, pages 3–15. Springer, 2005. 8.2

[23] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003. 8.1

[24] D. Garlan. An introduction to the AESOP system. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps, July 1995. 8.5.2

[25] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of Software Engineering (FSE)*, pages 175–188. ACM Press, 1994. 8.5.2

[26] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 1997. 1.1, 8.3

[27] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of component-based systems*, pages 47–67. Cambridge University Press, 2000. 8.3

[28] D. Gay, P. Levis, D. Culler, and E. Brewer. *nesC 1.1 Language Reference Manual*. http://nescc.sourceforge.net/, May 2003. 1.1, 2.2, 5.1.1

[29] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 1–11. ACM Press, 2003. 1.1, 2.2, 5.1.1

[30] A. Gokhale, K. Balasubramanian, and T. Lu. CoSMIC: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems. In *Companion to the 19th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 218–219. ACM Press, 2004. 8.4

[31] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, volume 841, pages 160–173. IEEE Computer Society Press, May 2003. 1.2.1

[32] J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, V. P. Ranganath, and Robby. Slicing and partial evaluation of CORBA component model designs for avionics system. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 1–2. ACM Press, 2003. 1.2.1

[33] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. 3.1.3

[34] International Organization for Standardization. Information technology – open systems interconnection – basic reference model: The basic model. ISO/IEC 7498-1, 1994. 7.2.1

[35] G. Jung and J. Hatcliff. A type-centric framework for specifying heterogeneous, large-scale, component-oriented, architectures. To appear in Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE), October 2007. 1.3

[36] G. Jung, J. Hatcliff, and V. P. Ranganath. A correlation framework for the CORBA component model. In M. Wermelinger and T. Margaria, editors, *7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 144–159. Springer, 2004. 1.3

[37] G. Karsai, M. Maróti, Á. Lédeczi, J. Gray, and J. Sztipanovits. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, March 2004. 8.4

[38] A. Kompanek. Modeling a system with Acme. http://www.cs.cmu.edu/~acme. 8.3

[39] Á. Lédeczi, M. Maróti, A. Bakay, G. Karsai, J. Garrett, C. Y. Thomason, G. G. Nordstrom, and P. Völgyesi. The generic modeling environment. In *Proceedings of the Workshop on Intelligent Signal Processing*, May 2001. 8.4

[40] Á. Lédeczi, G. Nordstrom, G. Karsai, P. Völgyesi, and M. Maróti. On metamodel composition. In *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA)*, pages 756–760, 2001. 8.4

[41] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. 8.5.2

[42] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995. 8.5.2

[43] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC)*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer, September 1995. 8.5.2

[44] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14. ACM Press, 1996. 8.5.2

[45] V. Matena, S. Krishnan, L. DeMichiel, and B. Stearns. *Applying Enterprise JavaBeans*. Addison Wesley, 2003. 1.1, 2.2

[46] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32. ACM Press, 1996. 8.5.2

[47] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A type theory for software architectures. Technical Report UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, February 1998. 8.5.1

[48] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 44–35. IEEE Computer Society Press, 1999. 8.5.2

[49] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. 8.5.2

[50] The MICO CORBA component project. http://www.fpx.de/MicoCCM/. 3.2.1

[51] R. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, CMU School of Computer Science, September 2000. 1.1, 8.3

[52] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995. 8.5.2

[53] Object Management Group. MDA guide version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf. 6.1, 6.1(a), 8.1

[54] Object Management Group. *OMG formal/06-04-01 (CORBA Component Model Specification, v4.0)*, April 2006. 1.1, 1.2.1, 2.2

[55] OpenCCM – the open CORBA component model platform. http://openccm.objectweb.org/. 3.2.1

[56] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 3.2.2

[57] D. A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing. MIT Press, 1994. 3.2.2

[58] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In D. A. Lamb, editor, *Selected papers from the Workshop on Studies of Software Design*, volume 1078 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 1993. 3.1.3

[59] M. Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):1–7, October 2004.

[60] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS)*, pages 2–10. IEEE Computer Society, 1996. 8.5.2

[61] M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996. 8.3

[62] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press / Addison-Wesley, 2nd edition, 2002. 3.1.1, 3.1, 3.1.3

[63] UML notation guide (version 1.1). ftp://ftp.omg.org/pub/docs/ad/97-08-05.pdf, September 1997. 8.1

[64] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison Wesley, 2nd edition, August 2003. 1.1, 8.1

[65] xADL 2.0 schemas. http://www.isr.uci.edu/projects/xarchuci/ext-overview.html. 8.3

APPENDIX A

GRAMMARS

The syntax of CALM is given in form of sets s of syntactic entities which are inductively defined as the smallest sets generated by rule templates of the form

$$\frac{v_1 \in s_1, \quad \ldots, \quad v_n \in s_n}{t \in s},$$

where $v_1, \ldots, v_n$ are variables for terms in the already defined sets $s_1, \ldots, s_n$, and t is a term which contains $v_1, \ldots, v_n$. By the rule template above, for any set of terms $t_1, \ldots, t_n$ we have that

$$t_1 \in s_1, \ldots t_n \in s_n \quad \Rightarrow \quad t[t_1/v_1, \ldots, t_n/v_n] \in s.$$

The syntactic sets $s, s_1, \ldots, s_n$ can be seen as non-terminals of a Bakus-Naur-Form grammar (BNF), since each rule template

$$\frac{v_1 \in s_1, \quad \ldots, \quad v_n \in s_n}{t \in s},$$

is equivalent to a BNF grammar production

$$s ::= t[s_1/v_1, \ldots, s_n/v_n].$$

Thus, a complete BNF grammar for CALM can be obtained by mechanically translating the rules into productions.

CALM assumes a given set Identifier of alphanumeric identifiers id which serve as names, style-specific keywords, and lookup references. Further, two sets of syntactic entities are used without further definition on each of the three levels. These two are Type-Spec and Literal. Both are tied to the attribute type system of CALM which is part of a separate effort.

A.1 STYLE TIER

$$\frac{id_0 \in \text{Identifier} \quad i \in \text{Import-Spec}, \quad b \in \text{Style-Body}}{\texttt{style}\ id_0\ i\ \{\ b\ \} \in \text{Style}} \tag{A.1}$$

$$\frac{}{\epsilon \in \text{Import-Spec}} \tag{A.2}$$

$$\frac{t \in \text{Union}}{\texttt{include}\ t \in \text{Import-Spec}} \tag{A.3}$$

$$\frac{t \in \text{Intersection}}{t \in \text{Union}} \tag{A.4}$$

$$\frac{t_1 \in \text{Intersection}, \quad t_2 \in \text{Union}}{t_1 + t_2 \in \text{Union}} \tag{A.5}$$

$$\frac{t \in \text{Atom}}{t \in \text{Intersection}} \tag{A.6}$$

$$\frac{t_1 \in \text{Atom}, \quad t_2 \in \text{Intersection}}{t_1\ \hat{}\ t_2 \in \text{Intersection}} \tag{A.7}$$

$$\frac{id \in \textsf{Identifier}}{id \in \textsf{Atom}} \tag{A.8}$$

$$\frac{t \in \textsf{Union}}{\texttt{(}\ t\ \texttt{)} \in \textsf{Atom}} \tag{A.9}$$

$$\frac{}{\epsilon \in \textsf{Style-Body}} \tag{A.10}$$

$$\frac{t_1, t_2 \in \textsf{Style-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Style-Body}} \tag{A.11}$$

$$\frac{t \in \textsf{Meta-Kind-Spec}}{t \in \textsf{Style-Body}} \tag{A.12}$$

$$\frac{t \in \textsf{Kind-Definition}}{t \in \textsf{Style-Body}} \tag{A.13}$$

$$\frac{t \in \textsf{Type-Var-Decl}}{t \in \textsf{Style-Body}} \tag{A.14}$$

$$\frac{t \in \textsf{Type-Var-Assert}}{t \in \textsf{Style-Body}} \tag{A.15}$$

$$\frac{t \in \textsf{Attribute-Type-Spec}}{t \in \textsf{Style-Body}} \tag{A.16}$$

$$\frac{t \in \textsf{Constant}}{t \in \textsf{Style-Body}} \tag{A.17}$$

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Extends-list}, \quad b \in \textsf{Interface-MK-Body}}{\texttt{\textbf{metainterface}}\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Meta-Kind-Spec}} \tag{A.18}$$

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Extends-list}, \quad b \in \textsf{Connector-MK-Body}}{\texttt{\textbf{metaconnector}}\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Meta-Kind-Spec}} \tag{A.19}$$

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Extends-list}, \quad b \in \textsf{Component-MK-Body}}{\texttt{\textbf{metacomponent}}\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Meta-Kind-Spec}} \tag{A.20}$$

$$\frac{}{\epsilon \in \textsf{Extends-list}} \tag{A.21}$$

$$\frac{i \in \textsf{Identifier-list}}{\texttt{\textbf{extends}}\ i \in \textsf{Extends-list}} \tag{A.22}$$

$$\frac{id \in \textsf{Identifier}}{id \in \textsf{Identifier-list}} \tag{A.23}$$

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}}{id\texttt{,}\, i \in \textsf{Identifier-list}} \tag{A.24}$$

$$\frac{}{\epsilon \in \textsf{Interface-MK-Body}} \tag{A.25}$$

$$\frac{t_1, t_2 \in \textsf{Interface-MK-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Interface-MK-Body}} \tag{A.26}$$

$$\frac{a \in \textsf{Attribute}}{a \in \textsf{Interface-MK-Body}} \tag{A.27}$$

$$\frac{}{\epsilon \in \textsf{Connector-MK-Body}} \tag{A.28}$$

$$\frac{t_1, t_2 \in \textsf{Connector-MK-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Connector-MK-Body}} \tag{A.29}$$

$$\frac{a \in \textsf{Attribute}}{a \in \textsf{Connector-MK-Body}} \tag{A.30}$$

$$\frac{r \in \textsf{Role-Option}}{r \in \textsf{Connector-MK-Body}} \tag{A.31}$$

$$\frac{}{\epsilon \in \textsf{Component-MK-Body}} \tag{A.32}$$

$$\frac{t_1, t_2 \in \textsf{Component-MK-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Component-MK-Body}} \tag{A.33}$$

$$\frac{a \in \textsf{Attribute}}{a \in \textsf{Component-MK-Body}} \tag{A.34}$$

$$\frac{p \in \textsf{Port-Option}}{p \in \textsf{Component-MK-Body}} \tag{A.35}$$

$$\frac{id \in \textsf{Identifier}, \quad t \in \textsf{Type-Spec}}{\texttt{\textbf{attribute}}\ id\ \texttt{:}\ t \in \textsf{Attribute}} \tag{A.36}$$

$$\frac{t_1, t_2 \in \textsf{Range}, \quad i_1, i_2 \in \textsf{Identifier}}{\texttt{\textbf{uses}}\ t_1\ i_1\ \texttt{:}\ i_2\ t_2\texttt{;} \in \textsf{Role-Option}} \tag{A.37}$$

$$\frac{t_1, t_2 \in \textsf{Range}, \quad i_1, i_2 \in \textsf{Identifier}}{\texttt{\textbf{provides}}\ t_1\ i_1\ \texttt{:}\ i_2\ t_2\texttt{;} \in \textsf{Role-Option}} \tag{A.38}$$

$$\frac{t_1, t_2 \in \textsf{Range}, \quad i_1, i_2 \in \textsf{Identifier}}{\texttt{\textbf{uses}}\ t_1\ i_1\ \texttt{:}\ i_2\ t_2\texttt{;} \in \textsf{Port-Option}} \tag{A.39}$$

$$\frac{t_1, t_2 \in \textsf{Range}, \quad i_1, i_2 \in \textsf{Identifier}}{\texttt{\textbf{provides}}\ t_1\ i_1\ \texttt{:}\ i_2\ t_2\texttt{;} \in \textsf{Port-Option}} \tag{A.40}$$

$$\frac{n \in \mathbb{N}_0, \quad m \in \mathbb{N}_0^\star, \quad n \leq m}{\texttt{[}n\texttt{..}m\texttt{]} \in \textsf{Range}} \tag{A.41}$$

$$\frac{n \in \mathbb{N}_0}{\texttt{[}n\texttt{]} \in \textsf{Range}} \tag{A.42}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad b \in \textsf{Interface-Export-Spec}}{\texttt{interfacekind}\ id_1\ \texttt{:}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Kind-Definition}} \tag{A.43}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad b \in \textsf{Connector-Export-Spec}}{\texttt{connectorkind}\ id_1\ \texttt{:}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Kind-Definition}} \tag{A.44}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad b \in \textsf{Component-Export-Spec}}{\texttt{componentkind}\ id_1\ \texttt{:}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Kind-Definition}} \tag{A.45}$$

$$\frac{}{\epsilon \in \textsf{Interface-Export-Spec}} \tag{A.46}$$

$$\frac{t_1, t_2 \in \textsf{Interface-Export-Spec}}{t_1\texttt{;}\ t_2 \in \textsf{Interface-Export-Spec}} \tag{A.47}$$

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Interface-Export-Spec}} \tag{A.48}$$

$$\frac{}{\epsilon \in \textsf{Connector-Export-Spec}} \tag{A.49}$$

$$\frac{t_1, t_2 \in \textsf{Connector-Export-Spec}}{t_1\texttt{;}\ t_2 \in \textsf{Connector-Export-Spec}} \tag{A.50}$$

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Connector-Export-Spec}} \tag{A.51}$$

$$\frac{t \in \textsf{Export-Kind-Spec}}{t \in \textsf{Connector-Export-Spec}} \tag{A.52}$$

$$\frac{t \in \textsf{Type-Var-Decl}}{t \in \textsf{Connector-Export-Spec}} \tag{A.53}$$

$$\frac{t \in \textsf{Type-Var-Assert}}{t \in \textsf{Connector-Export-Spec}} \tag{A.54}$$

$$\frac{}{\epsilon \in \textsf{Component-Export-Spec}} \tag{A.55}$$

$$\frac{t_1, t_2 \in \textsf{Component-Export-Spec}}{t_1\texttt{;}\ t_2 \in \textsf{Component-Export-Spec}} \tag{A.56}$$

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Component-Export-Spec}} \tag{A.57}$$

$$\frac{t \in \textsf{Export-Kind-Spec}}{t \in \textsf{Component-Export-Spec}} \tag{A.58}$$

$$\frac{t \in \textsf{Type-Var-Spec}}{t \in \textsf{Component-Export-Spec}} \tag{A.59}$$

$$\frac{t \in \textsf{Type-Var-Assert}}{t \in \textsf{Component-Export-Spec}} \tag{A.60}$$

$$\frac{id \in \textsf{Identifier}, \quad v \in \textsf{Literal}}{id\ \texttt{=}\ v \in \textsf{Attribute-Valuation}} \tag{A.61}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}}{id_1\ \texttt{->}\ id_2 \in \textsf{Export-Kind-Spec}} \tag{A.62}$$

$$\frac{i \in \textsf{Identifier-list}, \quad id \in \textsf{Identifier}}{\texttt{typevar}\ i\ \texttt{:}\ id \in \textsf{Type-Var-Decl}} \tag{A.63}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad \sim\ \in \{\texttt{<=}, \texttt{=}, \texttt{>=}\}}{\texttt{assert}\ id_1\ \sim\ id_2 \in \textsf{Type-Var-Assert}} \tag{A.64}$$

$$\frac{id \in \textsf{Identifier}}{\texttt{typedecl}\ id \in \textsf{Attribute-Type-Spec}} \tag{A.65}$$

$$\frac{id \in \textsf{Identifier}, \quad t \in \textsf{Type-Spec}}{\texttt{typedef}\ id\ \texttt{=}\ t \in \textsf{Attribute-Type-Spec}} \tag{A.66}$$

$$\frac{id \in \textsf{Identifier}, \quad t \in \textsf{Type-Spec}, \quad v \in \textsf{Literal}}{id\ \texttt{:}\ t\ \texttt{=}\ v \in \textsf{Constant}} \tag{A.67}$$

A.2 Module tier

$$\frac{id_0, id_1 \in \textsf{Identifier}, \quad i \in \textsf{Input}, \quad b \in \textsf{Module-Body}}{\texttt{module}\ id_0\ \texttt{of}\ id_1\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Module}} \tag{A.68}$$

$$\frac{}{\epsilon \in \textsf{Input}} \tag{A.69}$$

$$\frac{i \in \textsf{Identifier-list}}{\texttt{input}\ i \in \textsf{Input}} \tag{A.70}$$

$$\frac{id \in \textsf{Identifier}}{id \in \textsf{Identifier-list}} \tag{A.71}$$

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}}{id\,\texttt{,}\,i \in \textsf{Identifier-list}} \tag{A.72}$$

$$\frac{}{\epsilon \in \textsf{Module-Body}} \tag{A.73}$$

$$\frac{t_1, t_2 \in \textsf{Module-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Module-Body}} \tag{A.74}$$

$$\frac{t \in \textsf{Type}}{t \in \textsf{Module-Body}} \tag{A.75}$$

$$\frac{t \in \textsf{Constant}}{t \in \textsf{Module-Body}} \tag{A.76}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad i \in \textsf{Include}, \quad b \in \textsf{Type-Body}}{id\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Type}} \tag{A.77}$$

$$\frac{}{\epsilon \in \textsf{Include}} \tag{A.78}$$

$$\frac{i \in \textsf{Identifier-list}}{\texttt{include}\ i \in \textsf{Include}} \tag{A.79}$$

$$\frac{}{\epsilon \in \textsf{Type-Body}} \tag{A.80}$$

$$\frac{t_1, t_2 \in \textsf{Type-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Type-Body}} \tag{A.81}$$

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Type-Body}} \tag{A.82}$$

$$\frac{t \in \textsf{Port}}{t \in \textsf{Type-Body}} \tag{A.83}$$

$$\frac{id \in \textsf{Identifier}, \quad v \in \textsf{Literal}}{id\ \texttt{=}\ v \in \textsf{Attribute-Valuation}} \tag{A.84}$$

$$\frac{id_1, id_2, id_3 \in \textsf{Identifier}}{id_1\ id_2\ \texttt{:}\ id_3 \in \textsf{Port}} \tag{A.85}$$

$$\frac{id \in \textsf{Identifier}, \quad t \in \textsf{Type-Spec}, \quad v \in \textsf{Literal}}{id\ \texttt{:}\ t\ \texttt{=}\ v \in \textsf{Constant}} \tag{A.86}$$

A.3 Scenario tier

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}, \quad b \in \textsf{Scenario-Body}}{\texttt{scenario}\ id\ \texttt{includes}\ i\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Scenario}} \tag{A.87}$$

$$\frac{id \in \textsf{Identifier}}{id \in \textsf{Identifier-list}} \tag{A.88}$$

$$\frac{id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}}{id\,\texttt{,}\,i \in \textsf{Identifier-list}} \tag{A.89}$$

$$\frac{}{\epsilon \in \textsf{Scenario-Body}} \tag{A.90}$$

$$\frac{t_1, t_2 \in \textsf{Scenario-Body}}{t_1\texttt{;}\ t_2 \in \textsf{Scenario-Body}} \tag{A.91}$$

$$\frac{t \in \textsf{Component}}{t \in \textsf{Scenario-Body}} \tag{A.92}$$

$$\frac{t \in \textsf{Connector}}{t \in \textsf{Scenario-Body}} \tag{A.93}$$

$$\frac{t \in \textsf{Constant}}{t \in \textsf{Scenario-Body}} \tag{A.94}$$

$$\frac{id_0, id_1 \in \textsf{Identifier}, \quad b \in \textsf{Component-Body}}{id_0\ id_1\ \texttt{\{}\ b\ \texttt{\}} \in \textsf{Component}} \tag{A.95}$$

$$\frac{}{\epsilon \in \textsf{Component-Body}} \tag{A.96}$$

$$\frac{t_1, t_2 \in \textsf{Component-Body}}{t_1\texttt{;}t_2 \in \textsf{Component-Body}} \tag{A.97}$$

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Component-Body}} \tag{A.98}$$

$$\frac{t \in \textsf{Interface}}{t \in \textsf{Component-Body}} \tag{A.99}$$

$$\frac{id \in \textsf{Identifier}, \quad v \in \textsf{Literal}}{id\ \texttt{=}\ v \in \textsf{Attribute-Valuation}} \tag{A.100}$$

$$\frac{id \in \textsf{Identifier}, \quad a \in \textsf{Attributes}}{id\ \texttt{\{}\ a\ \texttt{\}} \in \textsf{Interface}} \tag{A.101}$$

$$\frac{}{\epsilon \in \textsf{Attributes}} \tag{A.102}$$

$$\frac{t_1, t_2 \in \textsf{Attributes}}{t_1 \texttt{;} t_2 \in \textsf{Attributes}} \tag{A.103}$$

$$\frac{a \in \textsf{Attribute-Valuation}}{a \in \textsf{Attributes}} \tag{A.104}$$

$$\frac{id \in \textsf{Identifier}, \quad b \in \textsf{Connector-Body}}{id_1 \texttt{ \{ } b \texttt{ \} } \in \textsf{Connector}} \tag{A.105}$$

$$\frac{}{\epsilon \in \textsf{Connector-Body}} \tag{A.106}$$

$$\frac{t_1, t_2 \in \textsf{Connector-Body}}{t_1 \texttt{;} t_2 \in \textsf{Connector-Body}} \tag{A.107}$$

$$\frac{t \in \textsf{Attribute-Valuation}}{t \in \textsf{Connector-Body}} \tag{A.108}$$

$$\frac{t \in \textsf{Binding}}{t \in \textsf{Connector-Body}} \tag{A.109}$$

$$\frac{id \in \textsf{Identifier}, \quad l \in \textsf{Port-list}}{id \texttt{ = } l \in \textsf{Binding}} \tag{A.110}$$

$$\frac{id \in \textsf{Identifier}, \quad l \in \textsf{Port-list}, \quad a \in \textsf{Attributes}}{id \texttt{ \{ } a \texttt{ \} } \texttt{ = } l \in \textsf{Binding}} \tag{A.111}$$

$$\frac{p \in \textsf{Port}}{p \in \textsf{Port-list}} \tag{A.112}$$

$$\frac{p \in \textsf{Port}, \quad l \in \textsf{Port-list}}{p \texttt{,} l \in \textsf{Port-list}} \tag{A.113}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}}{id_1 \texttt{.} id_2 \in \textsf{Port}} \tag{A.114}$$

$$\frac{id \in \textsf{Identifier}, \quad t \in \textsf{Type-Spec}, \quad v \in \textsf{Literal}}{id \texttt{ : } t \texttt{ = } v \in \textsf{Constant}} \tag{A.115}$$

APPENDIX B

STRUCTURES

Each syntactic set s of the CALM syntax has an interpretation relation which, assuming some given environment of lookup mappings, terms and sets of definitions, maps elements of s to new formal entities, which in most cases are updated versions of the entities found in the environment. The interpretation relations are defined through rule templates of the form

$$\frac{\mathit{premises}}{\mathit{environment} \vdash t \xrightarrow{s} \mathit{result}}$$

Elements id of the syntactic set Identifier serve as names and references. An efficient mapping lookup mechanism with elements of Identifier as keys is assumed for an implementation. Rules for interpreting elements of Type-Spec and Literal, which depend on the actual form of the attribute type system, are not given in this work. Also, CALM makes use of the sets $\mathbb{N}_0$ and $\mathbb{N}_0^{\star}$ (non-negative integers and non-negative integers including the special character $\star$). Elements of $\mathbb{N}_0$ and $\mathbb{N}_0^{\star}$ stand for both, their syntactical representation and the numeral they denote.

The interpretation rule templates are arranged in such a way that their numbers correspond to the syntax generation rule templates of the syntactic set they interpret.

B.1 STYLE TIER

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id \notin \mathrm{dom}(\sigma), \quad i \in \text{Import-Spec}, \quad \sigma \vdash i \xrightarrow{\text{Import-spec}} \hat{\mathrm{T}}_0, \gamma_0, \kappa_0, \Xi_0, \\ b \in \text{Style-Body}, \quad \hat{\mathrm{T}}_0, \gamma_0, \kappa_0, \Xi_0, \emptyset, \emptyset \vdash b \xrightarrow{\text{Style-Body}} \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \mathcal{V}, \mathcal{C} \end{array}}{\sigma \vdash \mathbf{style}\ id_0\ i\ \{\ b\ \} \xrightarrow{\text{Style}} \sigma[id : (\mathbf{s}\ (\hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \mathrm{cls}_{\mathcal{V}}(\Xi_1)))]} \tag{B.1}$$

$$\frac{}{\sigma \vdash \epsilon \xrightarrow{\text{Import-spec}} \emptyset, \emptyset, \emptyset, \emptyset} \tag{B.2}$$

$$\frac{t \in \text{Union}, \quad \sigma \vdash t \xrightarrow{\text{Union}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi}{\sigma \vdash \mathbf{include}\ t \xrightarrow{\text{Import-spec}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \tag{B.3}$$

$$\frac{t \in \text{Intersection}, \quad \sigma \vdash t \xrightarrow{\text{Intersection}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi}{\sigma \vdash t \xrightarrow{\text{Union}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \tag{B.4}$$

$$\frac{\begin{array}{c} t_1 \in \text{Intersection}, \quad t_2 \in \text{Union}, \quad \sigma \vdash t_1 \xrightarrow{\text{Intersection}} \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \\ \sigma \vdash t_2 \xrightarrow{\text{Union}} \hat{\mathrm{T}}_2, \gamma_2, \kappa_2, \Xi_2 \end{array}}{\sigma \vdash t_1 + t_2 \xrightarrow{\text{Union}} \hat{\mathrm{T}}_1 \sqcup \hat{\mathrm{T}}_2, \gamma_1 \sqcup \gamma_2, \kappa_1 \sqcup \kappa_2, \Xi_1 \sqcup \Xi_2} \tag{B.5}$$

$$\frac{t \in \textsf{Atom}, \quad \sigma \vdash t \xrightarrow{\textsf{Atom}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi}{\sigma \vdash t \xrightarrow{\textsf{Intersection}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \tag{B.6}$$

$$\frac{\begin{array}{c} t_1 \in \textsf{Atom}, \quad t_2 \in \textsf{Intersection}, \quad \sigma \vdash t_1 \xrightarrow{\textsf{Atom}} \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \\ \sigma \vdash t_2 \xrightarrow{\textsf{Intersection}} \hat{\mathrm{T}}_2, \gamma_2, \kappa_2, \Xi_2 \end{array}}{\sigma \vdash t_1 \;\texttt{\^{}}\; t_2 \xrightarrow{\textsf{Intersection}} \hat{\mathrm{T}}_1 \sqcap \hat{\mathrm{T}}_2, \gamma_1 \sqcap \gamma_2, \kappa_1 \sqcap \kappa_2, \Xi_1 \sqcap \Xi_2} \tag{B.7}$$

$$\frac{id \in \textsf{Identifier}, \quad id : (\mathbf{s}\,(\hat{\mathrm{T}}, \gamma, \kappa, \Xi)) \in \sigma}{\sigma \vdash id \xrightarrow{\textsf{Atom}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \tag{B.8}$$

$$\frac{t \in \textsf{Union}, \quad \sigma \vdash t \xrightarrow{\textsf{Union}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi}{\sigma \vdash \texttt{(}\ t\ \texttt{)} \xrightarrow{\textsf{Atom}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi} \tag{B.9}$$

$$\frac{}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C} \vdash \epsilon \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C}} \tag{B.10}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Style-Body}, \quad \hat{\mathrm{T}}_0, \gamma_0, \kappa_0, \Xi_0, \mathcal{V}_0, \mathcal{C}_0 \vdash t_1 \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \mathcal{V}_1, \mathcal{C}_1, \\ \hat{\mathrm{T}}_1, \gamma_1, \kappa_1, \Xi_1, \mathcal{V}_1, \mathcal{C}_1 \vdash t_2 \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}_2, \gamma_2, \kappa_2, \Xi_2, \mathcal{V}_2, \mathcal{C}_2 \end{array}}{\hat{\mathrm{T}}_0, \gamma_0, \kappa_0, \Xi_0, \mathcal{V}_0, \mathcal{C}_0 \vdash t_1 \texttt{;}\ t_2 \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}_2, \gamma_2, \kappa_2, \Xi_2, \mathcal{V}_2, \mathcal{C}_2} \tag{B.11}$$

$$\frac{t \in \textsf{Meta-Kind-Spec}, \quad \hat{\mathrm{T}}, \gamma_0 \vdash t \xrightarrow{\textsf{Meta-Kind-Spec}} \gamma_1}{\hat{\mathrm{T}}, \gamma_0, \kappa, \Xi, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}, \gamma_1, \kappa, \Xi, \mathcal{V}, \mathcal{C}} \tag{B.12}$$

$$\frac{t \in \textsf{Kind-Definition}, \quad \hat{\mathrm{T}}, \gamma, \kappa_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Kind-Definition}} \kappa_1, \Xi_1}{\hat{\mathrm{T}}, \gamma, \kappa_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}, \gamma, \kappa_1, \Xi_1, \mathcal{V}, \mathcal{C}} \tag{B.13}$$

$$\frac{t \in \textsf{Type-Var-Decl}, \quad \mathcal{V}_0 \vdash t \xrightarrow{\textsf{Type-Var-Decl}} \mathcal{V}_1}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}_0, \mathcal{C} \vdash t \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}_1, \mathcal{C}} \tag{B.14}$$

$$\frac{t \in \textsf{Type-Var-Assert}, \quad \Xi_0, \mathcal{V} \vdash t \xrightarrow{\textsf{Type-Var-Assert}} \Xi_1}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi_1, \mathcal{V}, \mathcal{C}} \tag{B.15}$$

$$\frac{t \in \textsf{Attribute-Type-Spec}, \quad \hat{\mathrm{T}}_0 \vdash t \xrightarrow{\textsf{Attribute-Type-Spec}} \hat{\mathrm{T}}_1}{\hat{\mathrm{T}}_0, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}_1, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C}} \tag{B.16}$$

$$\frac{t \in \textsf{Constant}, \quad \hat{\mathrm{T}}, \mathcal{C}_0 \vdash t \xrightarrow{\textsf{Constant}} \mathcal{C}_1}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C}_0 \vdash t \xrightarrow{\textsf{Style-Body}} \hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C}_1} \tag{B.17}$$

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\gamma), \quad i \in \textsf{Extends-list}, \quad \gamma \vdash i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}_0, \emptyset, \emptyset, \\ b \in \textsf{Interface-MK-Body}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash b \xrightarrow{\textsf{Interface-MK-Body}} \hat{\alpha}_1 \end{array}}{\hat{\mathrm{T}}, \gamma \vdash \texttt{metainterface}\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\textsf{Meta-Kind-Spec}} \gamma[id : (\mathbb{i}\ \hat{\alpha}_1)]} \tag{B.18}$$

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\gamma), \quad i \in \textsf{Extends-list}, \quad \gamma \vdash i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}_0, \hat{\rho}_0, \emptyset, \\ b \in \textsf{Connector-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho}_0 \vdash b \xrightarrow{\textsf{Connector-MK-Body}} \hat{\alpha}_1, \hat{\rho}_1 \end{array}}{\hat{\mathrm{T}}, \gamma \vdash \texttt{metaconnector}\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\textsf{Meta-Kind-Spec}} \gamma[id : (\mathbb{l}\ (\hat{\alpha}_1, \hat{\rho}_1))]} \tag{B.19}$$

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\gamma), \quad i \in \textsf{Extends-list}, \quad \gamma \vdash i \xrightarrow{\textsf{Extends-list}} \hat{\alpha}_0, \emptyset, \hat{\pi}_0, \\ b \in \textsf{Component-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi}_0 \vdash b \xrightarrow{\textsf{Component-MK-Body}} \hat{\alpha}_1, \hat{\pi}_1 \end{array}}{\hat{\mathrm{T}}, \gamma \vdash \texttt{metacomponent}\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\textsf{Meta-Kind-Spec}} \gamma[id : (\mathbb{c}\ (\hat{\alpha}_1, \hat{\pi}_1))]} \tag{B.20}$$

$$\frac{}{\gamma \vdash \epsilon \xrightarrow{\textsf{Extends-list}} \emptyset, \emptyset, \emptyset} \tag{B.21}$$

$$\frac{i \in \text{Identifier-list}, \quad \gamma \vdash i \xrightarrow{\text{Extends-list}} \hat{\alpha}, \hat{\rho}, \hat{\pi}}{\gamma \vdash \mathtt{extends}\ i \xrightarrow{\text{Extends-list}} \hat{\alpha}, \hat{\rho}, \hat{\pi}} \tag{B.22}$$

$$\frac{id \in \text{Identifier}, \quad \gamma \vdash id : (\mathbb{i}\ \hat{\alpha})}{\gamma \vdash id \xrightarrow{\text{Extends-list}} \hat{\alpha}, \emptyset, \emptyset}, \quad \frac{id \in \text{Identifier}, \quad \gamma \vdash id : (\mathbb{l}\ (\hat{\alpha}, \hat{\rho}))}{\gamma \vdash id \xrightarrow{\text{Extends-list}} \hat{\alpha}, \hat{\rho}, \emptyset}, \tag{B.23}$$

$$\frac{id \in \text{Identifier}, \quad \gamma \vdash id : (\mathbb{c}\ (\hat{\alpha}, \hat{\pi}))}{\gamma \vdash id \xrightarrow{\text{Extends-list}} \hat{\alpha}, \emptyset, \hat{\pi}}$$

$$\frac{\begin{gathered} id \in \text{Identifier}, \quad \gamma \vdash id \xrightarrow{\text{Extends-list}} \hat{\alpha}_1, \hat{\rho}_1, \hat{\pi}_1, \\ i \in \text{Identifier-list}, \quad \gamma \vdash i \xrightarrow{\text{Extends-list}} \hat{\alpha}_2, \hat{\rho}_2, \hat{\pi}_2, \\ \operatorname{dom}(\hat{\alpha}_1) \cap \operatorname{dom}(\hat{\alpha}_2) = \emptyset, \quad \operatorname{dom}(\hat{\rho}_1) \cap \operatorname{dom}(\hat{\rho}_2) = \emptyset, \quad \operatorname{dom}(\hat{\pi}_1) \cap \operatorname{dom}(\hat{\pi}_2) = \emptyset \end{gathered}}{\gamma \vdash id\mathtt{,}\, i \xrightarrow{\text{Extends-list}} \hat{\alpha}_1 \cup \hat{\alpha}_2, \hat{\rho}_1 \cup \hat{\rho}_2, \hat{\pi}_1 \cup \hat{\pi}_2} \tag{B.24}$$

$$\frac{}{\hat{\mathrm{T}}, \hat{\alpha} \vdash \epsilon \xrightarrow{\text{Interface-MK-Body}} \hat{\alpha}} \tag{B.25}$$

$$\frac{\begin{gathered} t_1, t_2 \in \text{Interface-MK-Body}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash t_1 \xrightarrow{\text{Interface-MK-Body}} \hat{\alpha}_1, \\ \hat{\mathrm{T}}, \hat{\alpha}_1 \vdash t_2 \xrightarrow{\text{Interface-MK-Body}} \hat{\alpha}_2 \end{gathered}}{\hat{\mathrm{T}}, \hat{\alpha}_0 \vdash t_1\mathtt{;}\ t_2 \xrightarrow{\text{Interface-MK-Body}} \hat{\alpha}_2} \tag{B.26}$$

$$\frac{a \in \text{Attribute}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\text{Attribute}} \hat{\alpha}_1}{\hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\text{Interface-MK-Body}} \hat{\alpha}_1} \tag{B.27}$$

$$\frac{}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\rho} \vdash \epsilon \xrightarrow{\text{Connector-MK-Body}} \hat{\alpha}, \hat{\rho}} \tag{B.28}$$

$$\frac{\begin{gathered} t_1, t_2 \in \text{Connector-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho}_0 \vdash t_1 \xrightarrow{\text{Connector-MK-Body}} \hat{\alpha}_1, \hat{\rho}_1, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}_1, \hat{\rho}_1 \vdash t_2 \xrightarrow{\text{Connector-MK-Body}} \hat{\alpha}_2, \hat{\rho}_2 \end{gathered}}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho}_0 \vdash t_1\mathtt{;}\ t_2 \xrightarrow{\text{Connector-MK-Body}} \hat{\alpha}_2, \hat{\rho}_2} \tag{B.29}$$

$$\frac{a \in \text{Attribute}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\text{Attribute}} \hat{\alpha}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\rho} \vdash a \xrightarrow{\text{Connector-MK-Body}} \hat{\alpha}_1, \hat{\rho}} \tag{B.30}$$

$$\frac{r \in \text{Role-Option}, \quad \gamma, \hat{\rho}_0 \vdash r \xrightarrow{\text{Role-Option}} \hat{\rho}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\rho}_0 \vdash r \xrightarrow{\text{Connector-MK-Body}} \hat{\alpha}, \hat{\rho}_1} \tag{B.31}$$

$$\frac{}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\pi} \vdash \epsilon \xrightarrow{\text{Component-MK-Body}} \hat{\alpha}, \hat{\pi}} \tag{B.32}$$

$$\frac{\begin{gathered} t_1, t_2 \in \text{Component-MK-Body}, \quad \hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi}_0 \vdash t_1 \xrightarrow{\text{Component-MK-Body}} \hat{\alpha}_1, \hat{\pi}_1, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}_1, \hat{\pi}_1 \vdash t_2 \xrightarrow{\text{Component-MK-Body}} \hat{\alpha}_2, \hat{\pi}_2 \end{gathered}}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi}_0 \vdash t_1\mathtt{;}\ t_2 \xrightarrow{\text{Component-MK-Body}} \hat{\alpha}_2, \hat{\pi}_2} \tag{B.33}$$

$$\frac{a \in \text{Attribute}, \quad \hat{\mathrm{T}}, \hat{\alpha}_0 \vdash a \xrightarrow{\text{Attribute}} \hat{\alpha}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}_0, \hat{\pi} \vdash a \xrightarrow{\text{Component-MK-Body}} \hat{\alpha}_1, \hat{\pi}} \tag{B.34}$$

$$\frac{p \in \text{Port-Option}, \quad \gamma, \hat{\pi}_0 \vdash p \xrightarrow{\text{Port-Option}} \hat{\pi}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \hat{\pi}_0 \vdash p \xrightarrow{\text{Component-MK-Body}} \hat{\alpha}, \hat{\pi}_1} \tag{B.35}$$

$$\frac{id \in \text{Identifier}, \quad id \notin \operatorname{dom}(\hat{\alpha}), \quad t \in \text{Type-Spec}, \quad \hat{\mathrm{T}} \vdash t \xrightarrow{\text{Type-Spec}} \hat{\tau}}{\hat{\mathrm{T}}, \hat{\alpha} \vdash \mathtt{attribute}\ id\ \mathtt{:}\ t \xrightarrow{\text{Attribute}} \hat{\alpha}[id : (\mathbb{a}\ \hat{\tau})]} \tag{B.36}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\rho}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\rho} \vdash \mathbf{\texttt{uses}}\ t_1\ id_1\ \mathbf{\texttt{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Role\text{-}Option}} \hat{\rho}[id_1 : (\mathbb{r}\ (\mathbf{\texttt{uses}}, id_2, q_1, q_2))]} \tag{B.37}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\rho}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\rho} \vdash \mathbf{\texttt{provides}}\ t_1\ id_1\ \mathbf{\texttt{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Role\text{-}Option}} \hat{\rho}[id_1 : (\mathbb{r}\ (\mathbf{\texttt{provides}}, id_2, q_1, q_2))]} \tag{B.38}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\pi}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\pi} \vdash \mathbf{\texttt{uses}}\ t_1\ id_1\ \mathbf{\texttt{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Port\text{-}Option}} \hat{\pi}[id_1 : (\mathbb{p}\ (\mathbf{\texttt{uses}}, id_2, q_1, q_2))]} \tag{B.39}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Range}, \quad t_1 \xrightarrow{\mathsf{Range}} q_1, \quad t_2 \xrightarrow{\mathsf{Range}} q_2, \\ id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \mathrm{dom}(\hat{\pi}), \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma \end{array}}{\gamma, \hat{\pi} \vdash \mathbf{\texttt{provides}}\ t_1\ id_1\ \mathbf{\texttt{:}}\ id_2\ t_2 \xrightarrow{\mathsf{Port\text{-}Option}} \hat{\pi}[id_1 : (\mathbb{p}\ (\mathbf{\texttt{provides}}, id_2, q_1, q_2))]} \tag{B.40}$$

$$\frac{n, m \in \mathbb{N}_0, \quad n \leq m}{\texttt{[}n\texttt{..}m\texttt{]} \xrightarrow{\mathsf{Range}} \{n, \ldots, m\}}, \qquad \frac{n \in \mathbb{N}_0}{\texttt{[}n\texttt{..*]} \xrightarrow{\mathsf{Range}} \{n, \ldots\}} \tag{B.41}$$

$$\frac{n \in \mathbb{N}_0}{\texttt{[}n\texttt{]} \xrightarrow{\mathsf{Range}} \{n\}} \tag{B.42}$$

$$\frac{\begin{array}{c} id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \kappa, \quad id_2 : (\mathbb{i}\ \hat{\alpha}) \in \gamma, \quad b \in \mathsf{Interface\text{-}Export\text{-}Spec}, \\ \hat{\mathrm{T}}, \hat{\alpha}, \emptyset, \mathcal{C} \vdash b \xrightarrow{\mathsf{Interface\text{-}Export\text{-}Spec}} \bar{\alpha}_1, \quad \mathcal{A}_k^m(\hat{\alpha}) = \bar{\alpha}_2, \quad \bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2, \quad \mathcal{Q}_a^m(\hat{\alpha}, \bar{\alpha}) \end{array}}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi, \mathcal{V}, \mathcal{C} \vdash \mathbf{\texttt{interfacekind}}\ id_1\ \mathbf{\texttt{:}}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\mathsf{Kind\text{-}Definition}} \kappa[id_1 : (\mathbf{i}\ \bar{\alpha})], \Xi} \tag{B.43}$$

$$\frac{\begin{array}{c} id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \kappa, \quad id_2 : (\mathbb{l}\ (\hat{\alpha}, \hat{\rho})) \in \gamma, \quad b \in \mathsf{Connector\text{-}Export\text{-}Spec}, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \emptyset, \hat{\rho}, \emptyset, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash b \xrightarrow{\mathsf{Connector\text{-}Export\text{-}Spec}} \bar{\alpha}_1, \bar{\rho}, \Xi_1, \mathcal{V}_1, \\ \mathcal{A}_k^m(\hat{\alpha}) = \bar{\alpha}_2, \quad \bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2, \quad \mathcal{Q}_a^m(\hat{\alpha}, \bar{\alpha}), \quad \mathcal{Q}_r^m(\hat{\rho}, \bar{\rho}), \quad \Xi_2 = \mathrm{cls}_{(\mathcal{V}_1 \setminus \mathcal{V}_0)}(\Xi_1) \end{array}}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash \mathbf{\texttt{connectorkind}}\ id_1\ \mathbf{\texttt{:}}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\mathsf{Kind\text{-}Definition}} \kappa[id_1 : (\mathbf{l}\ (\bar{\alpha}, \bar{\rho}))], \Xi_2} \tag{B.44}$$

$$\frac{\begin{array}{c} id_1, id_2 \in \mathsf{Identifier}, \quad id_1 \notin \kappa, \quad id_2 : (\mathbb{c}\ (\hat{\alpha}, \hat{\pi})) \in \gamma, \quad b \in \mathsf{Component\text{-}Export\text{-}Spec}, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \emptyset, \hat{\pi}, \emptyset, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash b \xrightarrow{\mathsf{Component\text{-}Export\text{-}Spec}} \bar{\alpha}_1, \bar{\pi}, \Xi_1, \mathcal{V}_1, \\ \mathcal{A}_k^m(\hat{\alpha}) = \bar{\alpha}_2, \quad \bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2, \quad \mathcal{Q}_a^m(\hat{\alpha}, \bar{\alpha}), \quad \mathcal{Q}_p^m(\hat{\pi}, \bar{\pi}) \quad \Xi_2 = \mathrm{cls}_{(\mathcal{V}_1 \setminus \mathcal{V}_0)}(\Xi_1) \end{array}}{\hat{\mathrm{T}}, \gamma, \kappa, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash \mathbf{\texttt{componentkind}}\ id_1\ \mathbf{\texttt{:}}\ id_2\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\mathsf{Kind\text{-}Definition}} \kappa[id_1 : (\mathbf{c}\ (\bar{\alpha}, \bar{\pi}))], \Xi_2} \tag{B.45}$$

$$\frac{}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}, \mathcal{C} \vdash \epsilon \xrightarrow{\mathsf{Interface\text{-}Export\text{-}Spec}} \bar{\alpha}} \tag{B.46}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Interface\text{-}Export\text{-}Spec}, \quad \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t_1 \xrightarrow{\mathsf{Interface\text{-}Export\text{-}Spec}} \bar{\alpha}_1, \\ \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_1, \mathcal{C} \vdash t_2 \xrightarrow{\mathsf{Interface\text{-}Export\text{-}Spec}} \bar{\alpha}_2 \end{array}}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t_1\texttt{;}\ t_2 \xrightarrow{\mathsf{Interface\text{-}Export\text{-}Spec}} \bar{\alpha}_2} \tag{B.47}$$

$$\frac{t \in \mathsf{Attribute\text{-}Valuation}, \quad \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\mathsf{Attribute\text{-}Valuation}} \bar{\alpha}_1}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\mathsf{Interface\text{-}Export\text{-}Spec}} \bar{\alpha}_1} \tag{B.48}$$

$$\frac{}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\rho}, \bar{\rho}, \Xi, \mathcal{V}, \mathcal{C} \vdash \epsilon \xrightarrow{\mathsf{Connector\text{-}Export\text{-}Spec}} \bar{\alpha}, \bar{\rho}, \Xi, \mathcal{V}} \tag{B.49}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \mathsf{Connector\text{-}Export\text{-}Spec}, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\rho}, \bar{\rho}_0, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash t_1 \xrightarrow{\mathsf{Connector\text{-}Export\text{-}Spec}} \bar{\alpha}_1, \bar{\rho}_1, \Xi_1, \mathcal{V}_1 \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_1, \hat{\rho}, \bar{\rho}_1, \Xi_1, \mathcal{V}_1, \mathcal{C} \vdash t_2 \xrightarrow{\mathsf{Connector\text{-}Export\text{-}Spec}} \bar{\alpha}_2, \bar{\rho}_2, \Xi_2, \mathcal{V}_2 \end{array}}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\rho}, \bar{\rho}_0, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash t_1\texttt{;}\ t_2 \xrightarrow{\mathsf{Connector\text{-}Export\text{-}Spec}} \bar{\alpha}_2, \bar{\rho}_2, \Xi_2, \mathcal{V}_2} \tag{B.50}$$

$$\frac{t \in \text{Attribute-Valuation}, \quad \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\text{Attribute-Valuation}} \bar{\alpha}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\rho}, \bar{\rho}, \Xi, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Connector-Export-Spec}} \bar{\alpha}_1, \bar{\rho}, \Xi, \mathcal{V}} \tag{B.51}$$

$$\frac{t \in \text{Export-Kind-Spec}, \quad \gamma, \hat{\rho}, \bar{\rho}_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Export-Kind-Spec}} \bar{\rho}_1, \Xi_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\rho}, \bar{\rho}_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Connector-Export-Spec}} \bar{\alpha}, \bar{\rho}_1, \Xi_1, \mathcal{V}} \tag{B.52}$$

$$\frac{t \in \text{Type-Var-Decl}, \quad \kappa, \mathcal{V}_0 \vdash t \xrightarrow{\text{Type-Var-Decl}} \mathcal{V}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\rho}, \bar{\rho}, \Xi, \mathcal{V}_0, \mathcal{C} \vdash t \xrightarrow{\text{Connector-Export-Spec}} \bar{\alpha}, \bar{\rho}, \Xi, \mathcal{V}_1} \tag{B.53}$$

$$\frac{t \in \text{Type-Var-Assert}, \quad \Xi_0, \mathcal{V} \vdash t \xrightarrow{\text{Type-Var-Assert}} \Xi_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\rho}, \bar{\rho}, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Connector-Export-Spec}} \bar{\alpha}, \bar{\rho}, \Xi_1, \mathcal{V}} \tag{B.54}$$

$$\frac{}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash \epsilon \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}, \Xi, \mathcal{V}} \tag{B.55}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \text{Component-Export-Spec}, \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash t_1 \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}_1, \bar{\pi}_1, \Xi_1, \mathcal{V}_1 \\ \hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_1, \hat{\pi}, \bar{\pi}_1, \Xi_1, \mathcal{V}_1, \mathcal{C} \vdash t_2 \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}_2, \bar{\pi}_2, \Xi_2, \mathcal{V}_2 \end{array}}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}_0, \mathcal{C} \vdash t_1\ \mathtt{;}\ t_2 \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}_2, \bar{\pi}_2, \Xi_2, \mathcal{V}_2} \tag{B.56}$$

$$\frac{t \in \text{Attribute-Valuation}, \quad \hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\text{Attribute-Valuation}} \bar{\alpha}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}_0, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}_1, \bar{\pi}, \Xi, \mathcal{V}} \tag{B.57}$$

$$\frac{t \in \text{Export-Kind-Spec}, \quad \gamma, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Export-Kind-Spec}} \bar{\pi}_1, \Xi_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}_0, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}_1, \Xi_1, \mathcal{V}} \tag{B.58}$$

$$\frac{t \in \text{Type-Var-Decl}, \quad \kappa, \mathcal{V}_0 \vdash t \xrightarrow{\text{Type-Var-Decl}} \mathcal{V}_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}_0, \mathcal{C} \vdash t \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}, \Xi, \mathcal{V}_1} \tag{B.59}$$

$$\frac{t \in \text{Type-Var-Assert}, \quad \Xi_0, \mathcal{V} \vdash t \xrightarrow{\text{Type-Var-Assert}} \Xi_1}{\hat{\mathrm{T}}, \gamma, \hat{\alpha}, \bar{\alpha}, \hat{\pi}, \bar{\pi}, \Xi_0, \mathcal{V}, \mathcal{C} \vdash t \xrightarrow{\text{Component-Export-Spec}} \bar{\alpha}, \bar{\pi}, \Xi_1, \mathcal{V}} \tag{B.60}$$

$$\frac{id \in \text{Identifier}, \quad id : \hat{\tau} \in \hat{\alpha}, \quad id \notin \mathrm{dom}(\bar{\alpha}), \quad v \in \text{Literal}, \quad \hat{\mathrm{T}}, \hat{\tau}, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \bar{\tau}}{\hat{\mathrm{T}}, \hat{\alpha}, \bar{\alpha}, \mathcal{C} \vdash id \mathtt{=} v \xrightarrow{\text{Attribute-Valuation}} \bar{\alpha}[id : \bar{\tau}]} \tag{B.61}$$

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : (\mathbb{r}\ (p, id_m, q_1, q_2)) \in \hat{\rho}, \quad id_2 : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad \mathcal{D}(id_m, id_2)}{\gamma, \hat{\rho}, \bar{\rho}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \mathtt{->} id_2 \xrightarrow{\text{Export-Kind-Spec}} \bar{\rho}[id_1 : (\mathbf{r}\ (p, id_2, q_1, q_2))], \Xi} \tag{B.62}$$

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : (\mathbb{r}\ (p, id_m, q_1, q_2)) \in \hat{\rho}, \quad id_2 : id_k \in \mathcal{V}, \quad \mathcal{D}(id_m, id_k)}{\begin{array}{c}\gamma, \hat{\rho}, \bar{\rho}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \mathtt{->} id_2 \xrightarrow{\text{Export-Kind-Spec}} \bar{\rho}[id_1 : (\mathbf{r}\ (p, id_k, q_1, q_2))], \\ \Xi[\mathrm{type}(this.id_1) = id_2]\end{array}}$$

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : (\mathbb{p}\ (p, id_m, q_1, q_2)) \in \hat{\pi}, \quad id_2 : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad \mathcal{D}(id_m, id_2)}{\gamma, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \mathtt{->} id_2 \xrightarrow{\text{Export-Kind-Spec}} \bar{\pi}[id_1 : (\mathbf{p}\ (p, id_2, q_1, q_2))], \Xi}$$

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : (\mathbb{p}\ (p, id_m, q_1, q_2)) \in \hat{\pi}, \quad id_2 : id_k \in \mathcal{V}, \quad \mathcal{D}(id_m, id_k)}{\begin{array}{c}\gamma, \hat{\pi}, \bar{\pi}, \Xi, \mathcal{V}, \mathcal{C} \vdash id_1 \mathtt{->} id_2 \xrightarrow{\text{Export-Kind-Spec}} \bar{\pi}[id_1 : (\mathbf{p}\ (p, id_k, q_1, q_2))], \\ \Xi[\mathrm{type}(this.id_1) = id_2]\end{array}}$$

$$\frac{i \in \text{Identifier-list}, \quad id \in \text{Identifier}, \quad id : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad \mathcal{V}_0, id \vdash i \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}_1}{\kappa, \mathcal{V}_0 \vdash \mathtt{typevar}\ i\ \mathtt{:}\ id \xrightarrow{\text{Type-Var-Decl}} \mathcal{V}_1} \tag{B.63}$$

$$\frac{id \in \text{Identifier}, \quad id \notin \text{dom}(\mathcal{V})}{\mathcal{V}, k \vdash id \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}[id : k]}$$

$$\frac{id \in \text{Identifier}, \quad id \notin \text{dom}(\mathcal{V}_0), \quad i \in \text{Identifier-list}, \quad \mathcal{V}_0[id : k], k \vdash i \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}_1}{\mathcal{V}_0, k \vdash id\texttt{,}\, i \xrightarrow{\text{Type-Var-Decl-Id}} \mathcal{V}_1}$$

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : k_1, id_2 : k_2 \in \mathcal{V}, \quad k_1 = k_2}{\Xi, \mathcal{V} \vdash \textbf{assert}\ id_1 \texttt{ <= } id_2 \xrightarrow{\text{Type-Var-Assert}} \Xi[id_1 \leq id_2]} \quad \text{(B.64)}$$

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : k_1, id_2 : k_2 \in \mathcal{V}, \quad k_1 = k_2}{\Xi, \mathcal{V} \vdash \textbf{assert}\ id_1 \texttt{ = } id_2 \xrightarrow{\text{Type-Var-Assert}} \Xi[id_1 = id_2]}$$

$$\frac{id_1, id_2 \in \text{Identifier}, \quad id_1 : k_1, id_2 : k_2 \in \mathcal{V}, \quad k_1 = k_2}{\Xi, \mathcal{V} \vdash \textbf{assert}\ id_1 \texttt{ >= } id_2 \xrightarrow{\text{Type-Var-Assert}} \Xi[id_1 \geq id_2]}$$

$$\frac{id \in \text{Identifier}, \quad id \notin \text{dom}(\hat{\mathrm{T}})}{\hat{\mathrm{T}} \vdash \textbf{typedecl}\ id \xrightarrow{\text{Attribute-Type-Spec}} \hat{\mathrm{T}}[id : \hat{\tau}^{id}]} \quad \text{(B.65)}$$

$$\frac{id \in \text{Identifier}, \quad id \notin \text{dom}(\hat{\mathrm{T}}), \quad t \in \text{Type-Spec}, \quad \hat{\mathrm{T}} \vdash t \xrightarrow{\text{Type-Spec}} \hat{\tau}^{t}}{\hat{\mathrm{T}} \vdash \textbf{typedef}\ id \texttt{ = } t \xrightarrow{\text{Attribute-Type-Spec}} \hat{\mathrm{T}}[id : \hat{\tau}^{t}]} \quad \text{(B.66)}$$

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id \notin \text{dom}(\mathcal{C}), \quad t \in \text{Type-Spec}, \quad \hat{\mathrm{T}} \vdash t \xrightarrow{\text{Type-Spec}} \hat{\tau}, \\ v \in \text{Literal}, \quad \hat{\mathrm{T}}, \hat{\tau}, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \bar{\tau} \end{array}}{\hat{\mathrm{T}}, \mathcal{C} \vdash id \texttt{ : } t \texttt{ = } v \xrightarrow{\text{Constant}} \mathcal{C}[id : \bar{\tau}]} \quad \text{(B.67)}$$

B.2 Module tier

$$\frac{\begin{array}{c} id_0, id_1 \in \text{Identifier}, \quad id_0 \notin \text{dom}(\mu), \quad id_1 : (\mathbf{s}\ (\hat{\mathrm{T}}, \gamma, \kappa, \Xi)) \in \sigma, \quad \mathcal{T}_k^m(\hat{\mathrm{T}}) = \bar{\mathrm{T}}_0, \\ i \in \text{Input}, \quad \mu \vdash i \xrightarrow{\text{Input}} \bar{\mathrm{T}}_1, \psi_0, \quad id_1 \mathrel{-\!\!\mid\!\!\models} \psi_0, \quad \bar{\mathrm{T}} = \bar{\mathrm{T}}_0 \cup \bar{\mathrm{T}}_1 \\ b \in \text{Module-Body}, \quad \bar{\mathrm{T}}, \kappa, \psi_0, \emptyset, \Xi \vdash b \xrightarrow{\text{Type-Body}} \psi_1, \mathcal{C} \end{array}}{\sigma, \mu \vdash \textbf{module}\ id_0\ \textbf{of}\ id_1\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\text{Module}} \mu[id_0 : (\mathbf{m}\ (\bar{\mathrm{T}}, \psi))]} \quad \text{(B.68)}$$

$$\frac{}{\mu \vdash \epsilon \xrightarrow{\text{Input}} \emptyset, \emptyset} \quad \text{(B.69)} \qquad \frac{i \in \text{Identifier-list}, \quad \mu \vdash i \xrightarrow{\text{Input}} \bar{\mathrm{T}}, \psi}{\mu \vdash \textbf{input}\ i \xrightarrow{\text{Input}} \bar{\mathrm{T}}, \psi} \quad \text{(B.70)}$$

$$\frac{id \in \text{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}, \psi)) \in \mu}{\mu \vdash id \xrightarrow{\text{Input}} \bar{\mathrm{T}}, \psi} \quad \text{(B.71)}$$

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}_0, \psi_0)) \in \mu, \quad i \in \text{Identifier-list}, \quad \mu \vdash i \xrightarrow{\text{Input}} \bar{\mathrm{T}}_1, \psi_1 \\ \text{dom}(\psi_0) \cap \text{dom}(\psi_1) = \emptyset \end{array}}{\mu \vdash id\texttt{,}\, i \xrightarrow{\text{Input}} \bar{\mathrm{T}}_0 \cup \bar{\mathrm{T}}_1, \psi_0 \cup \psi_1} \quad \text{(B.72)}$$

$$\frac{}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}, \Xi \vdash \epsilon \xrightarrow{\text{Module-Body}} \psi, \mathcal{C}} \quad \text{(B.73)}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \text{Module-Body}, \quad \bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}_0, \Xi \vdash t_1 \xrightarrow{\text{Module-Body}} \psi_1, \mathcal{C}_1 \\ \bar{\mathrm{T}}, \kappa, \psi_1, \mathcal{C}_1, \Xi \vdash t_1 \xrightarrow{\text{Module-Body}} \psi_2, \mathcal{C}_2 \end{array}}{\bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}_0, \Xi \vdash t_1\texttt{;}\ t_2 \xrightarrow{\text{Module-Body}} \psi_2, \mathcal{C}_2} \quad \text{(B.74)}$$

$$\frac{t \in \text{Type}, \quad \bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\text{Type}} \psi_1}{\bar{\mathrm{T}}, \kappa, \psi_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\text{Module-Body}} \psi_1, \mathcal{C}} \quad \text{(B.75)}$$

$$\frac{t \in \textsf{Constant}, \quad \bar{\mathrm{T}}, \mathcal{C}_0 \vdash \xrightarrow{\textsf{Constant}} \mathcal{C}_1}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}_0, \Xi \vdash t \xrightarrow{\textsf{Module-Body}} \psi, \mathcal{C}_1} \tag{B.76}$$

$$\frac{\begin{array}{c} id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbf{i}\ \bar{\alpha}) \in \kappa, \quad id_2 \notin \mathrm{dom}(\psi), \quad i \in \textsf{Include}, \\ \psi, id_1 \vdash i \xrightarrow{\textsf{Include}} \alpha_0, \emptyset, \quad b \in \textsf{Type-Body}, \quad \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \emptyset, \emptyset, \mathcal{C} \vdash b \xrightarrow{\textsf{Type-Body}} \alpha_1, \emptyset \\ \mathcal{A}^k_t(\bar{\alpha}) = \alpha_2, \quad \alpha = \alpha_1 \cup \alpha_2, \quad \mathcal{Q}^k_a(\bar{\alpha}, \alpha) \end{array}}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}, \Xi \vdash id\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\textsf{Type}} \psi[id_2 : (id_1\ \alpha_2)]} \tag{B.77}$$

$$\frac{\begin{array}{c} id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (\mathbf{c}\ (\bar{\alpha}, \bar{\pi})) \in \kappa, \quad id_2 \notin \mathrm{dom}(\psi), \quad i \in \textsf{Include}, \\ \psi, id_1 \vdash i \xrightarrow{\textsf{Include}} \alpha_0, \pi_0, \quad b \in \textsf{Type-Body}, \quad \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi_0, \mathcal{C} \vdash b \xrightarrow{\textsf{Type-Body}} \alpha_1, \pi \\ \mathcal{A}^k_t(\bar{\alpha}) = \alpha_2, \quad \alpha = \alpha_1 \cup \alpha_2, \quad \mathcal{Q}^k_a(\bar{\alpha}, \alpha), \quad \mathcal{Q}^k_p(\bar{\pi}, \pi), \quad \mathcal{K}(\Xi, \pi) \end{array}}{\bar{\mathrm{T}}, \kappa, \psi, \mathcal{C}, \Xi \vdash id\ id\ i\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\textsf{Type}} \psi[id_2 : (id_1\ (\alpha_2, \pi_2))]}$$

$$\frac{}{\psi, id \vdash \epsilon \xrightarrow{\textsf{Include}} \emptyset, \emptyset} \tag{B.78}$$

$$\frac{i \in \textsf{Identifier-list}, \quad \psi, id \vdash i \xrightarrow{\textsf{Include}} \alpha, \pi}{\psi, id \vdash \texttt{include}\ i \xrightarrow{\textsf{Include}} \alpha, \pi} \tag{B.79}$$

$$\frac{id \in \textsf{Identifier}, \quad id : (id_i\ \alpha) \in \psi}{\psi, id_i \vdash id \xrightarrow{\textsf{Include}} \alpha, \emptyset} \qquad \frac{id \in \textsf{Identifier}, \quad id : (id_c\ (\alpha, \pi)) \in \psi}{\psi, id_c \vdash id \xrightarrow{\textsf{Include}} \alpha, \pi}$$

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}, \quad id : (id_i\ \alpha_1) \in \psi, \quad \psi \vdash i \xrightarrow{\textsf{Include}} \alpha_2, \emptyset \\ \mathrm{dom}(\alpha_1) \cap \mathrm{dom}(\alpha_2) = \emptyset \end{array}}{\psi, id_i \vdash id\texttt{,}\, i \xrightarrow{\textsf{Include}} \alpha_1 \cup \alpha_2, \emptyset}$$

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad i \in \textsf{Identifier-list}, \quad id : (id_c\ (\alpha_1, \pi_1)) \in \psi, \quad \psi \vdash i \xrightarrow{\textsf{Include}} \alpha_2, \pi_2, \\ \mathrm{dom}(\alpha_1) \cap \mathrm{dom}(\alpha_2) = \emptyset, \quad \mathrm{dom}(\pi_1) \cap \mathrm{dom}(\pi_2) = \emptyset \end{array}}{\psi, id_c \vdash id\texttt{,}\, i \xrightarrow{\textsf{Include}} \alpha_1 \cup \alpha_2, \pi_1 \cup \pi_2}$$

$$\frac{}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha, \bar{\pi}, \pi, \mathcal{C} \vdash \epsilon \xrightarrow{\textsf{Type-Body}} \alpha, \pi} \tag{B.80}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \textsf{Type-Body}, \quad \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi_0, \mathcal{C} \vdash t_1 \xrightarrow{\textsf{Type-Body}} \alpha_1, \pi_1 \\ \bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_1, \bar{\pi}, \pi_1, \mathcal{C} \vdash t_2 \xrightarrow{\textsf{Type-Body}} \alpha_2, \pi_2 \end{array}}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi_0, \mathcal{C} \vdash t_1\texttt{;}\ t_2 \xrightarrow{\textsf{Type-Body}} \alpha_2, \pi_2} \tag{B.81}$$

$$\frac{t \in \textsf{Attribute-Valuation}, \quad \bar{\mathrm{T}}, \bar{\alpha}, \alpha_0, \mathcal{C} \vdash t \xrightarrow{\textsf{Attribute-Valuation}} \alpha_1}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha_0, \bar{\pi}, \pi, \mathcal{C} \vdash t \xrightarrow{\textsf{Type-Body}} \alpha_1, \pi} \tag{B.82}$$

$$\frac{t \in \textsf{Port}, \quad \psi, \bar{\pi}, \pi_0 \vdash t \xrightarrow{\textsf{Port}} \pi_1}{\bar{\mathrm{T}}, \psi, \bar{\alpha}, \alpha, \bar{\pi}_0, \pi, \mathcal{C} \vdash t \xrightarrow{\textsf{Type-Body}} \alpha_1, \pi_1} \tag{B.83}$$

$$\frac{id \in \textsf{Identifier}, \quad id : \bar{\tau} \in \bar{\alpha}, \quad id \notin \mathrm{dom}(\alpha), \quad v \in \textsf{Literal}, \quad \bar{\mathrm{T}}, \bar{\tau}, \mathcal{C} \vdash v \xrightarrow{\textsf{Literal}} \tau}{\bar{\mathrm{T}}, \bar{\alpha}, \alpha, \mathcal{C} \vdash id\ \texttt{=}\ v \xrightarrow{\textsf{Attribute-Valuation}} \alpha[id : \tau]} \tag{B.84}$$

$$\frac{id_1, id_2, id_3 \in \textsf{Identifier}, \quad id_1 : (\mathbf{p}\ (p, k, q_1, q_2)) \in \bar{\pi}, \quad id_2 \notin \mathrm{dom}(\pi), \quad id_3 : (k\ \alpha) \in \psi}{\psi, \bar{\pi}, \pi \vdash id_1\ id_2\ \texttt{:}\ id_3 \xrightarrow{\textsf{Port}} \pi[id_2 : (id_1\ (p, id_3, q_2))]} \tag{B.85}$$

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\mathcal{C}), \quad t \in \textsf{Type-Spec}, \quad \bar{\mathrm{T}} \vdash t \xrightarrow{\textsf{Type-Spec}} \bar{\tau}, \\ v \in \textsf{Literal}, \quad \bar{\mathrm{T}}, \bar{\tau}, \mathcal{C} \vdash v \xrightarrow{\textsf{Literal}} \tau \end{array}}{\bar{\mathrm{T}}, \mathcal{C} \vdash id\ \texttt{:}\ t\ \texttt{=}\ v \xrightarrow{\textsf{Constant}} \mathcal{C}[id : \tau]} \tag{B.86}$$

B.3 Scenario tier

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id \notin \mathrm{dom}(\zeta), \quad i \in \text{Identifier-list}, \quad \mu \vdash i \xrightarrow{\text{Include}} \bar{\mathrm{T}}_0, \psi, \\ \mathrm{Collect}(\mathcal{S}(\psi)) = (\hat{\mathrm{T}}, \gamma, \kappa, \Xi), \quad \mathcal{T}_k^m(\hat{\mathrm{T}}) = \bar{\mathrm{T}}_1, \quad \bar{\mathrm{T}} = \bar{\mathrm{T}}_0 \cup \bar{\mathrm{T}}_1, \quad \mathcal{T}_t^k(\hat{\mathrm{T}}) = \mathrm{T}, \\ b \in \text{Scenario-Body}, \quad \mathrm{T}, \kappa, \psi, \emptyset, \emptyset, \emptyset, \Xi \vdash b \xrightarrow{\text{Scenario-Body}} \psi_r, \theta, \Lambda, \mathcal{C}, \quad \mathcal{M}(\Lambda), \quad \mathcal{S}(\psi_r) \neq \emptyset \end{array}}{\sigma, \mu, \zeta \vdash \mathbf{scenario}\ id\ \mathbf{includes}\ i\ \{\ b\ \} \xrightarrow{\text{Scenario}} \zeta[id : (\mathbf{z}\ (\theta, \Lambda))]} \tag{B.87}$$

$$\frac{id \in \text{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}, \psi)) \in \mu}{\mu \vdash id \xrightarrow{\text{Include}} \bar{\mathrm{T}}, \psi} \tag{B.88}$$

$$\frac{id \in \text{Identifier}, \quad id : (\mathbf{m}\ (\bar{\mathrm{T}}_1, \psi_1)) \in \mu, \quad i \in \text{Identifier-list}, \quad \mu \vdash i \xrightarrow{\text{Include}} \bar{\mathrm{T}}_2, \psi_2}{\mu \vdash id, i \xrightarrow{\text{Include}} \bar{\mathrm{T}}_1 \sqcup \bar{\mathrm{T}}_2, \psi_1 \sqcup \psi_2} \tag{B.89}$$

$$\frac{}{\mathrm{T}, \kappa, \psi, \theta, \Lambda, \mathcal{C}, \Xi \vdash \epsilon \xrightarrow{\text{Scenario-Body}} \psi, \theta, \Lambda, \mathcal{C}} \tag{B.90}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \text{Scenario-Body}, \quad \mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}_0, \Xi \vdash t_1 \xrightarrow{\text{Scenario-Body}} \psi_1, \theta_1, \Lambda_1, \mathcal{C}_1, \\ \mathrm{T}, \kappa, \psi_1, \theta_1, \Lambda_1, \mathcal{C}_1, \Xi \vdash t_2 \xrightarrow{\text{Scenario-Body}} \psi_2, \theta_2, \Lambda_2, \mathcal{C}_2 \end{array}}{\mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}_0, \Xi \vdash t_1;\ t_2 \xrightarrow{\text{Scenario-Body}} \psi_2, \theta_2, \Lambda_2, \mathcal{C}_2} \tag{B.91}$$

$$\frac{t \in \text{Component}, \quad \mathrm{T}, \psi, \theta_0, \mathcal{C} \vdash t \xrightarrow{\text{Component}} \theta_1}{\mathrm{T}, \kappa, \psi, \theta_0, \Lambda, \mathcal{C}, \Xi \vdash t \xrightarrow{\text{Scenario-Body}} \psi, \theta_1, \Lambda, \mathcal{C}} \tag{B.92}$$

$$\frac{t \in \text{Connector}, \quad \mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\text{Connector}} \psi_1, \theta_1, \Lambda_1}{\mathrm{T}, \kappa, \psi_0, \theta_0, \Lambda_0, \mathcal{C}, \Xi \vdash t \xrightarrow{\text{Scenario-Body}} \psi_1, \theta_1, \Lambda_1, \mathcal{C}} \tag{B.93}$$

$$\frac{t \in \text{Constant}, \quad \mathrm{T}, \mathcal{C}_0 \vdash t \xrightarrow{\text{Constant}} \mathcal{C}_1}{\mathrm{T}, \kappa, \psi, \theta, \Lambda, \mathcal{C}_0, \Xi \vdash t \xrightarrow{\text{Scenario-Body}} \psi, \theta, \Lambda, \mathcal{C}_1} \tag{B.94}$$

$$\frac{\begin{array}{c} id_0, id_1 \in \text{Identifier}, \quad id_0 : (id_k\ (\alpha, \pi)) \in \psi, \quad id_1 \notin \theta_0, \quad b \in \text{Component-Body}, \\ \mathcal{A}_v^t(\alpha) = \dot{\alpha}_0, \quad \mathrm{T}, \psi, \theta_0, \alpha, \emptyset, \pi, \mathcal{C} \vdash b \xrightarrow{\text{Component-Body}} \theta_1, \dot{\alpha}_1, \quad \dot{\alpha} = \dot{\alpha}_0 \cup \dot{\alpha}_1, \quad \mathcal{Q}_a^t(\alpha, \dot{\alpha}) \\ \mathcal{P}_v^t(\pi) = \dot{\pi}, \quad \mathrm{IntGen}(\theta_1, \pi) = \theta_2, \quad \mathcal{I}[\theta_2, \pi] \end{array}}{\mathrm{T}, \psi, \theta_0, \mathcal{C} \vdash id_0\ id_1\ \{\ b\ \} \xrightarrow{\text{Component}} \theta_2[id_1 : (id_0\ (\dot{\alpha}, \dot{\pi}))]} \tag{B.95}$$

$$\frac{}{\mathrm{T}, \psi, \theta, \alpha, \dot{\alpha}, \pi, \mathcal{C} \vdash \epsilon \xrightarrow{\text{Component-Body}} \theta, \dot{\alpha}} \tag{B.96}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \text{Component-Body}, \quad \mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}_0, \pi, \mathcal{C} \vdash t_1 \xrightarrow{\text{Component-Body}} \theta_1, \dot{\alpha}_1, \\ \mathrm{T}, \psi, \theta_1, \alpha, \dot{\alpha}_1, \pi, \mathcal{C} \vdash t_2 \xrightarrow{\text{Component-Body}} \theta_2, \dot{\alpha}_2 \end{array}}{\mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}_0, \pi, \mathcal{C} \vdash t_1; t_2 \xrightarrow{\text{Component-Body}} \theta_2, \dot{\alpha}_2} \tag{B.97}$$

$$\frac{t \in \text{Attribute-Valuation}, \quad \mathrm{T}, \alpha, \dot{\alpha}_0, \mathcal{C} \vdash t \xrightarrow{\text{Attribute-Valuation}} \dot{\alpha}_1}{\mathrm{T}, \psi, \theta, \alpha, \dot{\alpha}_0, \pi, \mathcal{C} \vdash t \xrightarrow{\text{Component-Body}} \theta, \dot{\alpha}_1} \tag{B.98}$$

$$\frac{t \in \text{Interface}, \quad \mathrm{T}, \psi, \theta_0, \pi, \mathcal{C} \vdash t \xrightarrow{\text{Interface}} \theta_1}{\mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}, \pi, \mathcal{C} \vdash t \xrightarrow{\text{Component-Body}} \theta_1, \dot{\alpha}} \tag{B.99}$$

$$\frac{id \in \text{Identifier}, \quad id : \tau \in \alpha, \quad id \notin \mathrm{dom}(\dot{\alpha}), \quad v \in \text{Literal}, \quad \mathrm{T}, \tau, \mathcal{C} \vdash v \xrightarrow{\text{Literal}} \dot{\tau}}{\mathrm{T}, \alpha, \dot{\alpha}, \mathcal{C} \vdash id = v \xrightarrow{\text{Attribute-Valuation}} \dot{\alpha}[id : \dot{\tau}]} \tag{B.100}$$

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (id_p\ (\mathbf{provides}, id_i, q)) \in \pi, \quad id_i : (id_k\ \alpha) \in \psi, \quad \mathcal{A}_v^t(\alpha) = \dot{\alpha}_0, \\ a \in \text{Attributes}, \quad \mathrm{T}, \alpha, \emptyset, \mathcal{C} \vdash t \xrightarrow{\text{Attributes}} \dot{\alpha}_1, \quad \dot{\alpha} = \dot{\alpha}_0 \cup \dot{\alpha}_1, \quad \mathcal{Q}_a^t(\alpha, \dot{\alpha}) \end{array}}{\mathrm{T}, \psi, \theta, \pi, \mathcal{C} \vdash id\ \{\ a\ \} \xrightarrow{\text{Interface}} \theta[this.id : (id_i\ \dot{\alpha})]} \tag{B.101}$$

$$\frac{}{\mathrm{T}, \alpha, \dot{\alpha}, \mathcal{C} \vdash \epsilon \xrightarrow{\text{Attributes}} \dot{\alpha}} \tag{B.102}$$

$$\frac{t_1, t_2 \in \text{Attributes}, \quad \mathrm{T}, \alpha, \dot{\alpha}_0, \mathcal{C} \vdash t_1 \xrightarrow{\text{Attributes}} \dot{\alpha}_1, \quad \mathrm{T}, \alpha, \dot{\alpha}_1, \mathcal{C} \vdash t_2 \xrightarrow{\text{Attributes}} \dot{\alpha}_2}{\mathrm{T}, \alpha, \dot{\alpha}_0, \mathcal{C} \vdash t_1 \texttt{;} t_2 \xrightarrow{\text{Attributes}} \dot{\alpha}_2} \tag{B.103}$$

$$\frac{a \in \text{Attribute-Valuation}, \quad \mathrm{T}, \alpha, \dot{\alpha}_0, \mathcal{C} \vdash a \xrightarrow{\text{Attribute-Valuation}} \dot{\alpha}_1}{\mathrm{T}, \alpha, \dot{\alpha}_0, \mathcal{C} \vdash a \xrightarrow{\text{Attributes}} \dot{\alpha}_1} \tag{B.104}$$

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (\mathbf{l}\ (\bar{\alpha}, \bar{\rho})), \quad \mathcal{A}^k_t(\bar{\alpha}) = \alpha, \quad \mathcal{A}^t_v(\alpha) = \dot{\alpha}_0, \quad b \in \text{Connector-Body} \\ \mathrm{T}, \psi, \theta_0, \alpha, \emptyset, \bar{\rho}, \emptyset, \Lambda_0, \mathcal{C} \vdash b \xrightarrow{\text{Connector-Body}} \theta_1, \dot{\alpha}_1, \rho, \Lambda_1, \quad \dot{\alpha} = \dot{\alpha}_0 \cup \dot{\alpha}_1, \quad \mathcal{R}^t_v(\rho) = \dot{\rho}, \\ \mathcal{K}(\Xi, \rho), \quad \mathcal{Q}^k_r(\bar{\rho}, \rho), \quad \mathcal{Q}^k_a(\bar{\alpha}, \alpha), \quad \mathcal{Q}^t_a(\alpha, \dot{\alpha}), \quad id^t_l, id^v_l \in \text{Identifier}, \\ id^t_l \notin \mathrm{dom}(\psi), \quad id^v_l \notin \mathrm{dom}(\theta), \quad this = id^v_l \end{array}}{\mathrm{T}, \kappa, \psi, \theta, \Lambda_0, \mathcal{C}, \Xi \vdash id\ \texttt{\{}\ b\ \texttt{\}} \xrightarrow{\text{Connector}} \psi[id^t_l : (id\ (\alpha, \rho))], \theta[id^v_l : (id^t_l\ (\dot{\alpha}, \dot{\rho}))], \Lambda_1} \tag{B.105}$$

$$\frac{}{\mathrm{T}, \psi, \theta, \alpha, \dot{\alpha}, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash t \xrightarrow{\text{Connector-Body}} \theta, \dot{\alpha}, \rho, \Lambda} \tag{B.106}$$

$$\frac{\begin{array}{c} t_1, t_2 \in \text{Connector-Body}, \quad \mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}_0, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t_1 \xrightarrow{\text{Connector-Body}} \theta_1, \dot{\alpha}_1, \rho_1, \Lambda_1, \\ \mathrm{T}, \psi, \theta_1, \alpha, \dot{\alpha}_1, \bar{\rho}, \rho_1, \Lambda_1, \mathcal{C} \vdash t_2 \xrightarrow{\text{Connector-Body}} \theta_2, \dot{\alpha}_2, \rho_2, \Lambda_2 \end{array}}{\mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}_0, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t_1 \texttt{;} t_2 \xrightarrow{\text{Connector-Body}} \theta_2, \dot{\alpha}_2, \rho_2, \Lambda_2} \tag{B.107}$$

$$\frac{t \in \text{Attribute-Valuation}, \quad \mathrm{T}, \alpha, \dot{\alpha}_0, \mathcal{C} \vdash a \xrightarrow{\text{Attribute-Valuation}} \dot{\alpha}_1}{\mathrm{T}, \psi, \theta, \alpha, \dot{\alpha}_0, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash t \xrightarrow{\text{Connector-Body}} \theta, \dot{\alpha}_1, \rho, \Lambda} \tag{B.108}$$

$$\frac{\mathrm{T}, \psi, \theta_0, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t \xrightarrow{\text{Binding}} \theta_1, \rho_1, \Lambda_1}{\mathrm{T}, \psi, \theta_0, \alpha, \dot{\alpha}, \bar{\rho}, \rho_0, \Lambda_0, \mathcal{C} \vdash t \in\xrightarrow{\text{Connector-Body}} \theta_1, \dot{\alpha}, \rho_1, \Lambda_1} \tag{B.109}$$

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (\mathbf{r}\ (\texttt{uses}, id_k, q_1, q_2)) \in \bar{\rho}, \quad l \in \text{Port-list}, \\ \texttt{provides}, \theta \vdash l \xrightarrow{\text{Port-list}} \vec{p}, id_i, \quad id_i : (id_k\ \alpha) \in \psi, \quad id^t_r \in \text{Identifier}, \quad id^t_r \notin \mathrm{dom}(\rho) \end{array}}{\mathrm{T}, \psi, \theta, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash id\ \texttt{=}\ l \xrightarrow{\text{Binding}} \theta, \rho[id^t_r : (id\ (\texttt{uses}, id_i, q_2))], \Lambda[this.id^t_r : \vec{p}]} \tag{B.110}$$

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (\mathbf{r}\ (\texttt{provides}, id_k, q_1, q_2)) \in \bar{\rho}, \quad l \in \text{Port-list}, \\ \texttt{uses}, \theta \vdash l \xrightarrow{\text{Port-list}} \vec{p}, id_i, \quad id_i : (id_k\ \alpha) \in \psi, \quad \mathcal{A}^t_v(\alpha) = \dot{\alpha}, \quad \mathcal{Q}^t_a(\alpha, \dot{\alpha}), \\ id^t_r \in \text{Identifier}, \quad id^t_r \notin \mathrm{dom}(\rho), \quad this.id^t_r \notin \mathrm{dom}(\theta) \end{array}}{\begin{array}{c} \mathrm{T}, \psi, \theta, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash id\ \texttt{=}\ l \xrightarrow{\text{Binding}} \theta[this.id^t_r : (id_i\ \dot{\alpha})], \\ \rho[id^t_r : (id\ (\texttt{provides}, id_i, q_2))], \Lambda[this.id^t_r : \vec{p}] \end{array}}$$

$$\frac{\begin{array}{c} id \in \text{Identifier}, \quad id : (\mathbf{r}\ (\texttt{provides}, id_k, q_1, q_2)) \in \bar{\rho}, \\ l \in \text{Port-list}, \quad \texttt{uses}, \theta \vdash l \xrightarrow{\text{Port-list}} \vec{p}, id_i, \\ id_i : (id_k\ \alpha) \in \psi, \quad \mathcal{A}^t_v(\alpha) = \dot{\alpha}_0, \quad a \in \text{Attributes}, \quad \mathrm{T}, \alpha, \emptyset, \mathcal{C} \vdash a \xrightarrow{\text{Attributes}} \dot{\alpha}_1, \\ \dot{\alpha} = \dot{\alpha}_0 \cup \dot{\alpha}_1, \quad \mathcal{Q}^t_a(\alpha, \dot{\alpha}), \quad id^t_r \in \text{Identifier}, \quad id^t_r \notin \mathrm{dom}(\rho), \quad this.id^t_r \notin \mathrm{dom}(\theta) \end{array}}{\begin{array}{c} \mathrm{T}, \psi, \theta, \bar{\rho}, \rho, \Lambda, \mathcal{C} \vdash id\ \texttt{\{}\ a\ \texttt{\}}\ \texttt{=}\ l \xrightarrow{\text{Binding}} \theta[this.id^t_r : (id_i\ \dot{\alpha})], \\ \rho[id^t_r : (id\ (\texttt{provides}, id_i, q_2))], \Lambda[this.id^t_r : \vec{p}] \end{array}} \tag{B.111}$$

$$\frac{p \in \text{Port}, \quad c, \theta \vdash p \xrightarrow{\text{Port}} (id_c, id_p), id_i}{c, \theta \vdash p \xrightarrow{\text{Port-list}} \{(id_c, id_p)\}, id_i} \tag{B.112}$$

$$\frac{p \in \text{Port}, \quad c, \theta \vdash p \xrightarrow{\text{Port}} (id_c, id_p), id_i, \quad l \in \text{Port-list}, \quad c, \theta \vdash l \xrightarrow{\text{Port-list}} \vec{p}, id_i}{c, \theta \vdash p \texttt{,} l \xrightarrow{\text{Port-list}} \vec{p}[(id_c, id_p)], id_i} \tag{B.113}$$

$$\frac{id_1, id_2 \in \textsf{Identifier}, \quad id_1 : (id_c\ (\dot{\alpha}, \dot{\pi})) \in \theta, \quad id_2 : (id_p\ (c, id_i, q)) \in \dot{\pi}}{c, \theta \vdash id_1\ \mathbf{.}\ id_2 \xrightarrow{\textsf{Port}} (id_1, id_2), id_i} \tag{B.114}$$

$$\frac{\begin{array}{c} id \in \textsf{Identifier}, \quad id \notin \mathrm{dom}(\mathcal{C}), \quad t \in \textsf{Type-Spec}, \quad \mathrm{T} \vdash t \xrightarrow{\textsf{Type-Spec}} \tau, \\ v \in \textsf{Literal}, \quad \mathrm{T}, \tau, \mathcal{C} \vdash v \xrightarrow{\textsf{Literal}} \dot{\tau} \end{array}}{\mathrm{T}, \mathcal{C} \vdash id\ \mathbf{:}\ t = v \xrightarrow{\textsf{Constant}} \mathcal{C}[id : \dot{\tau}]} \tag{B.115}$$

APPENDIX C

ADDITIONAL SOURCE LISTINGS

C.1 NESC SOURCES

C.1.1 SURGE.NC

```
// $Id: Surge.nc,v 1.1 2007/06/25 19:11:47 jung Exp $

/*                                                                  tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */
/**
 *
 **/
includes Surge;
includes SurgeCmd;
includes MultiHop;


configuration Surge {
}
implementation {
  components Main, SurgeM, TimerC, LedsC, NoLeds, Photo, RandomLFSR,
    GenericCommPromiscuous as Comm, Bcast, MultiHopRouter as multihopM, QueuedSend, Sounder;

  Main.StdControl -> SurgeM.StdControl;
  Main.StdControl -> Photo;
  Main.StdControl -> Bcast.StdControl;
  Main.StdControl -> multihopM.StdControl;
  Main.StdControl -> QueuedSend.StdControl;
  Main.StdControl -> TimerC;
  Main.StdControl -> Comm;
  //  multihopM.CommControl -> Comm;

  SurgeM.ADC   -> Photo;
  SurgeM.Timer -> TimerC.Timer[unique("Timer")];
  SurgeM.Leds  -> LedsC; // NoLeds;
  SurgeM.Sounder -> Sounder;

  SurgeM.Bcast -> Bcast.Receive[AM_SURGECMDMSG];
  Bcast.ReceiveMsg[AM_SURGECMDMSG] -> Comm.ReceiveMsg[AM_SURGECMDMSG];

  SurgeM.RouteControl -> multihopM;
  SurgeM.Send -> multihopM.Send[AM_SURGEMSG];
  multihopM.ReceiveMsg[AM_SURGEMSG] -> Comm.ReceiveMsg[AM_SURGEMSG];
  //multihopM.ReceiveMsg[AM_MULTIHOPMSG] -> Comm.ReceiveMsg[AM_MULTIHOPMSG];
}
```

C.1.2 GenericCommPromiscuous.nc

```
// $Id: GenericCommPromiscuous.nc,v 1.1 2007/07/06 21:12:15 jung Exp $

/*                                                                      tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */
/*
 *
 * Authors:             Jason Hill, David Gay, Philip Levis
 * Date last modified:  $Id: GenericCommPromiscuous.nc,v 1.1 2007/07/06 21:12:15 jung Exp $
 *
 */

/**
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
 */


configuration GenericCommPromiscuous
{
  provides {
    interface StdControl as Control;
    interface CommControl;

    // The interface are as parameterised by the active message id
    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];

    // How many packets were received in the past second
    command uint16_t activity();

  }
  uses {
    // signaled after every send completion for components which wish to
    // retry failed sends
    event result_t sendDone();


  }
}
implementation
{
  // CRCPacket should be multiply instantiable. As it is, I have to use
  // RadioCRCPacket for the radio, and UARTNoCRCPacket for the UART to
  // avoid conflicting components of CRCPacket.
  components AMPromiscuous as AM,
    RadioCRCPacket as RadioPacket,
    UARTFramedPacket as UARTPacket,
    NoLeds as Leds,
    TimerC, HPLPowerManagementM;

  Control = AM.Control;
  CommControl = AM.CommControl;
  SendMsg = AM.SendMsg;
  ReceiveMsg = AM.ReceiveMsg;
  sendDone = AM.sendDone;

  activity = AM.activity;
  AM.TimerControl -> TimerC.StdControl;
  AM.ActivityTimer -> TimerC.Timer[unique("Timer")];

  AM.UARTControl -> UARTPacket.Control;
  AM.UARTSend -> UARTPacket.Send;
  AM.UARTReceive -> UARTPacket.Receive;

  AM.RadioControl -> RadioPacket.Control;
  AM.RadioSend -> RadioPacket.Send;
  AM.RadioReceive -> RadioPacket.Receive;
  AM.PowerManagement -> HPLPowerManagementM.PowerManagement;

  AM.Leds -> Leds;
}
```

C.1.3 TimerC.nc

```
// $Id: TimerC.nc,v 1.1 2007/07/06 21:12:15 jung Exp $

/*                                                                    tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */
/*
 * Authors:  Su Ping,  (converted to nesC by Sam Madden)
 *           David Gay,      Intel Research Berkeley Lab
 *           Phil Levis
 * Date:     4/12/2002
 * NesC conversion: 6/28/2002
 * interface cleanup: 7/16/2002
 * Configuration:     8/12/2002
 */

/**
 * @author Su Ping
 * @author (converted to nesC by Sam Madden)
 * @author David Gay
 * @author Intel Research Berkeley Lab
 * @author Phil Levis
 */


configuration TimerC {
  provides interface Timer[uint8_t id];
  provides interface StdControl;
}

implementation {
  components TimerM, ClockC, NoLeds, HPLPowerManagementM;

  TimerM.Leds -> NoLeds;
  TimerM.Clock -> ClockC;
  TimerM.PowerManagement -> HPLPowerManagementM;

  StdControl = TimerM;
  Timer = TimerM;
}
```

C.2 Additional CALM Sources

C.2.1 Some CORBA data types

Listing C.4: Some CORBA data types

```
attributes CorbaTypes {

  // Unspecified Types
  typedecl any;

  typedecl native;


  // Boolean type
  typedef BOOLEAN boolean;


  // IDL character, CDR 8-bit encoding,
  // GIOP codeset conversion:
  // Illustrates need for plugin editor.
  typedef char = INT[0..255];


  // IEEE standard single- and double- precision
  // floating point number
  // Illustrates need for plugin editor.
  // Faithful representation of the IEEE standard
  // might not be required for most cases,
  // abstractions are possible.
  typedef NonNumerical = ENUM {
    Zero, PosInf, NegInf, QuietNaN, SignallingNaN };

  typedef SingleNormalized = struct {
    Sign     : INT[0..1],
    Exponent : INT[-126..127],
    Fraction : INT[8388608..16777216] };

  typedef SingleDenormalized = struct {
    Sign     : INT[0..1],
    Exponent : INT[-126],
    Fraction : INT[0..8388608] };

  typedef DoubleNormalized = struct {
    Sign     : INT[0..1],
    Exponent : INT[-1022..1023],
    Fraction : INT[4503599627370496..9007199254740992] };

...
```

```
  typedef float = union {
    SingleNormalized,
    SingleDenormalized,
    NonNumerical };

  typedef double = union {
    DoubleNormalized,
    DoubleDenormalized,
    NonNumerical };

  typedef long_double {
    LongDoubleNormalized,
    LongDoubleDenormalized,
    NonNumerical };
```

...

```
  // Exception Types
  typedef Member = struct {
    name         : STRING,
    platformType : type };

  typedef exception = struct {
    ExcType : STRING,
    Members : Member bag };


  // The fixed type
  typedef fixed = struct {
    Digits : INT[1..31],
    Scale  : INT[0..*] };


  // short, long, long long
  typedef short = INT[-132768..32767];             // 16
  typedef unsigned_short = INT[0..65536];          // 16

  typedef long = INT[-2147483648..2147483647];     // 32
  typedef unsigned_long = INT[0..4294967296];      // 32

  typedef long_long = INT[-9223372036854775808.. 9223372036854775807]; // 64
  typedef unsigned_long_long = INT[0..18446744073709551616];           // 64
```

...

C.2.2 The Robot Model module

Listing C.7: The Robot Model module

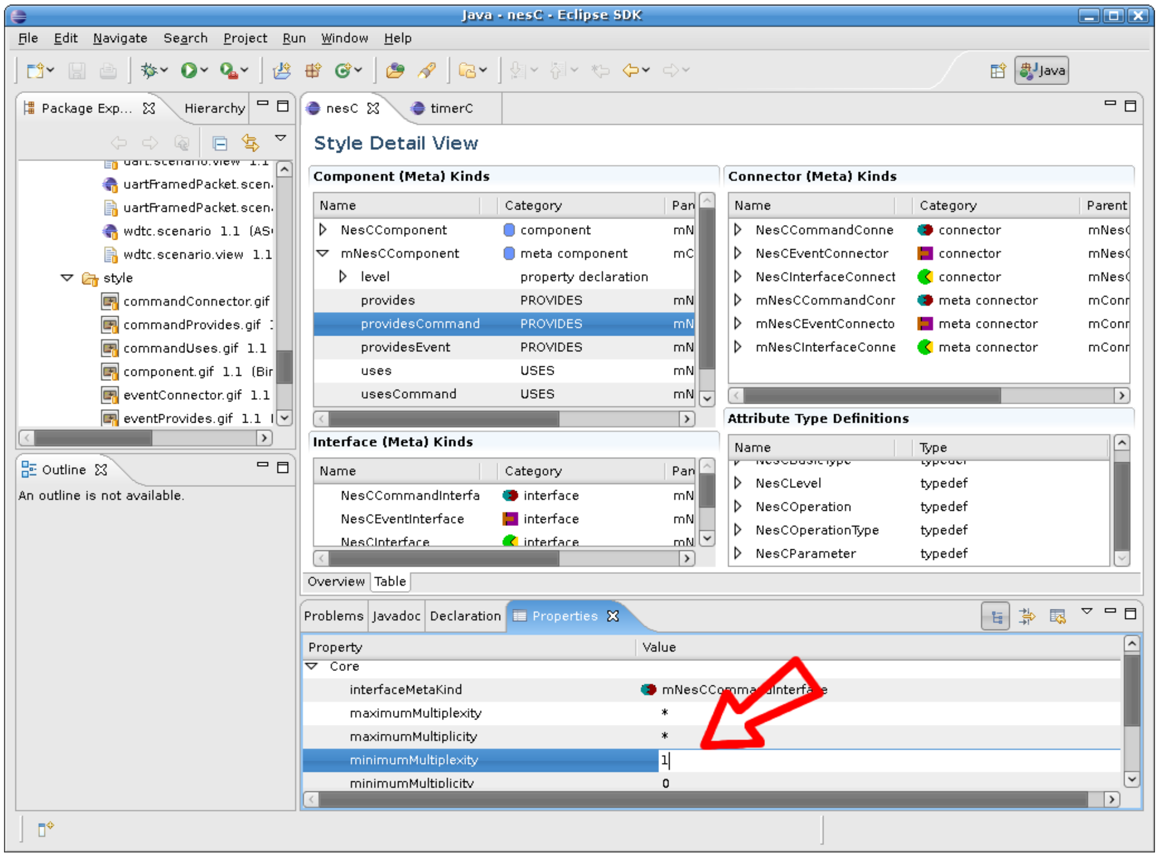
```
module robot of ccm {

  CCMEvent AlarmControl {};
  CCMEvent DescreteStatus {};
  CCMEvent DescreteUpdate {};
  CCMEvent DisplayAlert {};
  CCMEvent DisplayWork {};
  CCMEvent EquipmentControl {};
  CCMEvent EquipmentStatus {};
  CCMEvent IntrusionStatusReport {};
  CCMEvent IntrusionStatusReportManagement {};
  CCMEvent MovePalletRequest {};
  CCMEvent PalletProcessingStatus {};
  CCMEvent PalletStatusResponse {};
  CCMEvent PrepareToShutdown {};
  CCMEvent ProcessPallet {};
  CCMEvent ProductionStatus {};
  CCMEvent ProductionWorkOrder {};
  CCMEvent RobotControlMessage {};
  CCMEvent RobotStatusMessage {};
  CCMEvent Shutdown {};
  CCMEvent ShutdownStatus {};
  CCMEvent switchChanged {};
  CCMEvent SwitchStatus {};
  CCMEvent TimeNotification {};

  CCMInterface AlarmController {};
  CCMInterface ClockControl {};
  CCMInterface conveyorDriveControl {};
CCMInterface DiscreteControl {};
CCMInterface HMIController {};
CCMInterface MWIController {};
CCMInterface PCMAnalysis {};
CCMInterface PCMController {};
CCMInterface PowerControl {};
CCMInterface RadioFrequencyNeededResponse {};
CCMInterface RMAnalysis {};
CCMInterface RMController {};
CCMInterface RobotControl {};
CCMInterface RobotMovement {};
CCMInterface SwitchControl {};
CCMInterface TextWindows {};
CCMInterface WorkOrderResponses {};
CCMInterface WSMAnalysisCalls {};
CCMInterface WSMAnalysisOne {};
CCMInterface WSMAnalysisTwo {};
CCMInterface WSMController {};

CCMComponent AlarmHandler {
  consumes AlarmVolume  : AlarmControl;
  provides ControlAlarm : AlarmController };
CCMComponent ClockHandler {
  consumes  ShutdownOrder    : Shutdown;
  consumes  ShutdownWarning : PrepareToShutdown;
  emits     ShutdownResponse : ShutdownStatus;
  provides  control : ClockControl;
  publishes currentTime : TimeNotification };
CCMComponent Communications {
  consumes ProductionReport : ProductionStatus;
  consumes ShutdownOrder    : Shutdown;
  consumes ShutdownWarning  : PrepareToShutdown;
  consumes UrgentReport     : IntrusionStatusReportManagement;
  emits    ShutdownResponse : ShutdownStatus;
  emits    WorkOrder        : ProductionWorkOrder;
  provides Controller : MWIController };
CCMComponent ConvoyerDriveHandler {
  provides ConvoyerControl : conveyorDriveControl };
CCMComponent DiscreteController {
  emits    status : DescreteStatus;
  provides Control : DiscreteControl };
CCMComponent GraphicalUserInterface {
  provides Windows : TextWindows;
  uses     Responses : WorkOrderResponses };
CCMComponent PalletConveyor {
  consumes PalletRequests  : MovePalletRequest;
  consumes ShutdownOrder   : Shutdown;
  consumes ShutdownWarning : PrepareToShutdown;
  consumes update          : DescreteUpdate;
  emits    Alert            : DisplayAlert;
  emits    IntrusionReport  : IntrusionStatusReport;
  emits    PalletStatus     : PalletStatusResponse;
  emits    ShutdownResponse : ShutdownStatus;
  emits    SoundAlarm       : AlarmControl;
  provides CircleAnalysis : PCMAnalysis;
  provides Controller     : PCMController;
  uses     AnalysisTwo     : WSMAnalysisTwo;
  uses     control         : PowerControl;
  uses     convoyerControl : conveyorDriveControl };
CCMComponent PowerSwitchingHandle {
  provides Control : PowerControl };
CCMComponent ProductionController {
  consumes  IntrusionReport           : IntrusionStatusReport;
  consumes  PalletStatus              : PalletStatusResponse;
  consumes  ProcessingStatus          : PalletProcessingStatus;
  consumes  ProductionEquipmentStatus : EquipmentStatus;
  consumes  ShutdownReport            : ShutdownStatus;
  consumes  WorkOrder                 : ProductionWorkOrder;
  emits     Alert             : DisplayAlert;
  emits     Display           : DisplayWork;
  emits     MovePallet        : MovePalletRequest;
  emits     ProductionControl : ProcessPallet;
  emits     ProductionReport  : ProductionStatus;
  emits     Reports           : IntrusionStatusReportManagement;
  emits     SoundAlarm        : AlarmControl;
  provides  AnalysisCalls   : WSMAnalysisCalls;
  provides  AnalysisOne     : WSMAnalysisOne;
  provides  AnalysisTwo     : WSMAnalysisTwo;
  provides  Controller      : WSMController;
  provides  DisplayResponse : WorkOrderResponses;
  publishes ProductionEquipmentControl : EquipmentControl;
  publishes ShutdownOrder              : Shutdown;
```

```
    publishes ShutdownWarning            : PrepareToShutdown;
    uses      Analysis : RMAnalysis };
  CCMComponent RobotHardware {
    provides control : RobotControl };
  CCMComponent RobotMovementControl {
    consumes  ControlRobot : RobotControlMessage;
    provides  Movement : RobotMovement;
    publishes RobotStatus : RobotStatusMessage;
    uses      control : RobotControl };
  CCMComponent RobotWatchProduction {
    consumes  currentTime                : TimeNotification;
    consumes  ProcessPalletCommands      : ProcessPallet;
    consumes  ProductionEquipmentControl : EquipmentControl;
    consumes  RobotStatus                : RobotStatusMessage;
    consumes  ShutdownOrder              : Shutdown;
    consumes  ShutdownWarning            : PrepareToShutdown;
    emits     Display                   : DisplayWork;
    emits     ProcessingStatus          : PalletProcessingStatus;
    emits     ProductionEquipmentStatus : EquipmentStatus;
    emits     ShutdownResponse          : ShutdownStatus;
    provides  Analysis       : RMAnalysis;
    provides  Controller     : RMController;
    provides  RadioFrequency : RadioFrequencyNeededResponse;
    publishes ControlRobot : RobotControlMessage;
    uses      AnalysisOne    : WSMAnalysisOne;
    uses      CircleAnalysis : PCMAnalysis;
    uses      Movement       : RobotMovement;
    uses      RobotID        : DiscreteControl };
  CCMComponent SwitchController {
    consumes  status : SwitchStatus;
    provides  Control : SwitchControl;
    publishes switchChanges : switchChanged };
  CCMComponent SwitchDriver {
    publishes status : SwitchStatus };
  CCMComponent WatchProductionHMI {
    consumes Alert             : DisplayAlert;
    consumes ShutdownOrder     : Shutdown;
    consumes ShutdownWarning   : PrepareToShutdown;
    consumes switchChanges     : switchChanged;
    consumes WorkDisplayUpdate : DisplayWork;
    emits    ShutdownResponse : ShutdownStatus;
    provides Controller : HMIController;
    provides responses  : WorkOrderResponses;
    uses     HumanResponse : WorkOrderResponses;
    uses     RadioResponse : RadioFrequencyNeededResponse;
    uses     windowing     : TextWindows };
}
```

APPENDIX D

ADDITIONAL CADENA SCREENSHOTS

D.1 IMMEDIATE PROBLEM REPORT


Java - nesC - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Package Exp... Hierarchy
nesC timerC
Outline
Overview Table
Problems Javadoc Declaration Properties

uart.scenario.view 1.1
uartFramedPacket.scen
uartFramedPacket.scen
wdtc.scenario 1.1 (AS
wdtc.scenario.view 1.1
style
commandConnector.gif
commandProvides.gif
commandUses.gif 1.1
component.gif 1.1 (Bir
eventConnector.gif 1.1
eventProvides.gif 1.1

An outline is not available.

Style Detail View

Component (Meta) Kinds

Name	Category	Par
NesCComponent	component	mN
mNesCComponent	meta component	mC
level	property declaration	
provides	PROVIDES	mN
providesCommand	PROVIDES	mN
providesEvent	PROVIDES	mN
uses	USES	mN
usesCommand	USES	mN

Connector (Meta) Kinds

Name	Category	Parent
NesCCommandConne	connector	mNes
NesCEventConnector	connector	mNes
NesCInterfaceConnect	connector	mNes
mNesCCommandConr	meta connector	mConr
mNesCEventConnecto	meta connector	mConr
mNesCInterfaceConne	meta connector	mConr

Interface (Meta) Kinds

Name	Category	Par
NesCCommandInterfa	interface	mN
NesCEventInterface	interface	mN
NesCInterface	interface	mN

Attribute Type Definitions

Name	Type
NesCBasicType	typedef
NesCLevel	typedef
NesCOperation	typedef
NesCOperationType	typedef
NesCParameter	typedef

Property	Value
Core	
interfaceMetaKind	mNesCCommandInterface
maximumMultiplexity	*
maximumMultiplicity	*
minimumMultiplexity	1
minimumMultiplicity	0

Figure D.1: Changing the multiplexity of the providesComponent port option in CADENA

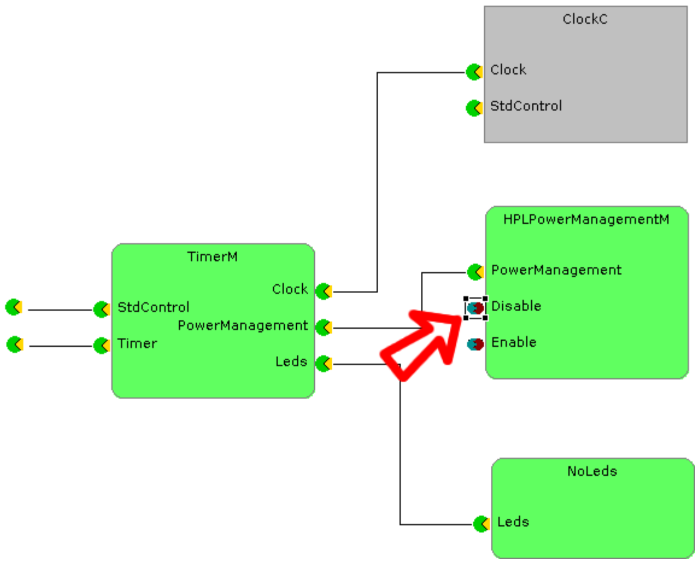
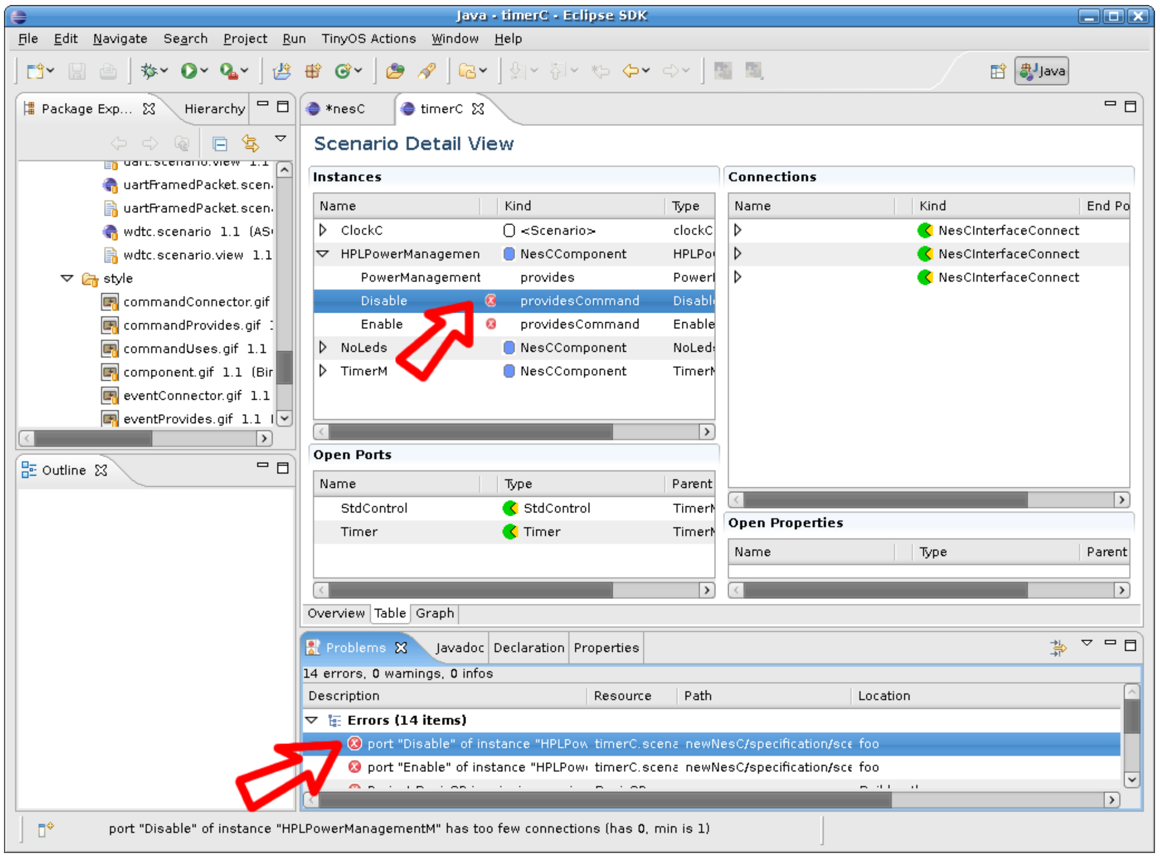


Figure D.2: Effect of a changed multiplexity, affected scenario in CADENA


Java - timerC - Eclipse SDK
File Edit Navigate Search Project Run TinyOS Actions Window Help
Java
Package Exp... Hierarchy
*nesC timerC
Outline
Overview Table Graph
Problems Javadoc Declaration Properties

uart.scenario.view 1.1
uartFramedPacket.scen
uartFramedPacket.scen
wdtc.scenario 1.1 (AS
wdtc.scenario.view 1.1
style
commandConnector.gif
commandProvides.gif
commandUses.gif 1.1
component.gif 1.1 (Bir
eventConnector.gif 1.1
eventProvides.gif 1.1

Scenario Detail View

Instances

Name	Kind	Type
ClockC	<Scenario>	clockC
HPLPowerManagemen	NesCComponent	HPLPo
PowerManagement	provides	Powerl
Disable	providesCommand	Disabl
Enable	providesCommand	Enable
NoLeds	NesCComponent	NoLed
TimerM	NesCComponent	Timerl

Connections

Name	Kind	End Po
	NesCInterfaceConnect	
	NesCInterfaceConnect	
	NesCInterfaceConnect	

Open Ports

Name	Type	Parent
StdControl	StdControl	Timerl
Timer	Timer	Timerl

Open Properties

Name	Type	Parent

14 errors, 0 warnings, 0 infos

Description	Resource	Path	Location
Errors (14 items)			
port "Disable" of instance "HPLPow	timerC.scena	newNesC/specification/sce	foo
port "Enable" of instance "HPLPow	timerC.scena	newNesC/specification/sce	foo

port "Disable" of instance "HPLPowerManagementM" has too few connections (has 0, min is 1)

Figure D.3: Effect of a changed multiplexity in the CADENA scenario form editor