

# Automated Resolution of Connector Architectures Using Constraint Solving (ARCAS method)

JAROSLAV KEZNIKL  
TOMÁŠ BUREŠ  
FRANTIŠEK PLÁŠIL  
PETR HNĚTYNKA

Charles University in Prague, Faculty of Mathematics and Physics  
Malostranske namesti 25, 118 00 Prague 1, Czech Republic  
{keznikl, bures, plasil, hnetynka}@d3s.mff.cuni.cz  
<http://d3s.mff.cuni.cz>

**Abstract:** In current software systems, connectors play an important role by encapsulating the communication and coordination logic. Since they share common patterns (elements) depending on characteristics of the connections, the elements can be predefined and reused. A method of connector implementation based on a composition of predefined elements naturally comprises two steps – resolution of the connector architecture, and creation of the actual connector code based on the architecture. However, manual resolution of a connector architecture is not feasible due to the number of factors to be considered. Thus, the challenge is to come up with an automated method, able to address all the important factors. In this paper, we present a method for automated resolution of connector architectures based on constraint solving techniques (ARCAS). We exploit the Alloy modeling language for defining a connector theory, reflecting both the predefined parts and the important resolution factors, and employ a constraint solver to find a suitable connector architecture as a model of the theory.

**Keywords:** Software Architecture, Software Connectors, Constraint Solving, Middleware-based Connectors, Connector Theory, Alloy

## 1. INTRODUCTION

Proposed with the aim of supporting the separation of concerns, software connectors [MMP00, TMD10] are entities solely encapsulating communication and coordination among components. In particular, connectors ensure distribution of communicating components [BP04] while encapsulating middleware (*middleware-based connectors*), provide adaptation in order to achieve middleware-level [IBB11, NTER06] and application-level [SI10, CCP11, IST11] interoperability (*adaptors*), and ensure synchronization of component communication [BS07, IST11] (*coordinators*). In this paper, we focus particularly on the middleware-based connectors.

Although the introduction of middleware-based connectors provides benefits in terms of separation of concerns and abstraction of particular middleware, it does not necessarily simplify the code development effort, since, in principle, the middleware-related code is moved from components to connectors. In the component models that include connectors, e.g., [RCGT09, TMD10], the connector lifecycle differs from the component lifecycle: although partially-specified connectors are employed during the application design phase, fully-specified connectors emerge at the earliest in the component deployment phase – after decisions on application architecture and deployment have been made. Moreover, deployment of a particular component application may vary from time to time, so that several variants of a connector may be required. Advantageously, these variants typically share common patterns related to a particular communication style [BP04, TMD10, IBB11], middleware, and non-functional properties (NFPs); therefore, related parts of connectors can be predefined/designed in advance.

Middleware-based connectors can also emerge later, even after some of the components are already running. This is particularly true in the case of independently deployed components available as services (e.g., web services). In principle, the task of a newly emerging connector is to mediate a client component's communication with such a service while respecting the particular middleware employed by the service. In this sense, services are middleware-aware components. Similarly, the emergence of such connectors can be desirable at runtime once architecture and deployment reconfiguration takes place (e.g., due to load balancing).

In this context, the challenge is to find an automated method for synthesis of middleware-based connectors at deployment time/run time, i.e., synthesis of *emergent connectors* [ISJB09], in such a way, that reuse of predefined connector parts is maximized. A related issue is to structure the predefined parts accordingly. Another challenge is to support NFPs in the actual connector synthesis and to structure the predefined parts in order to efficiently and flexibly capture variability of NFP requirements. The context of such automated connector synthesis in the design, deployment, and runtime phases of an application is illustrated in Fig. 1.

### 1.1 Goal of the Paper

The goal of this paper is to respond to the challenges above by introducing the ARCAS method (Automated Resolution of Connector Architectures using a constraint-Solving technique). In general, ARCAS is based on an automated composition of connectors from predefined hierarchical *elements* [BP04, RCGT09]. It produces a description of a hierarchical composition of elements reflecting the connector design and deployment requirements imposed on the components being connected.

Technically, given a design specification of a connector, including NFPs and decision on component deployment, ARCAS produces a connector instance configuration (CIC), describing a particular composition of available elements to realize the connector. From a big-picture perspective, ARCAS is the first phase in a two-step connector generation process [B06]. In the second step, CIC is used as the input for the actual code generation process [MPBH11] yielding a deployable connector code.

The basic idea of ARCAS (Fig. 2) is to employ a constraint-solving technique for automated resolution of CIC. For this purpose, we employ the Alloy modeling language [J02, J06, J11] for capturing a *connector theory*, i.e., a logic theory playing the role of a constraint specification reflecting both the predefined elements, and the

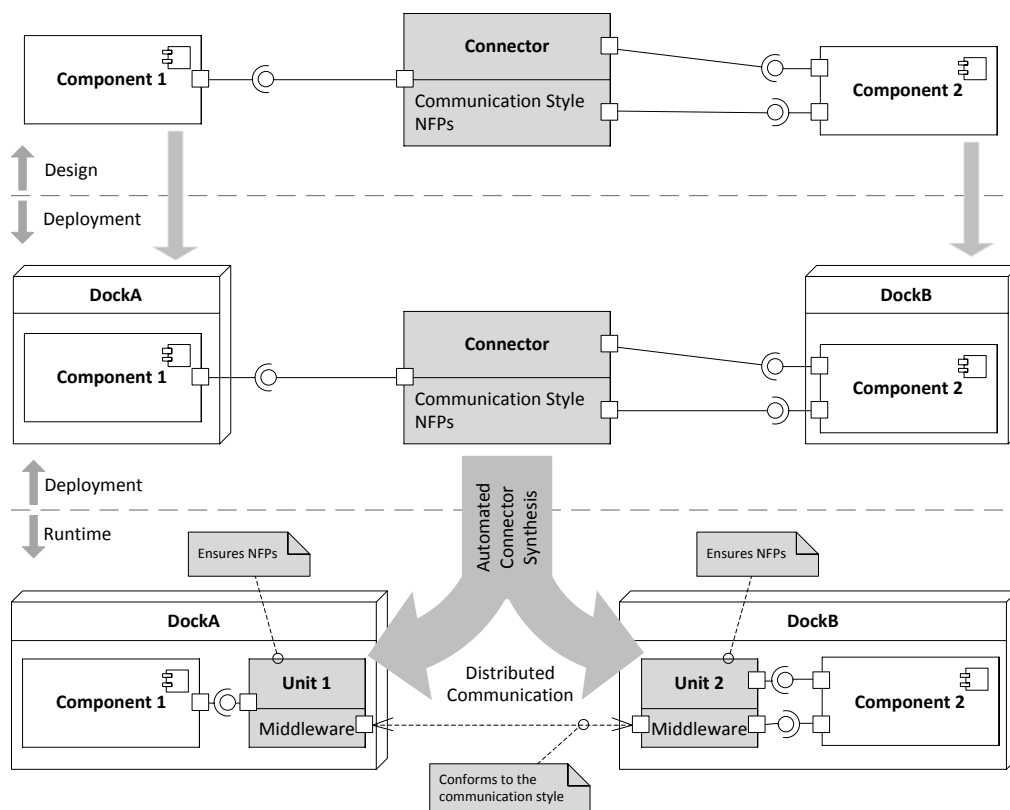


Fig. 1 Role of automated connector synthesis in connector lifecycle

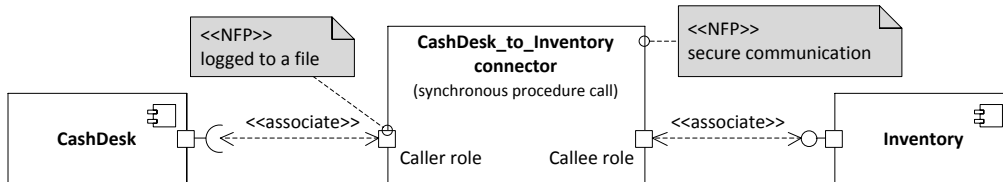


Fig. 3 Example of the application-designer perspective – requirements view

design and deployment requirements of the connector. Further, we employ the Alloy Analyzer as a constraint solver to find a model of the theory, representing a desired CIC. As an aside, the ARCAS method has applicability also in other domains as discussed further (Section 8).

## 1.2 Structure of the Paper

The paper is structured as follows: Section 2 presents the basic concepts of middleware connectors and illustrates these with an example. Section 3 gives a brief overview of the whole ARCAS method. Section 4 describes both abstract and concrete syntax of a middleware-connector specification. Section 5 describes the construction of a connector theory in terms of predicate logic and relational calculus. Section 6 provides a brief introduction to the Alloy modeling language and describes ARCAS in terms of Alloy. Section 7 surveys the related work, whereas Section 8 provides evaluation and discussion of the ARCAS method, and Section 9 concludes the paper while suggesting future work activities.

## 2. MIDDLEWARE CONNECTORS – BASIC CONCEPTS

In this section, we introduce the basic concepts of middleware connectors. A connector can be viewed from: (i) application designer perspective (*requirements view* and *deployment view*) and (ii) connector designer perspective (*design view*). While (i) focuses on the high-level task and properties of a connector in a particular component application, (ii) focuses strictly on the connector design and implementation. Here, the concepts in (i) are generally agreed in the area of middleware-based connectors [CL02], whereas the concepts in (ii) stem from our experience with connector design and implementation [BP04] and are thus rather specific to ARCAS. We recall and illustrate all the concepts on a simple example – a fragment of a distributed component-based application [HKW08] featuring the components CashDesk and Inventory bound together by a single connector. This setting will be used as a running example throughout the text.

From the application designer perspective, the requirements view of a connector (Fig. 3) focuses on describing the communication style and the required NFPs (henceforth referred to as *features*) of a component connection. The communication style defines *roles* – the connector’s endpoints for communication of components [CL02]; e.g., the procedure-call communication style defines the roles *Caller* and *Callee*. The components to be connected communicate using instances of the roles. Therefore, the requirements view of a connector has to comprise an association of the

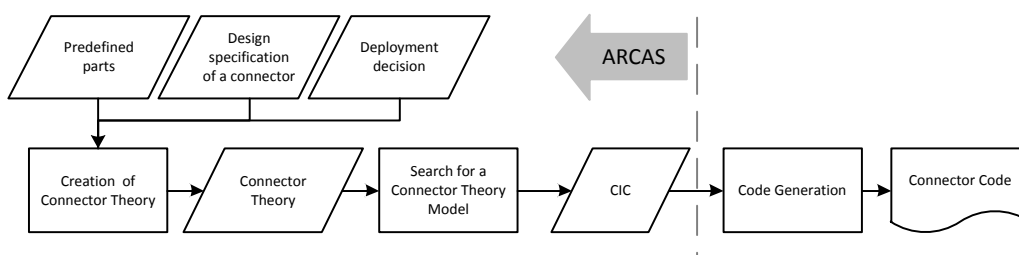


Fig. 2 ARCAS overview

respective component interfaces with particular instances of connector's roles; e.g., the CashDesk's required interface is associated with an instance of the role Caller and the Inventory's provided interface is associated with an instance of the role Callee. Finally, the requirements view of a connector defines the required features of the component connection; e.g., the requirement that all connector invocations have to be logged to a file and that the communication has to be secure. For the purpose of this text, the requirements view of a connector is assumed to be described by a *requirements specification*.

At the deployment time, a connector has to reflect the distribution of the connected components according to the actual deployment decision. This is the focus of the deployment view of a connector (Fig. 4). In the example, the distributed environment consists of two component containers (*deployment docks*) A and B. Thus at the deployment time a connector is viewed as an assembly of distributed connector *units*. The key purpose of a unit is to refine the roles associated with a particular component so that the communication between a unit and the corresponding component is local, whereas communication among units is (typically) remote; for instance, the connector defined in Fig. 4 is split into two units, each related to one of the CashDesk and Inventory components. In compliance with the desired component deployment, units have to conform to the *capabilities* of the selected deployment docks. Deployment dock capabilities are key properties of the execution environment, driving selections of middleware technology for remote communication and are based on the OMG D&C standard [OMG04]. For example, the capabilities of the dock "A" indicate the availability of Java virtual machine version JDK 1.4 and underlying Linux 2.4.28. Thus, the unit for the CashDesk component has to be able to run and communicate in such runtime environment. Information on the actual deployment decision and capabilities of all docks is given in a *deployment specification*.

From the connector designer perspective, the design view of a connector focuses on describing the connector implementation by means of (possibly hierarchical) *elements*. An important idea is that the individual parts of connector implementation are designed in advance and reused (this is supported by the hierarchical structure of the connector design). Thus, all henceforth-introduced concepts facilitate composability and reuse. To allow design in advance, the particular application (i.e., the components and their deployment) is abstracted away by considering just communication style, features, and capabilities (thus, the communication style, features, and capabilities are the binding concepts of the application-designer and connector-designer perspectives). At the top level, the connector is described by its *distribution architecture* (Fig. 5), defining the potential units, i.e., the anticipated distribution of the connector, and the refinement of the roles by the units (e.g., the units For\_Callee and For\_Caller). At this point, it also reflects the units as the top-

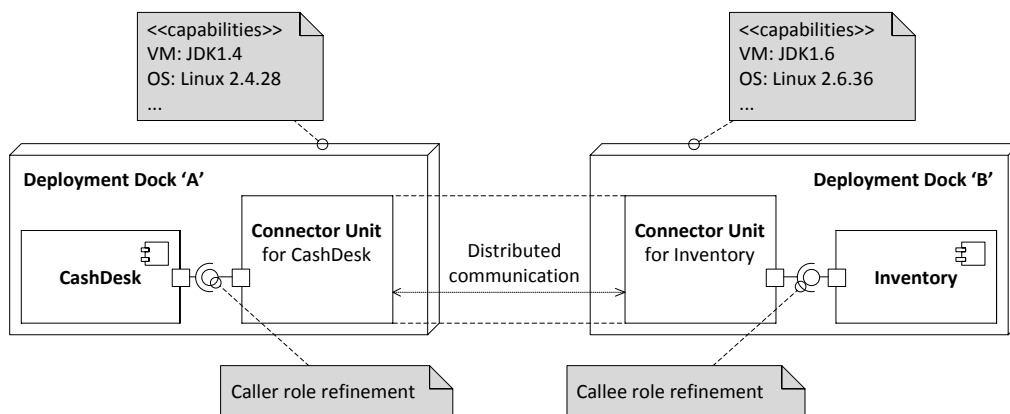


Fig. 4 Example of the application-designer perspective – deployment view

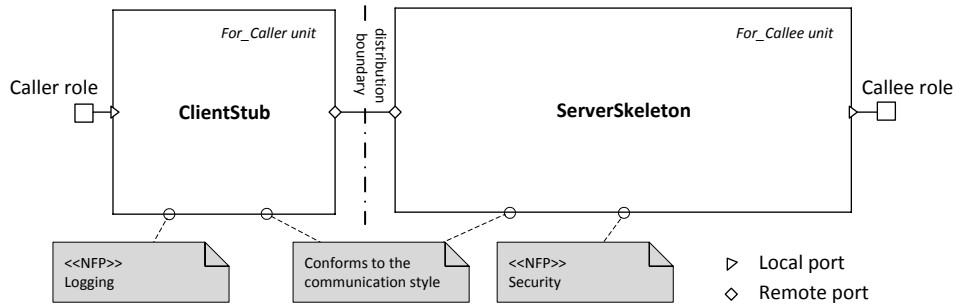


Fig. 5 Example of the distribution-architecture of a connector

level elements defined just by their type (**ClientStub** and **ServerSkeleton**, explained below), as well as the units' interconnections. A distribution architecture also determines how the particular requirements are addressed by individual units. The connector implementation at lower levels of the element hierarchy is defined recursively in such a way that each level of nesting is described by an *element architecture* specifying the horizontal composition of a specific element from its sub-elements (Fig. 6). Here, a sub-element is referred to just by its outer boundary – *element type*, which is to be further refined by an element architecture. This facilitates automated synthesis and reuse of hierarchically composable elements in a way similar to hierarchical components [BP04, CL02, TMD10]. For example, the unit *For\_Callee* is reflected as the **SerializedServerSkeleton** element (of type **ServerSkeleton**); the architecture refining its type introduces two sub-elements – **SocketFactorySkeleton** and **CallSerializer**. In a similar vein, the architecture refining the **SocketFactorySkeleton**'s type defines two sub-elements – **SocketFactoryProvider** and **RMISkeleton**.

Elements (including the top-level ones) interact via *ports*. A port can be either local or remote. A local port (e.g., in **FileLogger**) serves for internal communication of elements not directly participating in the distributed communication (based on local procedure calls). It is specified either as provided or required to emphasize where the communication is initiated in order to allow more precise design of elements, facilitating the automated synthesis. A remote port (e.g., in **RMISub**) serves for distributed communication among units. In this case, the communication depends on the employed middleware. From the connector-designer perspective, the specifics

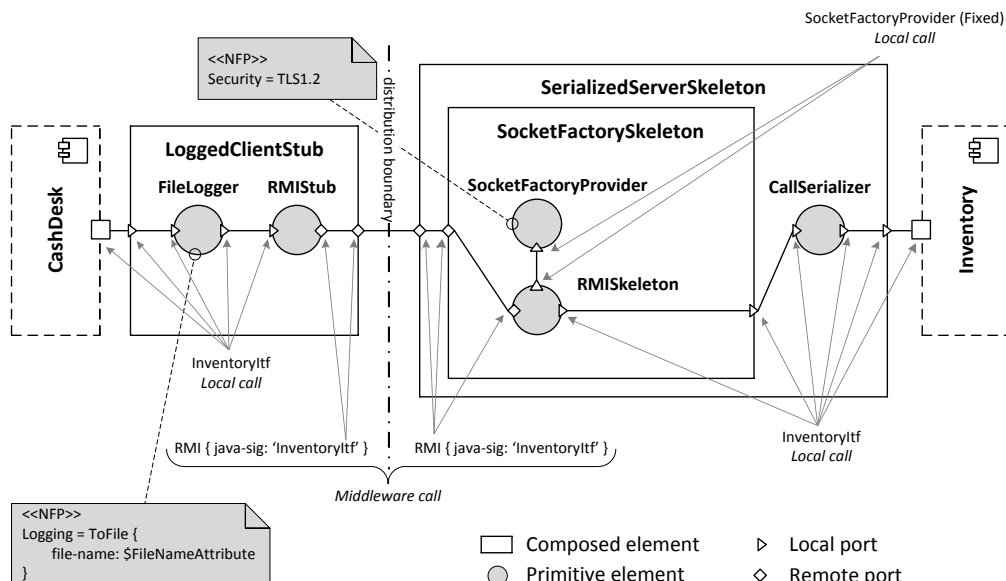


Fig. 6 Example of a full composition of a connector

of the middleware-based communication are intentionally abstracted. Port bindings are captured at the level of the parent element architecture and include both bindings at the same level of nesting (e.g., `FileLogger` and `RMISStub`) and port delegation (e.g., `SocketFactorySkeleton` and `RMISkeleton`).

Eventually, every port is associated with a *signature*, which determines the interface type of the port. The association is either explicitly defined (e.g., the port signature of `SocketFactoryProvider`), or propagated – inferred either from the association of the other element’s ports bound to the current port (implicit propagation), or from the association of the other ports of the same element (signature constraints). In a particular connector implementation, the propagated signatures are typically associated with the signature of a component interface (e.g., `InventoryItf`). Using signature constraints, signature association can be propagated either directly (e.g., `FileLogger`), or as parameters of structured signatures (e.g., the remote interface of `RMISStub`). In case of parameterized signatures, the implicit propagation implies that the parameterization of one signature has to match the parameterization of the other (e.g., `java-sig: ‘inventoryItf’`). This way, element architectures can be independent of the actual signatures (can use the signature propagation only) and are thus generic, which also facilitates reuse.

As for features, by following the idea of element composition, we assume every feature to be addressed by an element can be handled either (i) directly by the element, or (ii) by one of its sub-elements. This allows isolation of features and facilitates the element composition. An example of (i) is the logging feature in the unit `For_Caller`. In this case, the entire task of logging is addressed by `FileLogger`, which intercepts and logs the communication going through the unit. An example of (ii) is the security feature delegated by `SocketFactorySkeleton` to `SocketFactoryProvider`. In general, a part of the feature value may be left unspecified in an element, since it is to be later on determined from requirements specification. This is captured by the element *attribute* concept, i.e., a parameter of the element (e.g., the file name attribute for logging to a file in `FileLogger`).

From the connector designer perspective, the *runtime-environment requirements* of an element architecture are expressed as constraints on dock capabilities. In other words, runtime-environment requirements are abstraction of a particular deployment.

In general, all the concepts featuring in the design view are intended for reuse in multiple applications/connectors. For the purpose of this text, the design view of a connector is assumed to be described by an *artifact specification*.

## 2.1 Summary of the concepts and problem refinement

For the purpose of this paper, we will describe a particular connector instance using the sets and relations in Fig. 7, which shows key parts of the connector instance meta-model. In the rest of this paper, the description of a particular connector instance will be referred to as a connector instance configuration (CIC).

Specifically, all the key CIC concepts, i.e., connector, role, role instance, unit, element, sub-element, port, signature, feature, connector feature, attribute, and deployment dock, are represented explicitly via sets. However, the concepts of element type, communication style, and dock capabilities are not reflected explicitly, since they are rather a part of a design specification, being in a CIC “blended in” already. Further, the meta-model captures the key relations which characterize a particular connector instance and its elements.

A particular connector (its CIC) is therefore an instance of the meta-model. The instance is determined by (a) an artifact specification (predefined), and by (b) requirements and deployment specifications (application-specific).

However, given both (a) and (b), while some of the sets and relations in the meta-model are fully determined (solid line), some are underspecified (dashed line), and therefore determined only partially (*alterable sets and relations*). In other words,

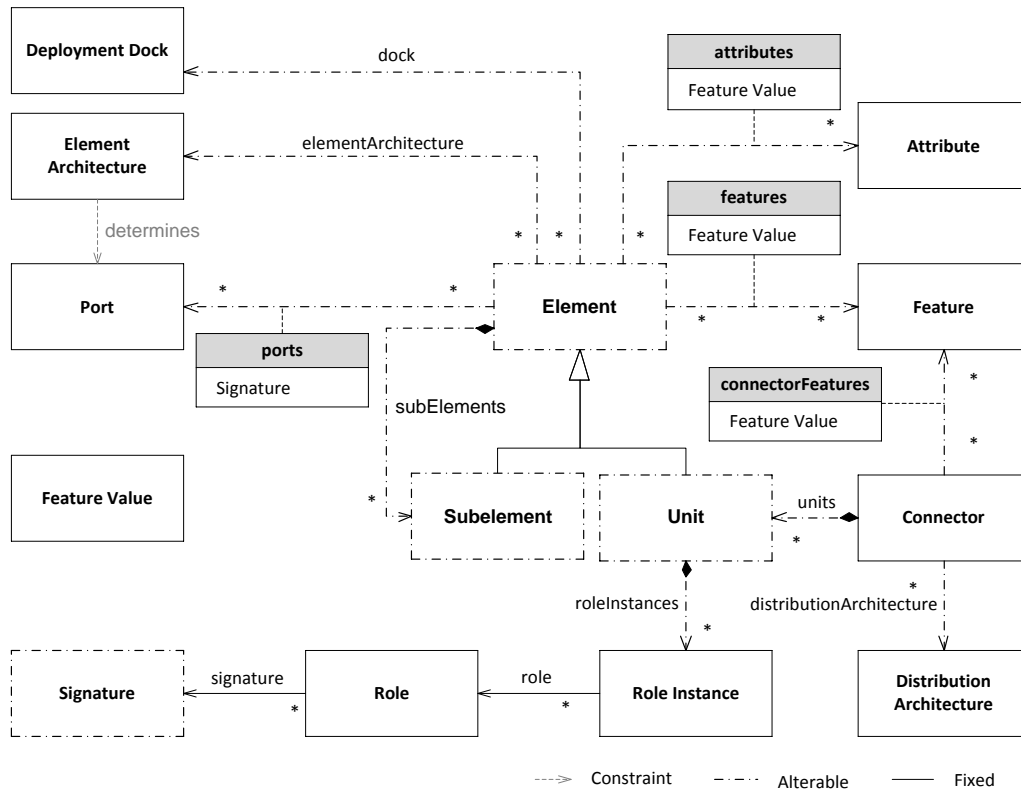


Fig. 7 Sets and relations describing a connector – meta-model

multiple realizations of the alterable sets and relations can satisfy the given (a) and (b); i.e., multiple variants of the connector exist. For example, the variants can be introduced by the existence of several element architectures suitable for refining a sub-element. The alterable sets and relations are determined by the constraints inferred from (a) and (b), e.g., *determines*. For the sake of brevity, we do not explicitly represent all of these constraints in the meta-model. Even though neither the concept of element architecture nor distribution architecture is required in CIC, both of them are included in the meta-model in order to make it possible to express these constraints explicitly.

Overall, the problem of finding a CIC can be interpreted as finding a realization of the alterable sets and relations with respect to (a) and (b). If multiple alternatives exist, an optimal one should be chosen.

Specifically, to be found by ARCAS, a realization of the alterable sets and relations in CIC embodies the following:

- (i) a particular distribution architecture that conforms to the communication style specified in the requirements specification and definition of connector units;
- (ii) vertical composition of element architectures (each of them describing a horizontal composition of its sub-elements) determining, which element architecture to choose for each (sub-) element in a distribution architecture (recursively at all levels of element nesting);
- (iii) actual parameters for the element architectures by providing actual signatures for the elements' ports.

### 3. OVERVIEW OF THE ARCAS METHOD

Following the ideas of Section 2.1, automated resolution of CIC can be viewed as a relational constraint solving problem where realization of the fixed sets and relations, as well as the constraints over the alterable sets and relations, form the constraint specification and a realization of the alterable sets and relations represents

a solution to the problem. For this purpose, it is advantageous to employ a constraint-solving technique based on a modeling language rich enough to express the required concepts. In general, relational constraint solving languages such as relational logic are well suited to this purpose.

Informally put, we create a constraint specification, referred to as a *connector theory* (CT) – in the sense of logic, capturing specification of a particular connector in terms of a logic theory, so that a model of such a theory represents a CIC of the specified connector. More precisely, since a CT may have more than one model, this theory basically represents a description of a set of all the alternative CICs of the specified connector. In other words, a model of a CT (in the sense of logic), provides a representation of all the sets and relations of the CIC meta-model from Fig. 7; thus, Fig. 7 also represents the meta-model of the CT.

Specifically, a CT consists of four parts (Fig. 8) based on the specifications determining a connector: (i) a definition of the abstractions global to all CT theories (reflecting the meta-model), (ii) images of element architectures, (iii) images of distribution architectures, (iv) an image of the requirements imposed by requirements and deployment specifications.

Semantically, a model of a CT represents (a) a correct CIC (the elements in the CIC are able to cooperate), and (b) a desired CIC (the corresponding connector complies with the requirements and deployment specifications).

To keep the CT simple and feasible for CIC resolution, the representation of communication styles, element types, and deployment is subject to “inlining”. For example, in (ii) and (iii) each element type is inlined by a list of all the element architectures suitable for this type.

It should be emphasized, that the parts (ii) and (iii) consider only a subset of the available element architectures resp. distribution architectures. In the former case, only the element architectures able to run in the deployment docks specified by the deployment specification are considered, in the latter case, only the distribution architectures conforming to the specified communication style are included.

Advantageously, except for part (i), which is shared for all applications and created by hand in advance, a CT can be constructed in an automated way by transformation of the specifications into formulas of a selected specification language. In ARCAS, for constructing each of the parts (ii)-(iv), there is a specific transformation (Fig. 8). A transformation processes a corresponding specification while considering several other specifications as additional parameters, producing the corresponding specification’s image. For example, the transformation of a single element architecture specification considers the specifications of other element architectures, element types, and deployment capabilities as additional parameters. A particular CT is constructed by applying transformations to all the relevant specifications.

Once a CT is constructed, a constraint solver available for the selected modeling language is employed to find out a model of the CT. This model can be easily programmatically converted to the corresponding CIC (Fig. 8).

The specifications are described in more detail in Section 4. The transformations of the specifications producing a particular CT are elaborated in Section 5. Since the Alloy modeling language [J02, J06, J11] and its solver Alloy Analyzer are good candidates for representing CT (it provides convenient syntax for definition of relations and their constraints), in Section 6 we describe the representation of CT using Alloy.

Referring back to Section 1 it should be emphasized that ARCAS is intended to be applied at either the deployment or runtime stage of an application. In the latter case, this would be due to a runtime modification of the architecture and/or deployment of the application. In this context, we interpret the term “emergent connector”.

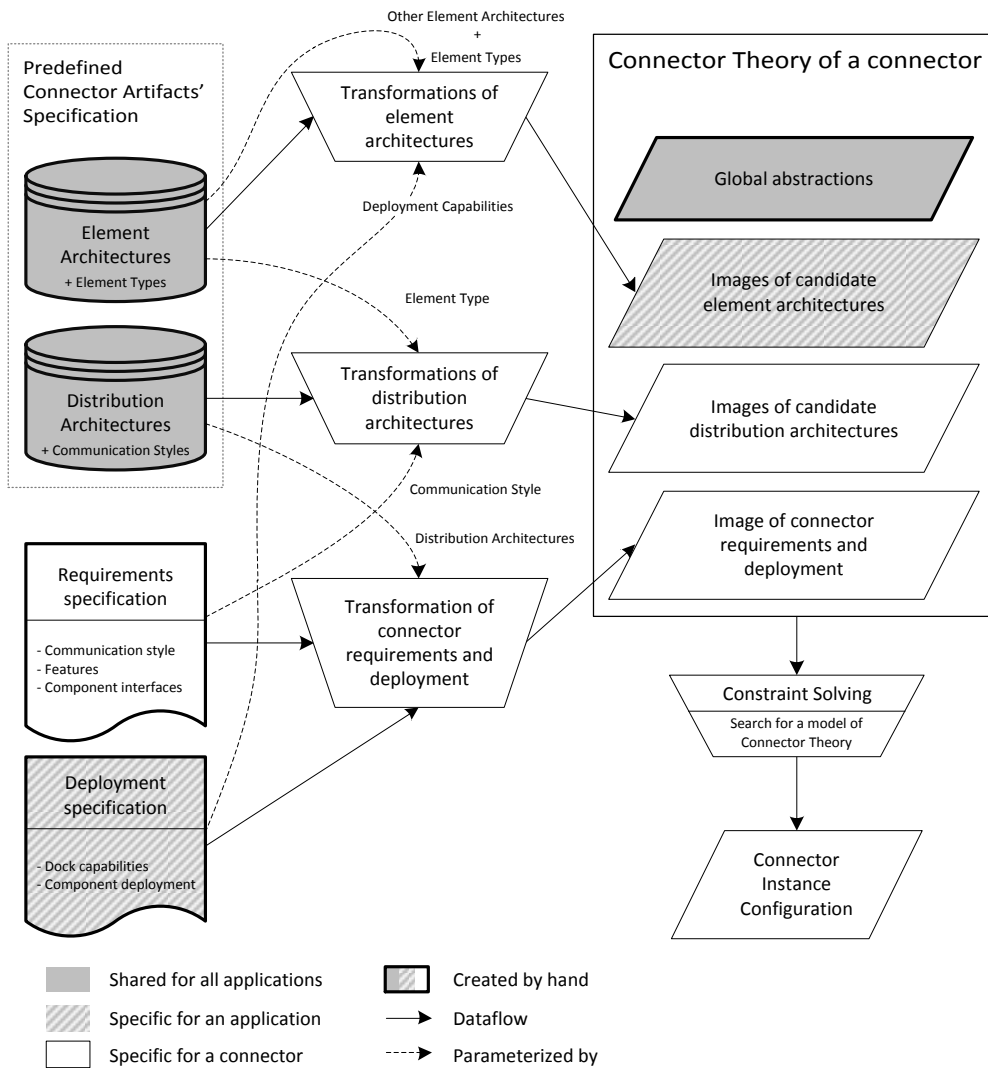


Fig. 8 Overview of the ARCAS method

#### 4. ARCAS INPUT: SPECIFICATIONS

In this section, we will elaborate on the specifications required as the input of the transformations in ARCAS (Fig. 8); the specifications were conceptually outlined in Section 2. In principle, each of them is defined by its meta-model, i.e., abstract syntax, and for practical reasons ARCAS includes connector definition language (CDL), i.e., concrete syntax, in which we will provide examples.

We will fully describe the abstract syntax and semantics, and give examples of the element type and element architecture specifications. For brevity, the other specifications (i.e., communication style, distribution architecture, requirements, and deployment specifications) are illustrated by an example based on Fig. 4 and Fig. 6, while their abstract syntax is provided in [KBPH12].

##### 4.1 Element Type & Element Architecture Specifications

The basic abstraction of `ElementArchitecture` (Fig. 9) is `ElementType`, which defines the external interface of an element, i.e., its ports. A port is defined as provided, required, or remote. The following illustrates specification of `ClientStub`, `SocketBasedSkeleton`, and `Logger` element types (the types of `LoggedClientStub`, `RMISkeleton`, and `FileLogger` from Fig. 6) in CDL:

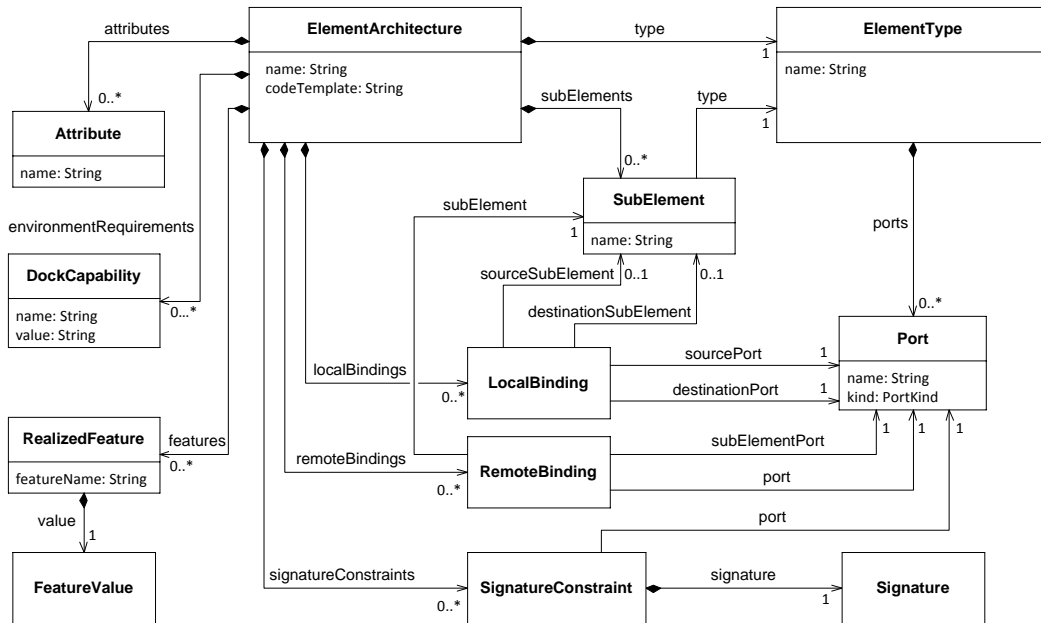


Fig. 9 Abstract syntax of Element Architecture specification

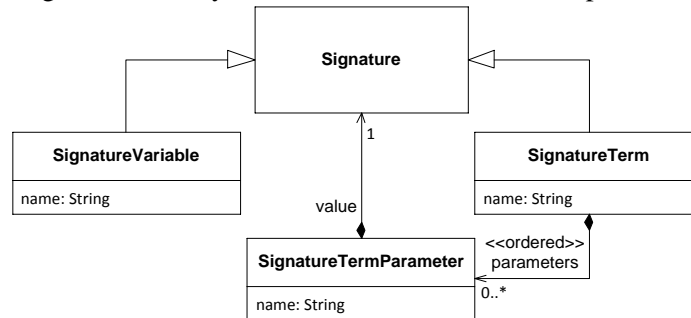


Fig. 10 Abstract syntax for Signature specification

```

element-type ClientStub
  local-provides: callIn      -- <port kind>: <port name>
  remote: mw
  
```

```

element-type SocketBasedSkeleton
  local-requires: factory
  local-requires: callOut
  mw: mw
  
```

```

element-type Logger
  local-provides: callIn
  local-requires: callOut
  
```

An element architecture refines an element type by determining a hierarchical composition of elements at two adjacent levels of nesting (the lower level is determined by the `subElements` relation in Fig. 9). Thus, only horizontal composition (of the elements at the lower level) is specified for `ElementArchitecture`; as to vertical composition, it is here expressed by enforcing a particular element type for a sub-element.

Further, indication of port bindings is defined separately for pairs of provided and required ports (`LocalBinding`), and for delegation among remote ports (`RemoteBinding`). The former case expresses either delegation or communication at the same level of nesting.

Port signature propagation is captured by expressing relations between the ports of a single element architecture (`SignatureConstraint`); these relations enforce equal or equally parameterized signatures of the related ports. This ensures an abstraction over element implementation, providing only the information important for vertical

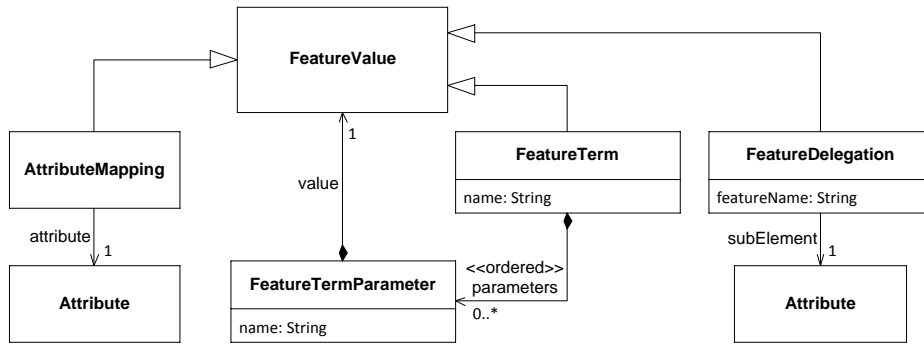


Fig. 11 Abstract syntax for specification of features

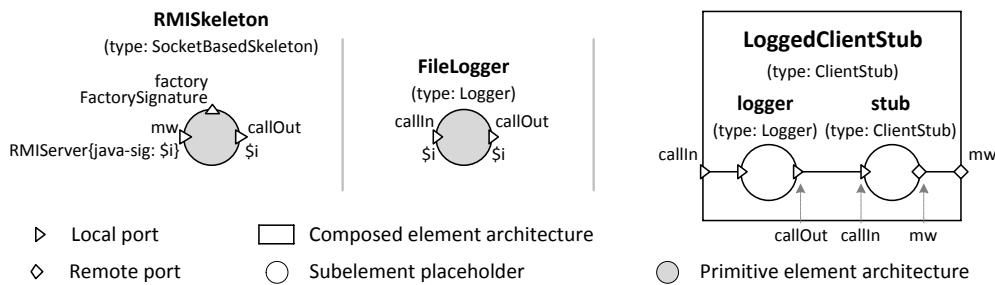


Fig. 12 Elaboration of FileLogger, RMISkeleton, and LoggedClientStub element architectures from Fig. 6

composition. The way *Signature* is defined in Fig. 10 indicates that a signature takes the form of either a signature variable, acting as a signature parameter, or a signature term (possibly parameterized).

As for the features (Fig. 11) realized by the element architecture (*RealizedFeature*), a feature value is represented by a possibly structured feature term. The value of a feature (sub-) term is either delegated to a sub-element (*FeatureDelegation*), explicit (set explicitly via a fixed feature term), or parameterized by an *attribute* (*AttributeMapping*), playing a similar role as signature variables in signature terms. Assuming an element *e*, the actual value of its attribute is defined either in a parent element (recursively) or directly at the level of the whole connector (in the requirements specification); in both cases, the value is propagated to *e* by feature delegation. For delegating a feature value, the higher-level element architecture needs to anticipate the features of its sub-elements. However, in a sub-element architecture corresponding features are not strictly required; if they are missing, this implies ignoring the delegated feature in the parent element.

For simplicity, environment requirements are represented by means of name-value pairs (*DockCapability*).

The CDL specification of the *LoggedClientStub*, *FileLogger*, and *RMISkeleton* element architectures from Fig. 12 (elaboration from Fig. 6) can take the following form:

```

element-architecture LoggedClientStub
  of-type: ClientStub
  sub-elements:
    stub: ClientStub           -- <name>: <type>
    logger: Logger
  bindings:
    this.callIn -> logger.callIn  -- -> local binding
    logger.callOut -> stub.callIn  -- <subelement>.<port>
    this.mw <-> stub.mw           -- <-> remote binding
  features:
    logging -> logger.logging     -- feature delegation

```

```

element-architecture FileLogger
  of-type: Logger
  signature-propagation:
    callIn: $I -- $I is a signature variable
    callOut: $I
  features:
    logging: ToFile{ -- explicit feature, set to ToFile{...}
      name: $FileName -- 'name' is a feature term parameter
    } -- $FileName is an attribute
  environment-requirements: {"Java VM" -> "JDK 1.4" }

element-architecture RMISkeleton
  of-type: SocketBasedSkeleton
  signature-propagation:
    callOut: $I -- $I is a signature variable
    mw: RMI{ java-sig: $I } -- parameterized signature
    -- 'java-sig' is a signature parameter
    factory: FactorySignature -- fixed port signature
  environment-requirements: {"Java VM" -> "JDK 1.6" }

```

Here, an important part is the definition of features and attributes, element types, signature propagation, and environment requirements:

`FileLogger` and `logging` give an example of an explicit feature. The feature value is parameterized by the `FileName` attribute, which is used in the implementation of `FileLogger`. The `logging` feature of `LoggedClientStub` illustrates the delegation of features. Here, the meaning is that `LoggedClientStub` provides `logging` only if this feature is provided by its `logger` sub-element (in the positive case, `logger` also defines the value of `logging`).

As to element types, notice that both `LoggedClientStub` and its sub-element `stub` are of the same element type; recall `RMISub` element architecture assigned to `stub` in Fig. 6. Thus, both `LoggedClientStub` and `RMISub` can be used to refine the `For_Caller` unit. In this example, `LoggedClientStub` was preferred over `RMISub` because of its `logging` feature.

Signature propagation is illustrated by `FileLogger`, where the use of the same signature variable `$I` for both `callIn` and `callOut` indicates that both ports have to have the same signature. A more complex example is `RMISkeleton`, where the signature of the `callOut` port is propagated as the `java-sig` parameter of the `mw` port's signature. The `RMISkeleton` specification also illustrates definition of a fixed port signature `FactorySignature`.

The environment requirements are expressed by means of required dock capability values. `RMISkeleton` requires the target deployment dock to support JDK 1.6.

## 4.2 Remaining Specifications

4.2.1 Communication Style. As outlined in Section 2, specification of a communication style determines the connector roles and their cardinality. The communication style of the connector from Fig. 3 can be specified as:

```

communication-style ProcedureCall
  roles:
    Callee -- lower bound = 1, upper bound = 1
    Caller(0..n) -- lower bound = 0, upper bound = n

```

4.2.2 Requirements. The key idea of requirements specification is that the particular connector architecture to be used for the connector is not specified explicitly; instead, it is declaratively determined by the selected communication style, required features, and desired deployment of components.

As the requirements specification is application-specific, the components to be connected and their interfaces (including signatures) have to be defined. Based on this, requirements are specified by selecting a communication style, mapping roles to

the component interfaces (i.e., defining connector endpoints), and by defining the required features of the connector. Feature requirements are determined by enumerating the acceptable feature values, where a value is represented by a feature term. There is also the option to define a feature requirement for a particular endpoint. Feature requirements can be composed using propositional operators.

The specification of requirements for the example from Fig. 3 can take the following form:

```

component CashDesk
  requires: InventoryItf inventory -- <interface kind>: <signature> <name>

component Inventory
  provides: InventoryItf inventory

connector CashDesk_to_Inventory
  communication-style: ProcedureCall
  endpoints:
    CashDesk.inventory as Caller -- <component.interface> as <role>
    Inventory.inventory as Callee
  features:
    security in { TLS1.2, SSL3 } -- <feature> in {<possible values>}
    Inventory.inventory.logging in { ToFile { name: "inventory.log" } }
    -- <component>.<interface>.<feature>
    -- 'name' is feature term parameter

```

4.2.3 Deployment. A deployment specification expresses the desired deployment of the connected components (not of connector deployment!). It determines the allocation of components to deployment docks and the capabilities of the deployment docks. The dock capabilities are based on the OMG D&C [OMG04, BB04] standard (syntactically expressed by means of typed name-value pairs). The specification of deployment for the example from Fig. 4 can take the following form:

```

allocation
  CashDesk to DockA
  Inventory to DockB

dock DockA
  capabilities: {"Java VM" -> "JDK 1.4", "OS" -> "Linux 2.4.28"}

dock DockB
  capabilities: {"Java VM" -> "JDK 1.6", "OS" -> "Linux 2.6.36"}

```

4.2.4 Distribution Architecture. The distribution architecture specification determines the desired communication style, the element type of the top-level element architectures refining units, their cardinality, the units' remote bindings, and mapping of roles to ports in units. The specification also explicitly states how features are addressed by specific units (feature delegation). The distribution architecture specification of the connector from Fig. 5 can take the following form:

```

distribution-architecture RPC
  communication-style: ProcedureCall
  units:
    For_Callee: ServerSkeleton -- <name>: <element type>
    For_Caller (0..n): ClientStub -- <name>(<cardinality>): <type>
  remote-bindings:
    For_Caller(i).mw <-> For_Callee.mw -- <unit>.<port>
  role-mapping:
    For_Caller(i).callIn as Caller(i) -- <unit>.<port> as <role>
    For_Callee.callOut as Callee
  features:
    security -> For_Callee.security -- feature delegation

```

An important concept is the specification of multiplicity of unit instances such as For\_Caller (0..n): ClientStub, which says that there will be up to n For\_Caller

units of the type `ClientStub`. In consequence, multiplicity is to be expressed also in remote-bindings and role-mapping. For example, `For_Caller(i).mw <-> For_Callee.mw` means that the `mw` port of all the `For_Caller` units is bound to the `mw` port of the `For_Callee` unit. Likewise, `For_Caller(i).callIn as Caller(i)` means that the `callIn` port of each `For_Caller` unit has the role `Caller`.

## 5. CONSTRUCTING CONNECTOR THEORY BY TRANSFORMATIONS

As outlined in Section 3, a connector theory can be constructed in an automated way by transformations of the specifications. In this section, we describe the transformations and their products in terms of propositional logic with relational calculus, using only basic logical and relational operators. In general, a transformation results in a formula. The set of all formulas resulting from transformations of the specifications relevant to a particular connector forms the corresponding connector theory.

The formulas express the constraints on alterable sets/relations by binding them to the fixed ones (Fig. 7). In the formulas, we rely on predicates, the semantics of which is explained informally in this section to help the reader get an intuitive understanding of them; detailed semantics is provided only for a selection of the predicates. We further elaborate on the description of selected predicates in Section 6 by implementing them in Alloy<sup>1</sup>. There, we also illustrate a construction of the fixed sets and relations. For the fixed sets and relations, we assume that a realization is available, implied by the specifications (both those of predefined artifacts, and those

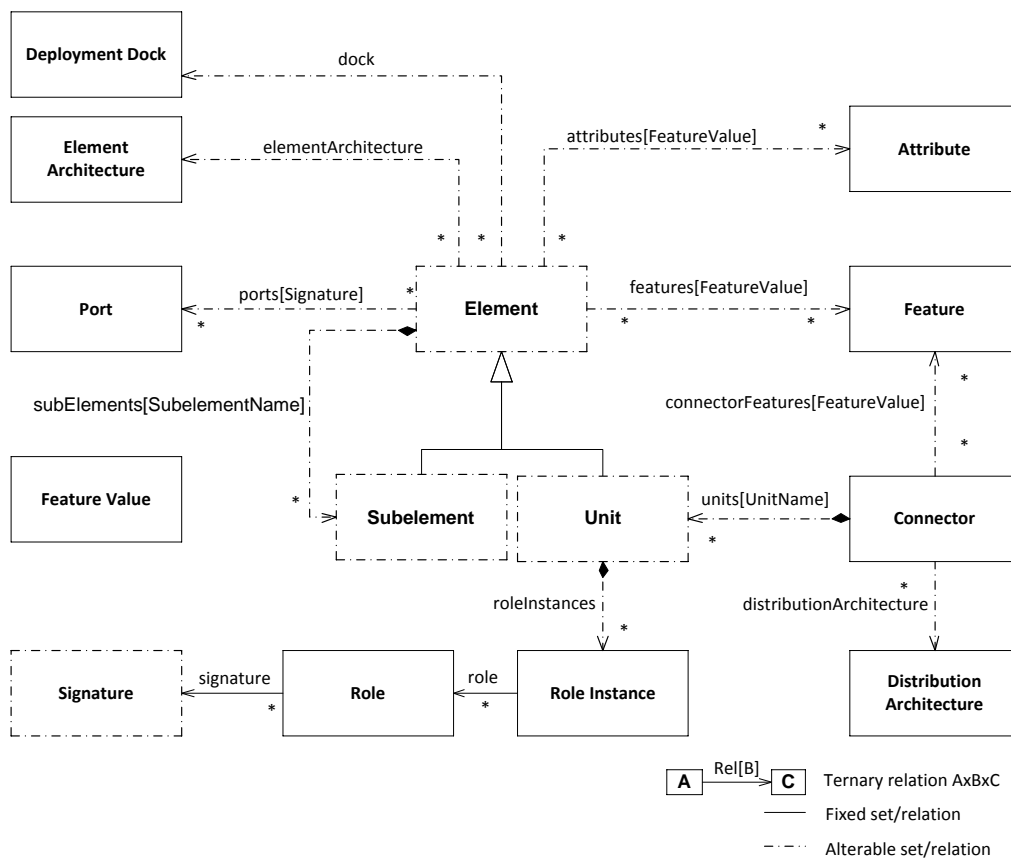


Fig. 13 Modified connector theory meta-model (by ternary relations)

<sup>1</sup> The full formalization in Alloy is provided at [http://d3s.mff.cuni.cz/projects/components\\_and\\_services/arcas/](http://d3s.mff.cuni.cz/projects/components_and_services/arcas/)

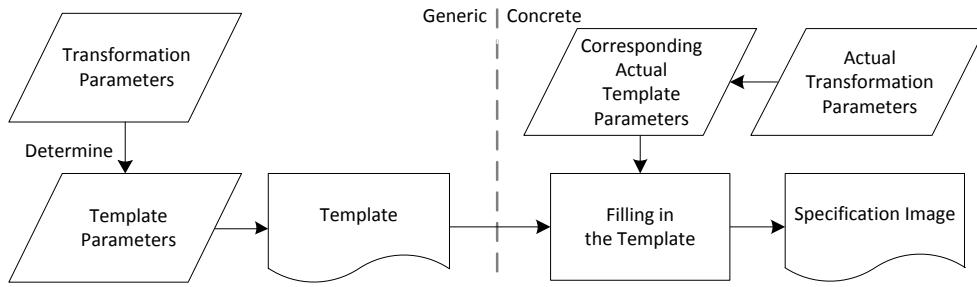


Fig. 14 Generic workflow of transformations

of requirements and deployment, specific to a connector).

For defining the actual transformations based on predicate logic with relational algebra, it is advantageous to introduce a slightly modified version (Fig. 13) of the connector meta-model from Fig. 7. The rationale for such modification is that certain relations of the original meta-model (i.e., *features*, *connectorFeatures*, *attributes*, and *ports*) are parameterized. In order to capture this in relational calculus, the originally binary parameterized relation is now expressed as a ternary relation (e.g., the relation *features* in Fig. 13). In addition, given an element  $e$ , in order to express constraints over its particular sub-element  $se$ , the name of the sub-element is explicitly employed. This is necessary, since the constraints on  $se$  are imposed in the element architecture of  $e$ . Therefore, the original binary relation *subElements* is replaced by a ternary relation introducing an identifier of the sub-element (the identifier is taken over from the definition of  $se$  in the element architecture of  $e$ ). The relation *units* is to be modified in a similar way to *subElements*.

Because the formulas resulting from a single transformation have a fixed internal structure, we present the transformations in terms of parameterized templates (Fig. 14). The template parameters are derived from the transformation parameters (Section 3). Based on actual transformation parameters, the transformation produces a specification's image (i.e., a part of the connector theory under construction) by filling in the template.

In detail, we describe the transformation of a composite element architecture specification focusing on the corresponding template and derivation of the template parameters. For brevity, the effect of other transformations (i.e., of primitive element architecture, distribution architecture, and requirements specifications) is illustrated by an example of their product, while their full description is provided in [KBPH12].

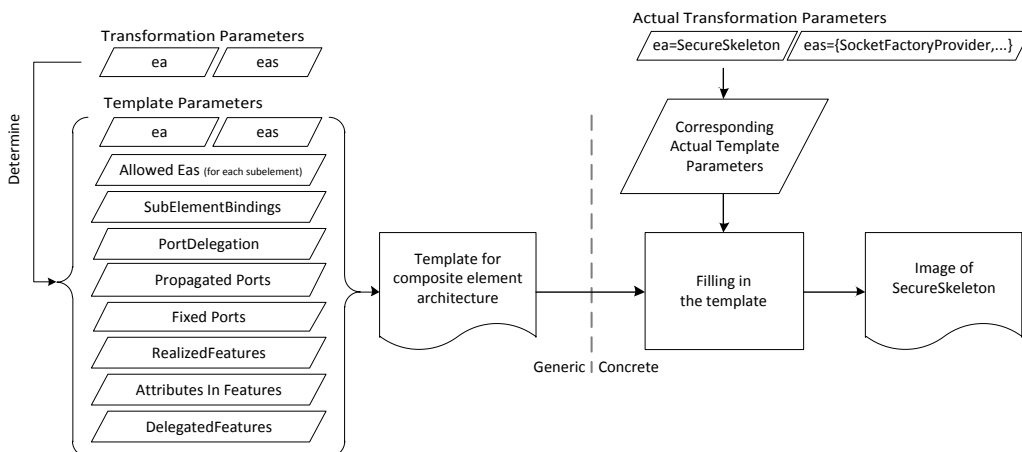


Fig. 15 Transformation workflow for composite element architecture

```

∀e ∈ Elements : ElementHasEA(e, <ea>) ⇒ (
  AvailablePorts(e, <ea.type.ports>) ∧
  SubElements(e, <ea.subElements>) ∧
  SupportedFeatures(e, <ea.features>) ∧
  RequiredAttributes(e, <ea.attributes>) ∧

  -- definition of sub-elements
  <foreach se in ea.subElements >
    -- <se> sub-element
    AllowedEAsForSubElement(e, <se>, <AllowedEAsse>) ∧
    PropagateDeployment(e, <se>) ∧
  <endforeach>

  -- port bindings
  <foreach (se1, port1, se2, port2) in SubElementBindings >
    PortBinding(e, <se1>, <port1>, <se2>, <port2>) ∧
  <endforeach>
  -- port delegation
  <foreach (port, se, seport) in PortDelegations>
    PortDelegation(e, <port>, <se>, <seport>) ∧
  <endforeach>

  -- signature propagation
  <foreach (port1, port2) in PropagatedPorts >
    PropagateSignature(e, <port1>, <port2>) ∧
  <endforeach>
  -- enforcing fixed signatures
  <foreach (port1, port2) in FixedPorts >
    PortHasSignature(e, <port>, <signature>) ∧
  <endforeach>

  -- enforcing values of the directly realized features
  <foreach f in RealizedFeatures >
    FeatureHasValue(e, <f.name>, <Encode(f.value)>) ∧
  <endforeach>
  -- defining attributes referred in feature values
  <foreach (feature, parameter, attribute) in AttributesInFeatures>
    FeatureValueUsesAttribute(e, <feature>, <parameter>, <attribute>) ∧
  <endforeach>
  -- feature delegation
  <foreach (feature, se, sefeature) in DelegatedFeatures >
    FeatureDelegation(e, <feature>, <se>, <sefeature>) ∧
  <endforeach>
)

```

Fig. 16 Template for image of composite element architecture

## 5.1 Transforming specification of a composite Element Architecture

The transformation workflow for a composite element architecture specification is illustrated in Fig. 15.

5.1.1 Template. The template (Fig. 16) formulates the constraints on any element  $e$  employing a particular composite element architecture  $ea$ , the specification of which is subject to the transformation. The constraints are derived from the semantics of the composite element architecture concept (Section 2 and 4.1). The template takes the form of an implication.

Technically, in the meta-model these constraints restrict the relations that are (potentially transitively) related to  $e$  as an element of the set *Element*. The antecedent of the implication assumes validity of the *ElementHasEA* predicate over the *elementArchitecture* relation ( $e$  employs  $ea$ ). The consequent of the implication

includes (i) general element architecture constraints, (ii) sub-element constraints, (iii) port constraints, (iv) signature constraints, and (v) feature constraints.

As to (i), after enforcing that  $e$  contains the ports defined by the element type of  $ea$  (the  $ports$  relation is constrained by *AvailablePorts*), the set of sub-elements of  $e$  is constrained to make it conform to  $ea$  (via constraining the  $subElements$  relation by *SubElements*). In addition, it is desirable to make sure that only the realized features and declared attributes of  $ea$  may be employed by  $e$  (the  $features$  and  $attributes$  relations are constrained by *SupportedFeatures* resp. *RequiredAttributes*).

As to (ii), the definitions of the sub-elements are reflected as follows. For each sub-element, the set of element architectures that can be employed for its refinement is explicitly prescribed (the  $subElements$  and  $elementArchitecture$  relations are constrained by *AllowedEAsForSubElement*). This ensures the refinement consistency of the sub-elements. Note that the sub-element's ports are determined by the prescribed element architectures (as they have the same element type). Since  $e$  has to be deployed on the same node as its sub-elements, their deployment is propagated to  $e$  ( $subElements$  and  $node$  are constrained by *PropagateDeployment*).

As to (iii), the port bindings and port delegations are reflected by forcing the signatures of the involved ports to be equal (via constraining the  $subElements$  and  $ports$  relations by *PortBinding* resp. *PortDelegation*). This, along with enforcing signature propagation in the images for sub-element architectures, ensures the composition consistency of the sub-elements.

As to (iv), according to the signature propagation constraints for  $ea$ , the signatures/signature parameters of the involved ports of  $e$  have to be equal ( $ports$  is constrained by *PropagateSignature*). Note that we have abstracted away details of signature propagation for the sake of brevity. It is also necessary to enforce the fixed port signatures ( $ports$  is constrained by *PortHasSignature*).

As to (v), the values of the feature directly realized by  $ea$  have to be reflected by  $e$  (via constraining  $features$  by *FeatureHasValue*). Since a feature value can use an attribute as its parameter, the attribute value and the respective feature parameter value have to be equal ( $features$  and  $attributes$  are constrained by *FeatureValueUsesAttribute*). Finally, the delegated features have to be captured ( $subElements$  and  $features$  are constrained by *FeatureDelegation*).

For illustration, the semantics of “representative” predicates is presented in Table 1 via a textual description and an equivalent logical formula.

Table 1. Semantics of predicates for composite element architecture image

Predicate name	Description	Formula
AvailablePorts ( $e$ , portSet)	the element $e$ contains exactly the ports in $portSet$	$\forall p \in Port, \forall s \in Signature:$ $(e, s, p) \in ports$ $\Leftrightarrow p \in portSet$
AllowedEAsFor-SubElement ( $e$ , subelement, eas)	the sub-element $se$ (of the element $e$ ), having the name $subelement$ , can refine only an element architecture from $eas$	$\forall se \in Element:$ $(e, subelement, se) \in subElements$ $\Rightarrow$ ( $\forall ea \in ElementArchitecture:$ $(se, ea) \in elementArchitecture$ $\Rightarrow ea \in eas$ )
PortBinding ( $e$ , subelement1, port1, subelement2, port2)	for the sub-elements $se1$ and $se2$ of $e$ with names $subelement1$ resp. $subelement2$ , the $port1$ of $se1$ and $port2$ of $se2$ have the same signature	$\forall se1, se2 \in Element,$ $(e, subelement1, se1) \in subElements$ $\wedge$ $(e, subelement2, se2) \in subElements$ $\Rightarrow$ ( $\forall s \in Signature:$ $(se1, s, port1) \in ports$ $\Leftrightarrow$ $(se2, s, port2) \in ports$ )

FeatureValueUses-Attribute (e, feature, parameter, attribute)	for the feature value ( $fv$ ) of $feature$ in the element $e$ , $attribute$ of $e$ and $parameter$ of $fv$ have the same value	$\forall fv \in FeatureValue,$ $\forall av \in FeatureValue:$ $\{(e, fv, feature) \in features\}$ $\wedge$ $(e, av, attribute) \in attributes\}$ $\Rightarrow$ $(fv, av) \in parameter$
FeatureDelegation (e, feature, subelement, se-feature)	for the sub-element $se$ (of the element $e$ ) with name $subelement$ , $feature$ of $e$ and $se$ - $feature$ of $se$ have the same value	$\forall se \in Element:$ $(e, subelement, se) \in subElements$ $\Rightarrow$ ( $\forall fv \in FeatureValue:$ $(e, fv, feature) \in features$ $\Leftrightarrow$ $(se, fv, se-feature) \in features$ )

5.1.2 Transformation parameters. The transformation parameters are: (i) the element architecture  $ea$  to be transformed and (ii) the set of all predefined element architectures  $eas$  (Fig. 15).

5.1.3 Template Parameters. In addition to the element architecture  $ea$  to be transformed and the set of all predefined element architectures  $eas$ , these are  $AllowedEAs_{se}$  (separately for each sub-element of  $ea$ ),  $SubElementBindings$ ,  $PortDelegation$ ,  $RealizedFeatures$ ,  $AttributesInFeatures$ , and  $DelegatedFeatures$  (Fig. 15).

$AllowedEAs_{se}$  contains the element architectures that may be used for refining the

$AllowedEAs_{se} = \{subea \mid subea \in eas \wedge subea.type = se.type\}$ $SubElementBindings = \{(se1, port1, se2, port2) \mid se1, se2 \in ea.subElements \wedge$ $port1 \in se1.type.ports \wedge port2 \in se2.type.ports \wedge$ $\exists lb \in ea.localBindings \wedge$ $IsSource(lb, se1, port1) \wedge IsDestination(lb, se2, port2)\}$ $PortDelegation = \{(port, se, seport) \mid port \in ea.type.ports \wedge se \in ea.subElements \wedge$ $se \in ea.subElements \wedge seport \in se.type.ports$ $\wedge$ $(\exists lb \in ea.localBindings \wedge IsSource(lb, ea, port) \wedge$ $IsDestination(lb, se, seport))$ $\vee$ $(\exists lb \in ea.localBindings \wedge IsSource(lb, se, seport) \wedge$ $IsDestination(lb, ea, port))$ $\vee$ $(\exists rb \in ea.remoteBindings \wedge IsEndpoint(lb, se, seport) \wedge$ $IsEndpoint(lb, ea, port))$ $\}$ $PropagatedPorts = \{(port1, port2) \mid port1, port2 \in ea.type.ports \wedge$ $sc1, sc2 \in ea.signatureConstraints \wedge$ $sc1.port = port1 \wedge sc2.port = port2 \wedge$ $UseTheSamePlaceholder(sc1.signature, sc2.signature)\}$ $FixedPorts = \{(p, sc.signature) \mid p \in ea.type.ports \wedge$ $sc \in ea.signatureConstraints \wedge$ $sc.port = p \wedge IsFixedTerm(sc.signature)\}$ $RealizedFeatures = \{f \mid f \in ea.features \wedge \neg IsFeatureMapping(f.value)\}$ $AttributesInFeatures = \{(f.name, param.name, a.name) \mid a \in ea.attributes \wedge$ $f \in ea.features \wedge$ $term \in TransitiveClosure(f.value) \wedge$ $param \in term.parameters \wedge$ $param.value = a\}$ $DelegatedFeatures = \{(f, se, sef) \mid se \in ea.subElements \wedge f \in ea.features \wedge$ $sef \in se.features \wedge IsFeatureMapping(f.value) \wedge$ $f.value.featureName = sef.featureName \wedge$ $f.value.subElement = se\}$
---

Fig. 17 Template parameters for composite element architecture

```

 $\forall e \in Elements : ElementHasEA(e, SocketFactorySkeleton) \Rightarrow ($ 
  AvailablePorts( $e, \{Mw, CallOut\}$ )  $\wedge$ 
  SubElements( $e, \{SocketFactory, Skeleton\}$ )  $\wedge$ 
  SupportedFeatures( $e, \{Security\}$ )  $\wedge$ 
  RequiredAttributes( $e, \emptyset$ )  $\wedge$ 

  -- definition of sub-elements
  -- SocketFactory sub-element
  AllowedEAsForSubElement( $e, SocketFactory, \{SocketFactoryProvider\}$ )  $\wedge$ 
  PropagateDeployment( $e, SocketFactory$ )  $\wedge$ 
  -- Skeleton sub-element
  AllowedEAsForSubElement( $e, Skeleton, \{RMISkeleton\}$ )  $\wedge$ 
  PropagateDeployment( $e, Skeleton$ )  $\wedge$ 

  -- port bindings
  PortBinding( $e, Skeleton, Factory, SocketFactory, Factory$ )  $\wedge$ 
  -- port delegation
  PortDelegation( $e, CallOut, Skeleton, CallOut$ )  $\wedge$ 
  PortDelegation( $e, Mw, Skeleton, Mw$ )  $\wedge$ 

  -- signature propagation
  -- enforcing fixed signatures
  -- enforcing values of the realized features
  -- defining attributes referred in feature values
  -- feature delegation
  FeatureDelegation( $e, Security, SocketFactory, Security$ )
 $)$ 

```

Fig. 18 Example: Image of SocketFactorySkeleton composite element architecture

sub-element  $se$  (decision is based on the element type of  $se$  defined by  $ea$ ). *SubElementBindings* comprises all pairs of sub-elements of  $ea$  and their ports for which  $ea$  defines a binding. *PortDelegation* contains the tuples ( $ea$  port  $port$ , sub-element  $se$ , sub-element port  $seport$ ) that participate in port delegation (including delegation of both local and remote ports). *RealizedFeatures* includes all the features realized directly by  $ea$ . *AttributesInFeatures* contains all the tuples ( $feature$ ,  $feature$  parameter,  $attribute$ ) such that  $attribute$  is used as the value of  $feature$  parameter which is part of the feature term associated with  $feature$ . *DelegatedFeatures* contains all the tuples ( $ea$  feature  $f$ , sub-element  $se$ , sub-element feature  $sef$ ) such that  $f$  is mapped to  $sef$ .

Mathematically, the template parameters are determined from transformation parameters as shown in Fig. 17.

5.1.4 Example: Image of SocketFactorySkeleton. Referring back to the example from Sections 2 and 4, consider the actual transformation parameters  $ea = SecureSkeleton$  and  $eas = \{SerializedServerSkeleton, SocketFactorySkeleton, SocketFactoryProvider, RMISkeleton, \dots\}$ ; the transformation will produce the image presented in Fig. 18.

## 5.2 Transforming specification of a primitive Element Architecture

To illustrate this transformation, we provide an example of the *FileLogger* primitive element architecture image (Fig. 19); a more detailed description of the template, as well as a description of the transformation and template parameters, is given in [KBPH12].

5.2.1 Example: Image of FileLogger. The image of *FileLogger* is very similar to an image of a composite element architecture (i.e., it formulates constraints on element  $e$  employing *FileLogger* in the form of an implication). Therefore, we will skip description of the common parts, referring the reader back to Section 5.1.

```

 $\forall e \in Elements : ElementHasEA(e, FileLogger) \Rightarrow ($ 
  AllowedDeployment(e, {DockA})  $\wedge$ 
  AvailablePorts(e, {CallIn, CallOut})  $\wedge$ 
  SubElementEAs(e,  $\emptyset$ )  $\wedge$ 
  SupportedFeatures(e, {Logging})  $\wedge$ 
  RequiredAttributes(e, {FileName})  $\wedge$ 

  -- signature propagation
  PropagateSignature(e, CallIn, CallOut)  $\wedge$ 

  -- enforcing fixed signatures

  -- enforcing values of the realized features
  FeatureHasValue(e, Logging, LoggingToFile)  $\wedge$ 

  -- defining attributes referred in feature values
  FeatureValueUsesAttribute(e, Logging, fileName, FileName)
 $)$ 

```

Fig. 19 Example: Image of FileLogger primitive element architecture

As for the constraints in the consequent, it is necessary to express that deployment of  $e$  is possible only to the docks compatible with *FileLogger* (*DockA*); this is reflected by constraining the relation *node* by *AllowedDeployment*. Further,  $e$  can feature only the ports defined by the *Logger* element type – the element type of *FileLogger* (*AvailablePorts*). It is also desirable to make sure that the set of sub-elements of  $e$  is empty since *FileLogger* is primitive (*SubElements*). As for the further constraints, only the realized features and declared attributes of *FileLogger* may be employed by  $e$  (*SupportedFeatures* resp. *RequiredAttributes*).

Finally, after capturing the potential fixed signatures and signature propagation (*PropagateSignature*), and reflecting the values of the directly realized features (*FeatureHasValue*), it is necessary to make the attributes and the particular feature values equal (*FeatureValueUsesAttribute*).

### 5.3 Transforming specification of a Distribution Architecture

To illustrate this transformation, we provide an example of the *RPC* distribution architecture (Fig. 20); a more detailed description of the template, as well as a description of the transformation and template parameters, is given in [KBPH12].

5.3.1 Example: Image of RPC Distribution Architecture. Again, the image formulates constraints in the form of an implication (Fig. 20). The constraints relate to the whole connector, i.e., the image imposes the constraints on the current connector in case it employs the distribution architecture *RPC*. Note, that the corresponding element of the *Connector* set from the meta-model is not explicitly mentioned (in contrast to  $e$  in the case of element architecture), since we assume it is unique. This assumption is correct, as a connector theory describes a single connector.

As for the general distribution architecture constraints, the connector can feature only the roles defined by the communication style refined by the *RPC* distribution architecture (the *role* relation is constrained by *AvailableRoles*). Further, the cardinality of the roles is reflected (*roleInstance* is constrained by *RolesWithSingleCardinality* and *RolesWithMultipleCardinality*). Similarly, only the units defined by the *RPC* distribution architecture are allowed (*unit* is constrained by *AvailableUnits*), and their cardinality is reflected (*unit* is constrained by *UnitsWithSingleCardinality* and *UnitsWithMultipleCardinality*). Further, the connector may refer only to the features realized by *RPC* (*featureRequirements* is constrained by *SupportedFeatures*).

The next part of the template focuses on properties of the units. For each unit, the set of element architectures that can be employed for its refinement is explicitly

```

EmployedDA(RPC) ⇒ (
  AvailableRoles({Caller, Callee}) ∧
  RolesWithSingleCardinality({Callee}) ∧
  RolesWithMultipleCardinality({Caller}) ∧

  AvailableUnits({For_Caller, For_Callee}) ∧
  UnitsWithSingleCardinality({For_Callee}) ∧
  UnitsWithMultipleCardinality({For_Caller}) ∧

  SupportedFeatures({Security}) ∧

  -- definition of units
  -- For_Caller unit
  AllowedEAsForUnit(For_Caller, {LoggedClientStub, RMISStub}) ∧
  -- For_Callee unit
  AllowedEAsForUnit(For_Callee, {SerializedServerSkeleton,
    SocketFactorySkeleton}) ∧

  -- association of units' ports with roles
  PortAssociatedWithRole(For_Caller, CallIn, Caller) ∧
  PortAssociatedWithRole(For_Callee, CallOut, Callee) ∧

  -- remote port bindings
  RemoteBinding(For_Callee, Mw, For_Caller, Mw) ∧

  -- feature delegation
  FeatureDelegation(Security, For_Callee, Security)
)

```

Fig. 20 Example: Image of RPC distribution architecture

prescribed (the *unit* and *elementArchitecture* relations are constrained by *AllowedEAsForUnit*). This ensures the refinement consistency of the units. Note that prescribing the element architectures for a unit determines its ports (as these element architectures have the same element type).

Further, the association of roles with units' ports is expressed (via constraining *role* and *unit* by *PortAssociatedWithRole*). The remote port bindings are reflected by forcing the signatures of the involved ports to be equal (*unit* and *ports* are constrained by *RemoteBinding*).

Finally, the feature delegation has to be captured (*unit* and *features* are constrained by *FeatureDelegation*).

#### 5.4 Transforming Requirements and Deployment specifications

To illustrate this transformation of requirements and deployment specifications, we provide an example of the *CashDesk\_to\_Inventory* connector (Fig. 21); the template, as well as a description of the transformation and template parameters, is given in [KBPH12].

5.3.3. Example: Image of CashDesk\_to\_Inventory Requirements and Deployment. The image formulates constraints on the selection of predefined distribution and element architectures with respect to required features and deployment. Since communication style is the binding concept of requirements specification and predefined artifacts, the constraints are focused on the actual endpoints of the connector (i.e., actual roles' instances).

Therefore, it is first necessary to define the available endpoints of the connector (via constraining the *units* and *roleInstances* relations by *DefinedEndpoints*). Further, the set of distribution architectures that can be employed for refinement of the connector is explicitly prescribed (*distributionArchitecture* is constrained by *AllowedDAs*).

```

DefinedEndpoints({CashdeskCaller, InventoyCallee}) ∧
AllowedDAs( {RPC, LocalProcedureCall, ... } ) ∧

-- definition of connector endpoints
-- CashdeskCaller endpoint
HasRole(CashdeskCaller, Caller) ∧
HasSignature(CashdeskCaller, InventoryItf) ∧
IsDeployedOn(CashdeskCaller, DockA) ∧
-- InventoyCallee endpoint
HasRole(InventoyCallee, Callee) ∧
HasSignature(InventoyCallee, InventoryItf) ∧
IsDeployedOn(InventoyCallee, DockB) ∧

-- definition of endpoints' features
EndpointFeatureRequirements(CashdeskCaller, Logging, LoggingToFileFoo) ∧

-- definition of global connector features
ConnectorFeatureRequirements( Security, SSL)

```

Fig. 21 Example: Image of CashDesk\_to\_Inventory requirements specification

As for each endpoint, its role is explicitly defined (*role* is constrained by *HasRole*), the signature of the associated component is assigned to the endpoint (*signature* is constrained by *HasSignature*), and its required deployment is enforced (via constraining *roleInstances* and *node* by *IsDeployedOn*).

Further, each feature requirement imposed directly on a particular endpoint is expressed (*roleInstances* and *features* are constrained by *EndpointFeatureRequirements*). Finally, the required values of the global connector features are enforced (via constraining *connectorFeatures* by *ConnectorFeatureRequirements*).

## 6. ARCAS IN ALLOY

In this section, after providing a very brief introduction to the Alloy [J02] modeling language, we show how a connector theory can be expressed in Alloy. We also discuss the approaches for selecting an optimal CIC with the help of Alloy Analyzer.

### 6.1 Brief Introduction to Alloy

This section gives a brief introduction to the Alloy modeling language - a formal modeling language based on a first-order predicate logic with operators from set theory (e.g., intersection, cartesian product), relational algebra (e.g., relational join, transitive closure), and basic arithmetics (e.g., integer operations, set cardinality) [J02, J06]. Further details of the current Alloy syntax can be found in [J11].

The language is based on the notions of *signature* and *relation*. A signature is a set of abstract elements; relations are defined upon such sets. Alloy allows the constraint of relations by *facts* (first-order logic formulas). A fact can employ named predicates and function symbols. In general, an Alloy specification represents a first-order logic theory (*Alloy theory*<sup>2</sup>) determined by signature, relation, and facts definitions.

Alloy Analyzer – the associated model solver – can either find a model of an Alloy theory, or check its models against a given property (expressed as a predicate). Alloy Analyzer converts the Alloy theory to a SAT formula; using an underlying general-purpose SAT solver, it solves the formula, and then maps the result to an Alloy theory model. As an aside, Alloy Analyzer requires the domains of signatures and relations to be explicitly bounded (due to the mapping to SAT).

<sup>2</sup> In the Alloy documentation, Alloy theory is called Alloy model and a model of an Alloy theory is called Alloy model instance.

```

01 one sig Connector {
02   distributionArchitecture: one DistributionArchitecture,
03   units: UnitName one -> set Unit,
04   connectorFeatures: Feature set -> lone FeatureValue
05 }
06
07 abstract sig Element {
08   subElements: SubElementName lone -> lone SubElement,
09   ...
10 }
11 sig SubElement extends Element {}

```

Fig. 22 Example of a signature definition in Alloy

In general, a definition of a signature *S* is interpreted in such a way that each of its fields *F* represents a relation between *S* and the signatures introduced by *F*. In a similar vein, *nesting* of signatures determines a subset relation on the signatures, while an *abstract* signature contains the elements of its nested signatures only.

In Fig. 22, the signature *Connector* is defined (*sig*) by listing relations *distributionArchitecture*, *units*, and *connectorFeatures*. Signature nesting is expressed by the *extends* keyword (line 11); *abstract* defines an abstract signature (line 7).

Obviously, the syntax complies with the object-oriented paradigm by defining signature as a structure constituting fields. Moreover, signature nesting resembles sub-typing, whereas abstract signatures are akin to an abstract super-type definition.

A definition of a signature/relation may contain multiplicity constraints *one*, *lone*, *set*, *some*: the *Connector* set has to have exactly one element (i.e., a *singleton* signature). Similarly, *lone* denotes zero or one, while *some* at least one, and *set* zero or more elements. Thus, *distributionArchitecture* associates each element of *Connector* with exactly one element of *DistributionArchitecture*. Further, *units* is a relation between three sets: *Connector*, *UnitName* and *Unit*. For each *c* of *Connector* and each *un* of *UnitName* there is a subset *SU* (-> set) of *Unit*, so that for *u* ∈ *SU* the triple <*c*, *un*, *u*> is in *units*. Moreover, for each *c* and *u* there is exactly one *un* (*one* ->), so that <*c*, *un*, *u*> is in *units*. The relation *connectorFeatures* is defined similarly.

A fact expresses a constraint over the sets and relations introduced by signature declarations. Each fact in an Alloy specification is an axiom of the theory determined by the specification.

```

01 pred isSubElement[parent: Element, child: SubElement] {
02   child in univ.(parent.subElements)
03 }
04 fact ElementHierarchyIsTree {
05   -- There are no cycles among elements
06   no iden & ^{ parent: Element, child: SubElement |
07     isSubElement[parent, child] }
08   -- Every element which is not a unit has exactly one parent
09   all e: Element | e in SubElement <=>
10     one parent: Element | isSubElement[parent, e]
11 }

```

Fig. 23 Example of a fact definition in Alloy

In Fig. 23, the fact *ElementHierarchyIsTree* (lines 4-11) describes the properties of *subElements* using the predicate *isSubElement*. In principle, it expresses that there are no cycles among the elements of *SubElement* with respect to the relation *subElements*, and that each element of *SubElement* has exactly one parent with respect to *subElements*.

The fact consists of two clauses (lines 6-7, 9-11) bound by a conjunction (expressed implicitly by a new line). To illustrate the basic Alloy constructs related to

facts, consider the first clause (lines 6-7). Here, the identity relation (`iden`) is forced to have an empty (no) intersection (`&`) with the transitive closure (`^`) of the relation defined in the curly brackets. This relation contains all the pairs `parent` (of `Element`) and `child` (of `SubElement`), such that they satisfy the predicate `isSubElement`. Note, that the operator `|` imposes a constraint on its left operand by the predicate being its right operand.

As to other syntactic constructs, consider the `isSubElement` predicate (line 1-3). The operator `in` stands for set/relation inclusion, while dot (`.`) denotes relational join; for example `parent.subElements` results in a relation of the sets `SubElementName` and `Element`, such that for each of its tuples (`se`, `e`) the tuple (`parent`, `se`, `e`) is in `subElements`. Further, `parent.subElements` is prefixed by an outer join, the left operand of which is the entire domain (`univ`) of `SubElementName`. Consequently, the outer join yields a subset of `SubElement` (set being a special case of relation). Thus, the `isSubElement` predicate is satisfied if and only if `parent`, an element of `SubElementName`, and `child` are in the `subElements` relation.

Reminiscent of object-oriented notation, relational join can be also expressed by `[]` (resembling indexing); thus, `univ.(parent.subElement)` can be rewritten as `parent.subElements[univ]`, or even `subElements[parent][univ]`. Note that `[]` also indicates the arguments of a predicate (e.g., `isSubElement`); the interpretation of `[]` depends on a particular context.

## 6.2 Connector Theory in Alloy

In this section, we describe how a connector theory (Section 5) can be represented by means of Alloy. We focus on representing the meta-model of a connector theory (Fig. 13), describe a realization of its sets, and provide examples of how its constraints (Section 5) can be reflected in Alloy.

6.2.1 Representing the Meta-Model. Since the meta-model (Fig. 13) is based on sets and relations, it can be represented in Alloy in a straightforward way as illustrated below. For the purpose of the following examples we have modified the relations `features`, `attributes`, and `ports` so that their second and third fields are swapped (e.g., the domain of `features` becomes  $Element \times Port \times Signature$  instead of  $Element \times Signature \times Port$ ). The rationale for doing so is that this provides a more convenient Alloy syntax for expressing the constraints.

```

abstract sig DeploymentDock {}
abstract sig Role
abstract sig SubElementName {}
abstract sig UnitName {}
abstract sig Attribute {}
abstract sig AttributeValue {}
abstract sig Signature {}
abstract sig DistributionArchitecture {}

abstract sig Port {}
abstract sig LocalProvidedPort {}
abstract sig LocalRequiredPort {}
abstract sig RemotePort {}
abstract sig Feature {}
abstract sig FeatureValue {}
abstract sig ElementArchitecture {}

abstract sig RoleInstance {
  signature: one Signature,
  role: one Role
}
abstract sig Element {
  elementArchitecture: one ElementArchitecture,
  ports: Port set -> lone Signature,
  subElements: SubElementName lone -> lone SubElement,
  dock: one DeploymentDock,
  features: Feature set -> lone FeatureValue,
  attributes: Attribute set -> lone AttributeValue
}
sig Unit extends Element {
  roleInstances: RoleInstance
}
sig SubElement extends Element {}

```

```

one sig Connector {
  distributionArchitecture: one DistributionArchitecture,
  units: UnitName lone -> set Unit,
  connectorFeatures: Feature set -> lone FeatureValue
}

```

The definition of the connector theory meta-model includes the integrity constraint `ElementHierarchyIsTree` articulated in Section 6.1. Note that the definition of the sets and relations also includes detailed cardinality constraints.

6.2.2 Realizing Meta-Model Sets. Realization of the fixed sets by enumeration of all their elements is an inherent part of a connector theory. In Alloy, such an element is realized as a singleton set. As an example, the realization of `DeploymentDock` (Section 4.2.3) can take the following form:

```

one sig DockA extends DeploymentDock {}
one sig DockB extends DeploymentDock {}

```

The representation of `FeatureValues` is more complicated, since feature values may be hierarchical with parameters. In Alloy, we represent a parameter  $p$  by a relation  $p$  between two `FeatureValues`.

Moreover, in the context of an element architecture, the value of a parameter may be represented by an attribute and expected to be given later in the requirements specification (e.g., `fileName` in `LoggingToFile` in Section 4.1). This is expressed in Alloy by marking the associated signature by `abstract` (instead of `one`). This means that multiple concrete feature values are expected to be explicitly defined (each represented by a singleton subset), while the abstract signature defines their required structure. As an example, consider the `LoggingToFile` feature value defined in the `FileLogger` element architecture having the `fileName` parameter as its attribute.

```

abstract sig LoggingToFile extends FeatureValue {
  fileName: one FeatureValue -- feature value parameter
}

```

A signature for a concrete value explicitly defines the value of the parameters. As an example, consider the `LoggingToFile_InventoryLog` (Section 4.2), assigning `fileName` to “`inventory.log`” value (represented by `InventoryLog`).

```

one sig InventoryLog extends FeatureValue {} -- stands for “inventory.log”
one sig LoggingToFile_InventoryLog extends LoggingToFile {} {
  fileName = InventoryLog -- assignment of a particular parameter value
}

```

As for the alterable sets, the individual elements are created automatically by the Alloy Analyzer according to the corresponding constraints. Specifically for the `Signature` set, the created elements include concrete parameters of the signatures. The rationale is that while all the concrete values of the feature value parameters are explicitly given by the requirements specification, the concrete values of signature parameters depend on the actual composition and bindings of the elements, also created by the Alloy Analyzer. Technically, the Alloy representation of a parameterized signature is similar to the representation of a parameterized feature value – via a dedicated relation. The fact that signatures are to be created by the Alloy Analyzer is reflected by not marking the associated signature `abstract`. As an example, consider the `RMI` signature having the `javaSig` parameter (Section 4.1).

```

sig RMI extends Signature {
  javaSig: one Signature -- signature parameter
}

```

6.2.3 Representing Constraints (Examples for Element Architecture). First, the associated elements of the fixed sets have to be defined, including `Ports`, `SubElementNames`, `Features`, and `Attributes` sets; their elements are obtained by scanning the specification and the associated element type. For each element

architecture, a singleton subset of the ElementArchitecture set is created. Thus, for the FileLogger element architecture (Section 4.1), the following will be created:

```
-- Ports
one sig CallIn extends LocalProvidedPort {}
one sig CallOut extends LocalRequiredPort {}
-- Features
one sig Logging extends Feature {}
abstract sig LoggingToFile extends FeatureValue {
  fileName: one AttributeValue
}
-- Attributes
one sig FileName extends Attribute {}
-- Element architecture
one sig FileLogger extends ElementArchitecture {}
```

The constraint itself is a direct representation of the clause of the connector theory illustrated in Fig. 19. In Alloy, a clause is represented by a fact. Compared to Fig. 19, the fact syntax in Alloy differs slightly in terms of logical operators, sets, and application of predicates (in addition, the conjunction after each predicate is in Alloy represented by a new line). Thus, for the FileLogger element architecture, the following Alloy fact will be created:

```
fact FileLogger {
  for all e: Element | ElementHasEA[e,FileLogger] => {
    AllowedDeployment[e, DockA]
    AvailablePorts[e, CallIn + CallOut]
    SubElements[e, none]
    SupportedFeatures[e, Logging]
    RequiredAttributes[e, FileName]
    -- signature propagation
    PropagateSignature[e,CallIn,CallOut]
    -- enforcing fixed signatures
    -- enforcing values of the realized features
    FeatureHasValue[e, Logging, LoggingToFile]
    -- defining attributes referred in feature values
    FeatureValueUsesAttribute[e, Logging, fileName, FileName]
  }
}
```

As for predicates, to illustrate their representation consider the AvailablePorts predicate from Table 1. It is an example of a constraint on the second element of a ternary relation, where the third element is not relevant. Note, that we have replaced the test for inclusion in a relation with a relational join.

```
pred AvailablePorts[e: Element, availablePorts: set Port] {
  e.ports.univ = availablePorts
}
```

To illustrate the advantages of the Alloy language, below we show a representation of the FeatureValueUsesAttribute predicate from Table 1. Since a feature-value parameter is represented by a dedicated relation, we exploit the possibility of passing a relation as a parameter to an Alloy predicate. Here, relational join expresses the context of the constraint (e.g., that both feature value and attribute value are in the features and attributes relations with e).

```
pred FeatureValueUsesAttribute[e: Element, feature: Feature,
  parameter: FeatureValue -> AttributeValue, attr: Attribute] {
  e.features[feature].parameter = e.attributes[attr]
}
```

Compared to the corresponding formula from Table 1, it is obvious that the Alloy representation is much more concise and comprehensive, as it resembles an expression in a regular object-oriented programming language.

In a similar vein, `PortBinding` below is more concise than the corresponding predicate from Table 1. Here, the relational join allows expressing the context of the constraint transitively (combining the relations `subElements` and `ports`).

```
pred PortBinding[e: Element,
  subelement1: SubElementName, port1: LocalRequiredPort,
  subelement2: SubElementName, port2: LocalProvidedPort] {
  e.subElements[subelement1].ports[port1]
  = e.subElements[subelement2].ports[port2]
}
```

### 6.3 Selecting an Optimal CIC

The set of models of a connector theory can be of a large cardinality; nevertheless just a single model is needed. Thus, it is necessary to make a choice and select the “best” one (the best CIC). A practical necessity is to automate the selection process.

There are different criteria for judging what the “best” CIC is, ranging from memory consumption and CPU utilization, latency and throughput, to robustness, reliability, and stability, represented by means of valuation of the elements involved in CIC. Naturally, it is necessary to find rules for composability of the criteria. This leads to an optimization problem where the task is to find an optimal CIC given a valuation of its elements (their element architectures in particular) by applying the rules. A simple example of such a valuation is a manual assignment of a fixed cost to each element architecture where the valuation of CIC is the sum of costs for all the elements in it. The CIC with the lowest cost is pronounced the “best”.

The Alloy language itself does not provide any explicit support for solving optimization problems. In principle, there are three possible approaches to employing the Alloy framework for finding an optimal CIC: (i) to enumerate all the models of a given connector theory and select an optimal one by applying valuation criteria outside the resolution process (an enumeration of all models is a standard feature of the Alloy Analyzer); (ii) to encode the optimization problem into a “standard” Alloy theory; and (iii) to extend the Alloy language and Alloy Analyzer accordingly.

We do not consider other approaches, such as extending the Alloy Analyzer without changing the Alloy language (i.e., instrumenting the Analyzer with custom heuristics), or moving from Alloy to another constraint-solving technique. Preferring relative simplicity and efficiency, we have used approach (i) for the experiments and evaluation.

**6.3.1 Enumerating All Connector Theory Models.** A standard feature of the Alloy Analyzer is the enumeration of all models of a given Alloy theory by incremental execution of its underlying SAT solver. This feature can be effectively exploited for finding an optimal CIC. Here, the Alloy Analyzer finds all the models of a given connector theory and a valuation of the models is provided by a dedicated external tool (i.e., a partial order upon the models is created); finally an optimal CIC is determined based on the partial order.

By delegating the optimization to a dedicated external tool, this method does not require expressing optimization criteria in CT. Consequently, the model valuation strategy is not limited by the expressive power of the Alloy language, but it is entirely determined by the options the external tool provides.

Even though it is necessary to explore all the models, this method is still tangible, since each of the models is evaluated separately.

**6.3.2 Encoding Optimization Problems in Alloy.** Modeling optimization problems in Alloy is based on expressing a partial order over the set of CICs where the “best” CIC is the least element. Since the order of a particular CIC is typically determined by the employed element architectures, it is necessary to define a global partial order of all the available element architectures, as well as a way of inferring the CIC partial order from this global order.

Defining such a way of inferring is a challenge. Since the use of natural numbers provided by Alloy is not feasible due to the increased complexity, the standard Alloy relations have to be used. Here, it is necessary to assess the order of composite element architectures based on the order of their sub-element architectures, which is also a challenge. Another option is also to employ the state-of-the-art alternatives for solving Alloy theories [EGT11, EGT10], that offer better support for arithmetic and unbounded integer types.

**6.3.3 Extending the Alloy Framework.** The Alloy language could be extended to allow for specifying optimization criteria. The Alloy Analyzer would have to translate such optimization criteria into a SAT formula. Specifically, such an extension can be achieved by employing pseudo-boolean (PB) formulas instead of SAT formulas. Here, an assumption is that the Alloy Analyzer supports a PB solver such as MiniSAT+ [ES06] (this appears to be realistic in the future, since the MiniSAT solver is already supported by Alloy Analyzer). Overall, such an extension of the Alloy framework requires a major modification of its current implementation.

## 7. RELATED WORK

In general, the related work spans three areas: (i) composing software with the help of constraint solving, (ii) Alloy-based resolution and verification of component architectures, and (iii) automated connector synthesis.

(i) **Composing Software with the Help of Constraint Solving.** Constraint solving techniques have been already used for a variety of tasks in software composition ranging from automated dependency management [LBP08] to verification of composition in product lines [TBKC07]. In [LBP08] the idea is to employ a SAT solver to resolve dependencies; this approach was used in several contemporary software tools such as OSGi implementation Equinox p2 and Maven<sup>3</sup>.

The approach presented in [TBKC07] is based on formalizing the notion of a feature model by introducing a specific feature algebra. Having such formalization available, a SAT solver is employed for verification of safe feature-model composition in a product line.

Compared to ARCAS, both methods leverage on simple propositional formulas for expressing the given problem. Such formulas merely express variability and transitivity, but do not reflect more complex properties of the software parts to be composed (e.g., interface bindings, runtime environment requirements, and features).

(ii) **Alloy-based Resolution and Verification of Component Architectures.** Alloy has been already extensively used in the domain of CBSE for the purpose of both property checking and model finding. In [GMK02], the authors examine the feasibility of using architectural constraints as the basis for specification, design, and implementation of self-organizing architectures in Darwin. In this context, Alloy serves as a tool for an automated resolution of the self-organizing reconfigurations from an inconsistent to a consistent state in terms of a particular architectural style.

In a similar way, [HI10] employs Alloy for specification of the possible architecture reconfiguration actions in the context of a generic component model based on OSGi. Alloy is employed in two ways: (a) architecture change verification and (b) architectural change planning. The former focuses on soundness of the reconfiguration actions and preservation of the properties specific to a particular architectural style. The latter comprises finding a fitting sequence of reconfiguration actions from the current consistent state to a given consistent state while preserving a particular architectural style in all intermediate states.

A methodology for verifying soundness of self-configuration scenarios via their

---

<sup>3</sup> <http://www.eclipse.org/equinox/p2/> and <http://maven.apache.org/>

specification in Alloy is presented in [TMS10]. The underlying formal method – FracToy – formalizes a concrete self-configurable system, as well as the corresponding self-organizing actions. The verification targets both static and dynamic properties. Compared to our approach, models of the Alloy theory only prove the consistency/soundness of the theory and are not used for any other purpose.

In [KG06], Alloy is employed for the formal specification of various architectural styles. The main goal is to check the important properties of an architectural style such as consistency, satisfaction of a predicate over an architectural style instance, composability, and refinement of architectural styles. The Alloy formalization of an architectural style is obtained programmatically from its Acme specification (which serves a similar purpose as our CDL specification).

Apart from reconfigurations, in [JS00] a formal specification of valid component compositions in the COM component model is analyzed, while in [MS08] a fully-fledged Alloy formalization of the Fractal component model is presented.

(iii) Automated Connector Synthesis. The ARCAS method stems from our previous research [BP04, BB05, B06, BHP06]. In [BP04, B06], a connector model designed for automated connector synthesis based on a high-level specification is presented. Despite leveraging on similar concepts to those presented in Section 2, it does not explicitly capture NFPs. The key difference lies in the way a connector configuration is selected - it is performed via term matching in Prolog. Breaking the separation of abstraction levels, the method imposes the inclusion of Prolog terms directly in the requirements specification.

An extensive effort has been put into research of automated synthesis of connectors ensuring application-layer and middleware-layer interoperability [IBB11]. For brevity, we call the former API mediation and the latter middleware bridging. While we have focused on cases where a component/service being deployed is to be connected to another one (potentially already deployed and running), [IBB11] exploits a slightly different scenario – connecting solely the already deployed and running components/services. The connected components may require both API mediation and middleware bridging. Thus, letting NFPs aside, in [IBB11] the input is an API specification and a determination of a particular middleware for each of the connected components; the goal is to synthesize (in an automated way) a connector implementation which does the mediation and/or bridging. For comparison, in ARCAS the input is a deployment and requirements specification; the goal is to find a fitting connector implementation employing a suitable middleware/middleware bridging (a potential API mediation is expected to be done by a separate component).

In [RRSGWT05], the concept of a connector is introduced for CCM. The goal is to benefit from a light communication middleware – lighter than CORBA – in CCM-based applications by introducing connectors (strictly separating the component communication from business logic). The connector-generation process employs an extended IDL for defining connector templates. These are however not composable. Similar to ARCAS, connector specification is accompanied by a description of connector-specific features in an extended OMG D&C specification. Connectors are generated by manually selecting a particular template parameterized by the actual interfaces; the connector templates have also to be created manually.

In [RCGT09], targeting model-based generation of component-based applications from UML-MARTE models, the concept of compositional connectors is also employed. A connector's architecture has to be explicitly captured (manually) by an UML-MARTE model. The abstractions describing a connector's structure are similar to those presented in Section 2.

## 8. DISCUSSION AND EVALUATION

In this section, we discuss the important decisions and trade-offs we had to make, and also mention open issues we still face. In particular, these include (i) interpretation of

a connector theory model, (ii) focus of a connector theory, (iii) addressing NFPs, (iv) finding of an optimal connector theory model, (v) experience and case study, (vi) moving ARCAS to other domains, and (vii) features to be addressed.

(i) Model Interpretation. The ARCAS method exploits the Alloy Analyzer's capability of finding a model of a given Alloy theory. In contrast to a typical usage of this feature – interpreting the existence of a model as consistency/soundness of the theory and using such a model as feedback while developing a theory, e.g., [ABGR08, TMS10] – the ARCAS method directly interprets/employs a model as the desired CIC.

(ii) Focus of the Theory. The connector theory in ARCAS differs from similar formal models [MS08] in its focus. While in [MS08] focuses on the meta-model level, ARCAS concentrates on the model level by specifying semantics of a particular connector via concrete instances of meta-model entities (e.g., `FileLogger` as an instance of element architecture). This is facilitated by the automated transformation of the connector specification into a connector theory, whereas formal models at the meta-model level are typically created manually. The ARCAS method thus provides the option of reasoning about a particular connector instead of reasoning solely about properties common to all possible connectors.

(iii) Addressing NFPs. The ARCAS method makes it possible to address NFPs in the synthesized connector. This is in contrast to some of the state-of-the-art methods for automated synthesis of middleware-based connectors [IBB11]. Since connector theory also comprises element composability with respect to NFPs, it allows programmatic synthesis of a connector complying with the NFP requirements. Nevertheless, since the Alloy Analyzer lacks a specialized support for arithmetic operators, it is feasible to address only qualitative NFPs (i.e., those expressible by enumeration), whereas efficient addressing of quantitative NFPs (such as latency) would be hard to achieve. However, there are constraint-solving techniques providing advanced support for arithmetic operations and working with inequalities [DMB08], which might be sufficient for addressing quantitative NFPs. In this respect, a challenging issue is to integrate these constraint-solving techniques with the Alloy framework, or adapt the ARCAS method into another framework supporting such techniques.

(v) Moving ARCAS to Other Domains. Although the ARCAS method has been presented as having its principal application in automated resolution of middleware-based connectors, the method itself is generic and applicable in other domains. In general, ARCAS is applicable in a domain with the following key characteristics: (i) it employs hierarchical component-based architectures, (ii) if NFPs are required, they can be articulated in a composable way, (iii) the related specifications can be transformed to a theory (i.e., a constraint specification) programmatically, (iv) the theory can be articulated in such a way that automated model-finding is possible, and (v) criteria for selecting an optimal model of a theory can be formed.

Specifically, ARCAS universally applies to architectural patterns based on pipe-and-filter (including their nesting). A typical example illustrating such a potentially nested pattern is a media player. Such a player employs a number of codecs (audio and video), filters, muxers, and demuxers, which have to be correctly organized in an architecture in order to process the content from an input stored in a file or available on-line. From the ARCAS perspective, the whole architecture of a media player can be likened to a connector, and each of the codecs, filters, muxers, and demuxers can be likened to a connector element. The key property of ARCAS, applicable in this scenario, is the automated assembly of the elements while ensuring their compatibility and meeting requirements on NFPs. Thus, it is possible, for example, to handle the tradeoff between computational complexity and video image quality.

Another domain where ARCAS can be applied is the domain of embedded real-

time systems. Here, ARCAS can provide an automated selection of hardware sensors, corresponding device drivers, and proper API of the operating system. In analogy with a connector, each sensor, device driver, and particular API variant can be likened to an element. The option of specifying NFPs can be employed, e.g., for determining the required sampling rate of a sensor.

(vi) Experience and Case Study. As a proof-of-the-concept, we have developed an EMF<sup>4</sup>-based demonstrator<sup>5</sup> involving the transformations of specifications and integrating Alloy Analyzer. In addition, we have developed an experimental database of connector artifacts (including both the specifications and their Alloy images). The examples from this text are simplified versions of the artifacts in this database. We have employed this database in a case study involving a non-trivial part of a real-life component-based application based on the procedure-call communication style [HKW08]. Various client-server connection scenarios differing in NFPs and deployment were considered. This helped demonstrate the soundness and feasibility of the Alloy-based CIC resolution on a real-life example.

We have also performed several benchmarks in order to assess the performance scaling factors of ARCAS. For this purpose, the actual database of connector artifacts was generated in an automated way by cloning and introducing new variants in the original database. This technique closely mimics a general case, since the performance of Alloy Analyzer mainly depends on the cardinality of the sets and relations rather than on the complexity and variability of the constraints. Based on our measurements, even though the computational complexity is exponential in principle (Alloy Analyzer employs a SAT solver), ARCAS is feasible up to hundreds of element architectures and tens of distribution architectures. Specifically, for 100 element architectures and 10 distribution architectures the execution times are in the order of 5 seconds (Alloy set up for MiniSAT and 512MB on Intel i5 2.6 GHz); for 200 element architectures and 20 distribution architectures the execution time is around 14 seconds. The performance can be further improved by representing connector theory directly in Kodkod [T09] (the underlying relational solver) instead of in Alloy. Note that the numbers above (100/200) pertain just to the element architectures and distribution architectures that are applicable for a particular connector (these architectures were selected from a much larger database).

(vii) Issues to be Addressed. The ARCAS meta-model does not reflect (a) cardinality of ports and sub-elements (e.g., important when multiple client stubs have to be served by a single server skeleton), and (b) composite port signatures (a typical phenomenon when a server unit supports a number of middleware protocols simultaneously). In this paper, these concepts were left out for simplicity, but the ARCAS method can be enhanced to support both (a) and (b). Addressing (a) comprises an extension to the element type and element architecture abstract syntax, as well as straightforward modifications of the transformation of element architectures and distribution architectures. Addressing (b) involves modifications of the connector theory and associated transformations. As an aside, both (a) and (b) have been already experimentally proved feasible.

Since the behavior of middleware-based connectors is rather simple and driven by the communication style, the responsibility of the behavioral compliance between element types and element architectures is upon the elements' designer.

## 9. CONCLUSION AND FUTURE WORK

In this paper, we presented a method for automated resolution of connector

---

<sup>4</sup> <http://www.eclipse.org/modeling/emf/>

<sup>5</sup> [http://d3s.mff.cuni.cz/projects/components\\_and\\_services/arcas/](http://d3s.mff.cuni.cz/projects/components_and_services/arcas/)

architectures based on constraint solving – the ARCAS method. An important benefit of ARCAS is the ability to address the required NFPs, which, as well as transparent distribution, is the major concern of middleware-based connectors. In ARCAS, we assume middleware connectors are based on hierarchical elements, similar to hierarchical components. This allows definition of the individual parts of a connector in advance, and thus facilitates reuse. The key idea of ARCAS is to resolve a description of a particular connector instance (CIC) as a model of a theory based on a first-order logic and relational calculus – a connector theory. We have defined automated transformations, which convert the predefined connector artifact, requirements, and deployment specifications to such a connector theory. Overall, ARCAS can be employed whenever the requirements or deployment changes (even at runtime). Moreover, we have articulated the characteristics another domain would have to satisfy to make ARCAS applicable.

As a proof-of-the-concept, we described the representation of a connector theory in the Alloy modeling language. Moreover, by using the Alloy representation, we have shown the feasibility of ARCAS on a real-life example. We have also developed a demonstrator involving the transformations of specifications and integrating Alloy Analyzer.

In our future work, we intend to focus on providing support for modeling optimization problems in Alloy, as well as on introducing support for quantitative NFPs (either by extending the Alloy framework or employing another constraint solver framework). Finally, we plan to explore the possibility of integrating state-of-the-art methods for middleware and application interoperability [IBB11, IST11, BPGG11] to achieve a resolution-based synthesis of emergent connectors.

## REFERENCES

- [ABGR08] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, “On challenges of model transformation from UML to Alloy,” *Software & Systems Modeling*, vol. 9, no. 1, pp. 69-86, Dec. 2008.
- [B06] T. Bures, “Generating Connectors for Homogeneous and Heterogeneous Deployment”, *PhD dissertation*, Dept. of Distributed and Dependable Systems, Charles University in Prague, 2006.
- [BB04] L. Bulej, T. Bures, “Addressing Heterogeneity in OMG D&C-based Deployment”, Tech. Report No. 2004/7, Dep. of SW Engineering, Charles University, Prague, <http://d3s.mff.cuni.cz/publications/No2004>.
- [BB05] L. Bulej, T. Bureš: “Deploying Heterogeneous Applications using OMG D&C and Software Connectors”, Tech. Report No. 2005/10, Dep. of SW Engineering, Charles University, Prague, <http://d3s.mff.cuni.cz/publications/>, Nov 2005.
- [BG07] S. Benmokhtar, N. Georgantas, and V. Issarny, “COCOA: CONversation-based service COMposition in pervAsive computing environments with QoS support,” *Journal of Systems and Software*, vol. 80, Dec. 2007, p. 1941–1955.
- [BHP06] T. Bures, P. Hnetynka, and F. Plasil, “SOFA 2: Balancing Advanced Features in a Hierarchical Component Model”, Proc. of 4<sup>th</sup> International Conference on Software Engineering Research, Management and Applications (SERA '06), IEEE Computer Society, Washington, DC, USA, 2006.
- [BP04] T. Bures and F. Plasil, “Communication style driven connector configurations,” in *Software Engineering Research and Applications*. Springer Berlin/Heidelberg, 2004.
- [BPGG11] G. Blair, M. Paolucci, P. Grace, N. Georgantas: “Interoperability in complex distributed systems.” In: Bernardo, M., Issarny, V. (eds.) *SFM 2011*. LNCS, vol. 6659, Springer, Heidelberg (2011)
- [BS07] S. Bliudze and J. Sifakis. “The algebra of connectors: structuring interaction in BIP”. In Proc. of *EMSOFT '07*. ACM, New York, NY, 11-20, 2007.
- [CCP11] J. Cubo, C. Canal, and E. Pimentel, “Context-Aware Composition and Adaptation based on Model Transformation,” *The Journal of Universal Computer Science*, vol. 17, 2011, pp. 777-806.
- [CL02] I. Crnkovic, M. Larsson: “Building Reliable Component-Based Software Systems.” Artech House, Inc., Norwood, MA, USA, 2002.
- [DMB08] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, p. 337–340.
- [EGT10] A.A. El Ghazi and M. Taghdiri, “Analyzing Alloy Constraints using an SMT Solver: A Case Study,” *5th International Workshop on Automated Formal Methods (AFM)*, Edinburgh, UK, 2010.
- [EGT11] A. El Ghazi and M. Taghdiri, “Relational Reasoning via SMT Solving,” *FM 2011: Formal Methods*, 2011, p. 133–148.

- [ES06] N. Eén and N. Sörensson, “Translating pseudo-boolean constraints into SAT,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, 2006, p. 1–26.
- [GMK02] I. Georgiadis, J. Magee, and J. Kramer, “Self-organising software architectures for distributed systems,” *Proceedings of the first workshop on Self-healing systems*, New York, USA, 2002.
- [HI10] K.M. Hansen and M. Ingstrup, “Modeling and analyzing architectural change with alloy,” *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, 2010, p. 2257.
- [HKW08] S. Herold, H. Klus, Y. Welsch, et al., “CoCoME-The Common Component Modeling Example”, *The Common Component Modeling Example*, 2008, p. 16–53.
- [IBB11] V. Issarny, A. Benameur, and Y.D. Bromberg, “Middleware-layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability,” *Formal Methods for Eternal Networked Software Systems*, M. Bernardo and V. Issarny, eds., Berlin / Heidelberg: Springer, 2011, pp. 217-255.
- [ISJB09] V. Issarny, B. Steffen, B. Jonsson, G. Blair, P. Grace, M. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, and A. Sabetta, “CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems,” *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, 2009, pp. 154-161.
- [IST11] P. Inverardi, R. Spalazzese, and M. Tivoli, “Application-layer connector synthesis,” *Formal Methods for Eternal Networked Software Systems*, 2011, p. 148–190.
- [J02] D. Jackson, “Alloy: a lightweight object modelling notation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, 2002, p. 256–290.
- [J06] D. Jackson: *Software Abstractions: Logic, Language, and Analysis.* MIT Press, Cambridge, MA, USA, and London, England, 2006.
- [J11] D. Jackson: “Alloy Language Reference”, <http://alloy.mit.edu>, 2011.
- [JS00] D. Jackson and K. Sullivan, “COM revisited: tool-assisted modelling of an architectural framework,” *ACM SIGSOFT Software Engineering Notes*, vol. 25, 2000, p. 149–158.
- [KBPH12] J. Keznikl, T. Bureš, F. Plášil, P. Hnětynka: “Automated Resolution of Connector Architectures Using Constraint Solving (ARCAS method)”, Tech. Report No. D3S-TR-2012-01, Dep. of Distributed and Dependable Systems, Charles University, <http://d3s.mff.cuni.cz/publications/>, 2012
- [KG06] J.S. Kim and D. Garlan, “Analyzing architectural styles with alloy,” *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis - ROSATEA '06*, 2006.
- [LBP08] D. Le Berre, A. Parrain: “On SAT Technologies for Dependency Management and Beyond.” *Proceedings of 12th International Software Product Line (SPLC 2008)* vol. 2, 2008, p. 197-200.
- [MMP00] N. R. Mehta, N. Medvidovic, and S. Phadke: “Towards a taxonomy of software connectors,” in *Proceedings of the 22nd international conference on Software engineering*. ACM, 2000.
- [MPBH11] M. Malohlava, F. Plášil, T. Bureš, P. Hnětynka: “Interoperable DSL Families for Code Generation,” Tech. Report No. D3S-TR-2011-04, Dep. of Distributed and Dependable Systems, Charles University, Prague, <http://d3s.mff.cuni.cz/publications/>, Apr 2011
- [MS08] P. Merle and J.B. Stefani, “A formal specification of the Fractal component model in Alloy,” Research Report RR-6721, INRIA, <http://hal.inria.fr/inria-00338987/en/>, 2008.
- [NTER06] J. Nakazawa, H. Tokuda, W.K. Edwards, and U. Ramachandran, “A Bridging Framework for Universal Interoperability in Pervasive Systems,” *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, 2006, pp. 3-3.
- [OMG04] Object Management Group, “Deployment and Configuration of Component-based Distributed Applications Specification”, <http://www.omg.org/cgi-bin/doc?formal/06-04-02.pdf>, Feb 2004
- [RCGT09] A. Radermacher, A. Cuccuru, S. Gerard, and F. Terrier, “Generating execution infrastructures for component-oriented specifications with a model driven toolchain: a case study for MARTE’s GCM and real-time annotations,” *Proceedings of the eighth international conference on Generative programming and component engineering*, ACM, 2009, p. 127–136.
- [RRSGWT05] S. Robert, A. Radermacher, V. Seignole, S. Gérard, V. Watine, and F. Terrier, “Enhancing interaction support in the corba component model”, *From Specification to Embedded Systems Application*, 2005, p. 137–146.
- [SI10] R. Spalazzese, P. Inverardi: “Mediating Connector Patterns for Components Interoperability”. In: *Babar, M.A., Gorton, I. (eds.) ECSA 2010*. LNCS, vol. 6285, pp. 335–343. Springer, Heidelberg (2010)
- [TBKC07] S. Thaker, D. Batory, D. Kitchin, W. Cook: “Safe composition of product lines.” *Proceedings of the 6th international conference on Generative programming and component engineering*, ACM, 2007, p. 95–104.
- [TMD10] R.N. Taylor, N. Medvidovic, E.M. Dashofy: “Software architecture: foundations, theory, and practice.” Wiley, Hoboken, 2010.
- [TMS10] A. Tiberghien, P. Merle, and L. Seinturier: “Specifying Self-configurable Component-Based Systems with FracToy,” *Abstract State Machines, Alloy, B and Z*, vol. 5977, 2010, p. 91–104.
- [T09] E. Torlak: “A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications.” Ph.D. Thesis, MIT, February 2009