A Type Theory for Software Architectures

Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor

Technical Report UCI-ICS-98-14

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
{neno,dsr,taylor}@ics.uci.edu

April 1998

ABSTRACT

Software architectures have the potential to substantially improve the development and evolution of large, complex, multi-lingual, multi-platform, long-running systems. However, in order to achieve this potential, specific architecture-based modeling, analysis, and evolution techniques must be provided. This paper motivates and presents one such technique: a type theory for software architectures, which allows flexible, controlled evolution of software components in a manner that preserves the desired architectural relationships and properties. Critical to the type theory is a taxonomy that divides the space of subtyping relationships into a small set of well defined categories. The paper also investigates the effects of large-scale development and off-the-shelf reuse on establishing type conformance between interoperating components in an architecture. An existing architecture is used as an example to illustrate a number of different applications of the type theory to architectural modeling and evolution.[1]

1. INTRODUCTION

In order for large, complex, multi-lingual, multi-platform, long-running systems to be economically viable, they need to be evolvable. Support for software evolution includes techniques and tools that aid interchange, reconfiguration, extension, and scaling of software modules and/or systems. Evolution in the current economic context also requires support for reuse of third-party components. The costs of system maintenance (i.e., evolution) are commonly estimated to be as high as 60% of overall development costs [7]. Practitioners have traditionally faced many problems with curbing these costs. The problems are often the result of poor understanding of a system's overall architecture, unintended and complex dependencies among its components, decisions that are made too early in the development process, and so forth. Traditional development approaches (e.g., structural programming or object-oriented analysis and design) have in particular failed to properly decouple computation from communication within a system, thus supporting only limited reconfigurability and reuse. Evolution techniques have also typically been programming language (PL) specific (e.g., inheritance) and applicable on the small scale (e.g., separation of concerns or isolation of change). This is only partially adequate in the case of development with preexisting, large, multi-lingual, multi-platform components that originate from multiple sources.

An explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large systems. Software architectures present a high level view of a system, enabling developers to abstract away the irrelevant details and focus on the "big picture." Another key property is their explicit treatment of software connectors, which separate communication issues from computation in a system. However, existing architecture research has thus far largely failed to take advantage of this potential for adaptability: few specific techniques have been developed to support flexible architecture-based design and evolution.

Three distinct building blocks of a software architecture are components, connectors, and architectural configurations (topologies) [16]. Each of them may evolve. Our work to date has addressed the evolution of connectors and topologies [14, 15, 19, 25]. This paper proposes a technique for evolving software components. This technique has resulted from the recognition that researchers in software architectures, and particularly in architecture description languages (ADLs), can learn from extensive experience in the area of PLs, and object-oriented languages (OOPLs) in particular. The particular lesson in this case is that an existing software module can evolve in a controlled manner via subtyping.

Our approach to component evolution is indeed based on a type theory. We treat each component specification in an architecture as a type and support its evolution via subtyping. However, while PLs (and several existing ADLs [5, 6, 10]) support a single subtyping mechanism, we have demonstrated that architectures may require multiple subtyping mechanisms, many of which are not commonly supported in PLs [13]. Therefore, existing PL type theories are inadequate for use in software architectures.

Beyond evolution, types are also useful in establishing certain correctness criteria about a program or an architecture. Several existing ADLs support type checking (e.g., Aesop [5], Darwin [12], Rapide [10], and UniCon [23]). However, as with most all of the existing PLs, these ADLs essentially establish simple syntactic matches among interacting components. Our approach also establishes semantic conformance of components.

Furthermore, all existing type checking mechanisms regard types as either compatible or incompatible. Although it is beneficial to characterize component compatibility in this way, determining the *degree* of compatibility, and thus the potential for component interoperability, is more useful. One of the goals of the software architecture and component-based-development communities is to provide more extensive support for building

1. Effort partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-97-2-0021 and F30602-94-C-0218, and by the Air Force Office of Scientific Research under grant number F49620-98-1-0061. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973.

systems out of existing parts. Those parts will typically not perfectly conform to each other. We have demonstrated that partially mismatched components can in certain cases still be effectively combined in an architecture [14, 15]. Establishing the degree of compatibility can also help determine the amount of work necessary to retrofit a component for use in a system.

The contributions of this paper are threefold:

- a taxonomy that divides the space of potentially complex subtyping relationships into a small set of well defined, manageable subspaces;
- a flexible type theory for software architectures that is domain-, style-, and ADL-independent. By adopting a richer notion of typing, this theory is applicable to a broad class of design and reuse circumstances; and
- an approach to establishing type conformance between interoperating components in an architecture. This approach is better suited to support the “large scale development with off-the-shelf reuse” philosophy on which architecture research is largely based than other existing techniques.

The remainder of this paper is organized as follows. Section 2 briefly discusses the architecture that is a basis of examples used throughout the paper to illustrate the concepts of the type theory. Section 3 introduces and discusses the general principles of the architectural type theory. Section 4 formally defines a particular instance of the type theory, or a type system, that exhibits the necessary properties for component evolution and architectural type checking. A discussion of our results to date, conclusions, and future work round out the paper.

2. EXAMPLE ARCHITECTURE

To illustrate the concepts in this paper, we use the architecture shown in Fig. 1. This architecture resulted from the case study conducted during the Second International Software Architecture Workshop (ISAW-2) [26]. The system that the architecture models is a *call center customer care* (C4) system for a large telecommunications company. The architecture includes several subsystems identified by the telephone company:

- Corporate databases and billing — customer account management;
- Network Operations Support System (NOSS) — management and provisioning of the physical network;
- Downstream systems — e.g., long-distance carrier services, 911 service, voice mail, and so forth;
- “Quick” service — enables customers to directly communicate with the system; and
- “C4 core” — manages the above parts of the system and provides support for service negotiations, account management, and trouble-call management.

The architecture in Fig. 1 addresses most of the major system requirements and has previously been discussed in more detail [27]. It is modeled in the C2 architectural style: a component communicates with components above and below it in the architecture by sending asynchronous messages, which are then routed to the appropriate components by connectors [25]. Although some concerns have been voiced about the suitability of the C2 style for this particular application, the specifics of the style and of the architecture are not critical for the purpose of demonstrating the concepts introduced in this paper. Instead, we chose this particular example because it has well-defined requirements from an actual project and describes a large-scale problem, for which software architectures are particularly well suited.

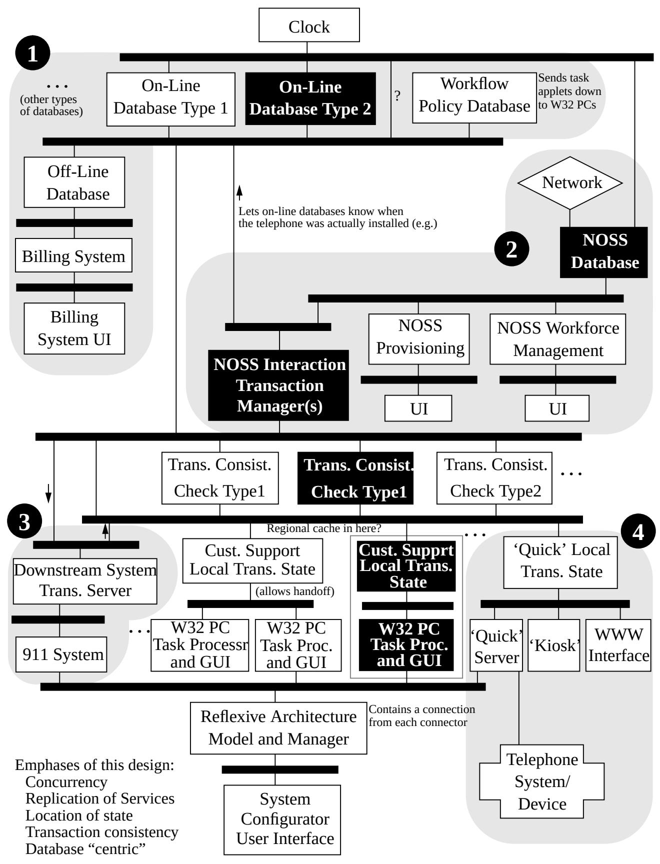
Clock
1
. . .
(other types of databases)
On-Line Database Type 1
On-Line Database Type 2
?
Workflow Policy Database
Sends task applets down to W32 PCs
Off-Line Database
Billing System
Billing System UI
Lets on-line databases know when the telephone was actually installed (e.g.)
Network
2
NOSS Database
NOSS Interaction Transaction Manager(s)
NOSS Provisioning
NOSS Workforce Management
UI
UI
Trans. Consist. Check Type1
Trans. Consist. Check Type1
Trans. Consist. Check Type2
. . .
3
Regional cache in here?
. . .
4
Downstream System Trans. Server
Cust. Support Local Trans. State
(allows handoff)
Cust. Supprt Local Trans. State
‘Quick’ Local Trans. State
911 System
. . .
W32 PC Task Processr and GUI
W32 PC Task Proc. and GUI
W32 PC Task Proc. and GUI
‘Quick’ Server
‘Kiosk’
WWW Interface
Reflexive Architecture Model and Manager
Contains a connection from each connector
Telephone System/ Device
Emphases of this design:
Concurrency
Replication of Services
Location of state
Transaction consistency
Database “centric”
System Configurator User Interface

Fig. 1. Call Center Customer Care (C4) System architecture:
(1) Corporate databases and billing;
(2) Network Operations Support System (NOSS);
(3) Downstream systems;
(4) “Quick” service.
Highlighted components have been modeled extensively.

We have extensively modeled the components highlighted in Fig. 1. These components were selected because they constitute a logical subsystem (basic customer service request handling) and exhibit interesting properties. The examples used in the remainder of the paper will be drawn from this modeling effort.

3. GENERAL PRINCIPLES OF THE TYPE THEORY

Explicit treatment of types enables *subtyping*, the evolution of a given type to satisfy new requirements, and *type checking*, the determination of whether instances of one type may be legally used in places where another type is expected. This notion of legality can help software developers keep program semantics close to programmer intentions, and thus discipline the evolution and (re)use of objects. Furthermore, a combination of type declarations and type checking supports source code understandability and, ultimately, the generation of efficient executable code.

A useful overview of PL subtyping relationships is given by Palsberg and Schwartzbach [21]. They describe a consensus in the OO typing community regarding the definition of a range of OO typing relationships. *Arbitrary subclassing* allows any class to be declared a subtype of another, regardless of whether they share a common set of methods. *Name compatibility* demands that there exist a shared set of method names available

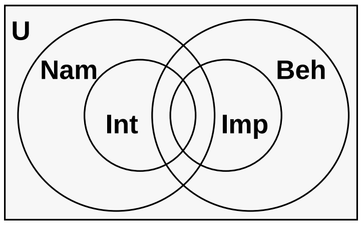

Fig. 2. A framework for understanding OO subtyping relationships as regions in a space of type systems.

in both classes. *Interface conformance* constrains name compatibility by requiring that the shared methods have conforming signatures. *Monotone subclassing* requires that the subclass relationship be declared and that the subclass must preserve the interface of the superclass, while possibly extending it. *Behavioral conformance* [2, 3, 11, 29] allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the supertype. Finally, *strictly monotone subclassing* additionally demands that the subtype preserve the particular implementations used by the supertype.

Protocol conformance goes beyond the behavior of individual methods to specify constraints on the order in which methods may be invoked. Explicitly modeling protocols has been shown to have practical benefits [1, 9, 18, 28, 29]. However, component invariants and method preconditions and postconditions can be used to describe all state-based protocol constraints and transitions. Thus, our notion of behavioral conformance implies protocol conformance, and we do not address them separately.

We have developed a framework for understanding these subtyping relationships as regions in a space of type systems, shown in Fig. 2. The entire space of type systems is labeled *U*. The regions labeled *Int* and *Beh* contain systems that demand that two conforming types share interface and behavior, respectively. The *Imp* region contains systems that demand that a type share particular implementations of all supertype methods, which also implies that types preserve the behavior of their supertypes. The *Nam* region demands only shared method names, and thus includes every system that demands interface conformance.

Each subtyping relationship described in [21] and summarized above can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a type be preserved, corresponds to the intersection of the Int and Beh regions and is expressed as *int* **and** *beh* (Fig. 3b). Each region in Fig. 2 encompasses a set of variations of a given subtyping relationship, rather than a single relationship. Thus, for example, the different flavors of the behavioral conformance relationship, described by Zaremski and Wing [29], represent different points in the *int* **and** *beh* subspace. The architectural type system we propose in the next section also represents a selection of individual points within the different subspaces.

This type theory was motivated by our previous work, where we have encountered numerous situations in which new components were created by preserving one or more aspects of one or more existing components [13, 15]. Several examples are shown in Fig. 3:

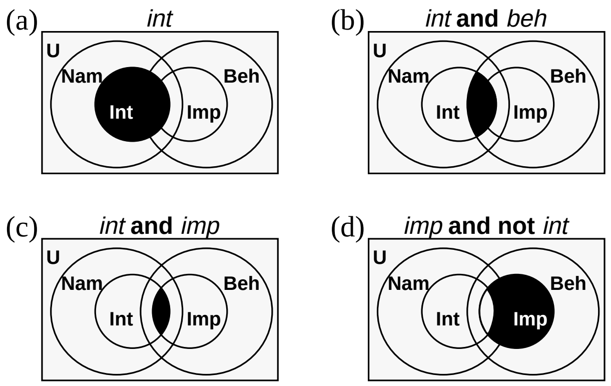

Fig. 3. Examples of component subtyping relationships we have encountered in practice.

- *interface conformance* (Fig. 3a) has proven useful in interchanging components that communicate via asynchronous message passing (e.g., C2 architectural style [25]), without affecting dependent components in a given architecture;
- *behavioral conformance* (Fig. 3b) guarantees correctness during component substitution;
- *strictly monotone subclassing* (Fig. 3c) enables extension of the behavior of an existing component while preserving correctness relative to the rest of the architecture;
- *implementation conformance with different interfaces* (Fig. 3d) allows a component to be fitted into an alternate domain of discourse (e.g., by using software adaptors [28]);
- *multiple conformance mechanisms* allow creation of a new type by subtyping from several types using different subtyping mechanisms.

Note that we referred to the first three examples (Fig. 3a-c) using the terminology from the Palsberg-Schwartzbach taxonomy. However, while in OOPLs the three subtyping mechanisms would be provided by three separate languages, in architectures they all need to be supported by the same ADL and may actually be applied to components in a single architecture. Also, the example in Fig. 3d does not have a corresponding OOPL mechanism, further motivating the need for a flexible type theory for software architectures.

At the same time, by giving a software architect more latitude in choosing the direction in which to evolve a component, we allow some potentially undesirable side effects. For example, by preserving a component's interface, but not its behavior, the component and its resulting subtype may not be interchangeable in a given architecture. However, it is up to the architect to decide whether to preserve architectural type correctness, in a manner similar to America [2], Liskov and Wing [11], Leavens et al. [3], and others (depicted in Fig. 3b), or simply to enlarge the palette of design elements in a controlled manner, in order to use them in the future.

4. ARCHITECTURAL TYPE SYSTEM

In [13] we discussed the types of syntactic constructs needed in an ADL in order to support our type theory. In this section we present a type system for software architectures that instantiates the type theory. The two possible applications of an architectural type theory—evolution of existing components by software architects, and type checking of architectural descriptions—are discussed below in Sections 4.2 and 4.3, respectively. All definitions are specified in Z, a language for modeling mathematical objects based on first order logic and set theory [24]. Z uses standard logical connectives ($\vee$, $\wedge$, $\Rightarrow$, etc.) and set-theoretic operations ($\mathbb{P}$ to denote sets, $\in$, $\cup$, $\cap$, etc.).

4.1. Components

Every component specification is an *architectural type*. We distinguish architectural types from *basic types* (e.g., integers, strings, arrays, records, etc.). Unlike OOPLs, in which objects communicate by passing around other objects, in software architectures components are distinguished from the data they exchange during communication. In other words, a "component" in the sense in which we use it here is never passed from one component in an architecture to another.

A component has a name, a set of interface elements, an associated behavior, and (possibly) an implementation. Each interface element has a direction indicator (*prov*ided or *req*uired), a name, a set of parameters, and (possibly) a result. Each parameter, in turn, has a name and a type.

A component's behavior consists of an invariant and a set of operations. The invariant is used to specify any protocol constraints on the use of the component. Each operation has preconditions, postconditions, and (possibly) a result. Since operations are decoupled from interface elements, they also provide a set of variables used to express preconditions and postconditions. Like interface elements, operations can be *prov*ided or *req*uired. Only provided operations will have an implementation in a given component. The preconditions and postconditions of required operations express the *expected* semantics for those operations. Formal specification of an architectural type (component) is shown in Fig. 4.[2]

$$\begin{array}{|l}
\hline
\textit{Variable} \\
\quad name : STRING \\
\quad type : BASIC\_TYPE \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Int\_Element} \\
\quad dir : DIRECTION \\
\quad name : STRING \\
\quad params : \mathbb{P}\ Variable \\
\quad result : BASIC\_TYPE \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Operation} \\
\quad vars : \mathbb{P}\ Variable \\
\quad precond : Logic\_Pred \\
\quad postcond : Logic\_Pred \\
\quad result : BASIC\_TYPE \\
\quad dir : DIRECTION \\
\quad implementation : \mathrm{seq}\ STATEMENT \\
\hline
\quad dir = req \Rightarrow implementation = \varnothing \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Component} \\
\quad Basic\_Type\_Conformance \\
\quad name : STRING \\
\quad interface : \mathbb{P}\ Int\_Element \\
\quad invariant : Logic\_Pred \\
\quad operations : \mathbb{P}\ Operation \\
\quad int\_op\_map : Int\_Element \twoheadrightarrow Operation \\
\hline
\quad \mathrm{dom}\ int\_op\_map = interface \\
\quad \mathrm{ran}\ int\_op\_map = operations \\
\quad \forall\, ie : Int\_Element;\ o : Operation \mid \\
\quad\quad ie \in interface \land o \in operations \bullet \\
\quad\quad\quad (ie, o) \in int\_op\_map \\
\quad\quad\quad\quad \Leftrightarrow \\
\quad\quad\quad ie.dir = o.dir \land \\
\quad\quad\quad (ie.result, o.result) \in Basic\_Conf \land \\
\quad\quad\quad (\forall\, iv : Variable \mid iv \in ie.params \bullet \\
\quad\quad\quad\quad \exists\, ov : Variable \mid ov \in o.vars \bullet \\
\quad\quad\quad\quad\quad (ov.type, iv.type) \in Basic\_Conf) \\
\hline
\end{array}$$

Fig. 4. Z specification of architectural types (components). Relation *Basic_Conf* is defined in the schema *Basic_Type_Conformance* and relates two basic types, the first of which is a supertype of the second.

In the interest of space, Fig. 4 does not specify the relationship of component invariants to operation pre- and postconditions. This relationship can be summarized semi-formally as follows. Given a component C and operation O provided by C, for all valid input states of O that satisfy C's invariant and O's precondition, there exists a valid output state that satisfies both O's postcondition and C's invariant.

Since we separate the interface from the behavior, we define a function, *int_op_map*, which maps every interface element to an operation of the behavior. This function is a total surjection: each interface element is mapped to a single operation, while each operation implements at least one interface. An interface element can be mapped to an operation only if the types of its parameters are subtypes of the corresponding variable types in the operation, while the type of its result is a supertype of operation's result type. This property directly enables a single operation to export multiple interfaces.

An example of component specification, given in C2's ADL, is shown in Fig. 5. Only partial specifications of the *On-Line Database* (*OnLineDB*) and *NOSS Interaction Transaction Manager* (*NOSS_Mgr*) components from Fig. 1 are shown due to space constraints. For example, we show only the portion of *NOSS_Mgr* that is intended to interact with *OnLineDB*.

In a C2 architecture, a component has no dependencies on components below it (principle of *substrate independence*). Thus, *OnLineDB* has no dependencies on *NOSS_Mgr* and consequently has no required services. An example of mapping two interface elements to the same operation is given in the *NOSS_Mgr* component: both *ActivatePhoneLine* and *ActivateSpecialLine* (e.g., information or emergency) can be mapped to *op2* if *SPEC_NUM* is a subtype of *PHONE_NUM*.

4.2. Architectural Type Conformance

Informally, a subtyping relation, $\leq$, between two components, C_1 and C_2, is defined as the disjunction of the *nam*, *int*, *beh*, and *imp* relations shown in Fig. 2:

$$(\forall C_1, C_2 : \mathtt{Component})(C_2 \leq C_1 \;\Leftrightarrow\; C_2 \leq_{nam} C_1 \lor C_2 \leq_{int} C_1 \lor C_2 \leq_{beh} C_1 \lor C_2 \leq_{imp} C_1)$$

We consider these four relations in more detail below.

4.2.1. Name Conformance

Name conformance requires that a subtype share its supertype's interface element names and all interface parameter names. The subtype may introduce additional interface elements and additional parameters to existing interface elements. Two interface elements in a single component can have identical names, but then their sets of parameter names must differ. Name conformance rules are formally specified in Fig. 6.

Note that the possibility of introducing additional parameters to existing interface elements is different from method overloading and is typically not allowed in a PL. However, software architectures are at a level of abstraction

2. Capitalized identifiers are the basic (unelaborated) types in a Z specification. Trivial schemas and schemas whose meanings are obvious are omitted for brevity.

```
Component OnLineDB is
  State
    Customers  : set CUST;
    CustIds    : set CUST_ID;
    CustById   : CUST_ID  →  CUST;
  Interface
    prov ip1:  AddNewCust(new_cust:CUST);
    prov ip2:  RemoveCust(cust:CUST_ID);
    prov ip3:  ModifyCust(cust:CUST_ID; new_rec:CUST);
    prov ip4:  AccessCust(cust:CUST_ID) : CUST;
  Invariant
    #Customers  ≥  0;
  Operations
    prov op1:
      Let   c:CUST;
      Pre   c  ∉  Customers;
      Post  Customers’ = Customers  ∪  {c};
    prov op2:
      Let   c:CUST;
      Pre   c  ∈  Customers;
      Post  Customers’ = Customers - {c};
    prov op3:
      Let   id:CUST_ID;
            c:CUST;
      Pre   id  ∈  CustIds  ∧  c  ∉  Customers;
      Post  c  ∈  Customers  ∧  CustById(id) = c;
    prov op4:
      Let   id:CUST_ID;
      Pre   id  ∈  CustIds;
      Post  result = CustById(id);
  Map
    ip1 -> op1(new_cust -> c);
    ip2 -> op2(cust -> c);
    ip3 -> op3(cust -> id, new_rec -> c);
    ip4 -> op4(cust -> id);
end OnLineDB;
```

```
Component NOSS_Mgr is
  State
    Numbers : ADDR  →  PH_NUM;
  Interface
    prov ip1:  ActivatePhoneLine(a:ADDR; num:PH_NUM);
    prov ip2:  ActivateSpecialLine(a:ADDR; n:SPEC_NUM);
    prov ip3:  DeactivatePhoneLine(a:ADDR; num:PH_NUM);
    req ir1:   AddNewCust(new_cust:CUST);
    req ir2:   RemoveCust(cust:CUST_ID);
  Operations
    prov op1:
      Let   addr:ADDR;
            pn:PH_NUM;
      Pre   pn  ∉  Numbers(addr);
      Post  pn  ∈  Numbers’(addr);
    prov op2:
      Let   addr:ADDR;
            pn:PH_NUM;
      Pre   pn  ∈  Numbers(addr);
      Post  pn  ∉  Numbers’(addr);
    req or1:
      Let   c:CUST;
            cust_sv:STATE_VARIABLE;
      Pre   c  ∉  cust_sv;
      Post  c  ∈  cust_sv’;
    req or2:
      Let   c:CUST;
            cust_sv:STATE_VARIABLE;
      Pre   c  ∈  cust_sv;
      Post  c  ∉  cust_sv’;
  Map
    ip1 -> op1 (a -> addr, num -> pn);
    ip2 -> op1 (a -> addr, n -> pn);
    ip3 -> op2 (a -> addr, num -> pn);
    ir1 -> or1 (new_cust -> c);
    ir2 -> or2 (cust -> c);
end NOSS_Mgr;
```

Fig. 5. Partial specifications of the *OnLineDB* and *NOSS_Mgr* components from Fig. 1 in the C2 ADL. Basic type STATE_VARIABLE is discussed in Section 4.3.1. Labels for interface elements and operations (e.g., ip1, or2) are a notational convenience.

that is above source code and this feature may be supported by the architecture implementation infrastructure. For example, the implementation of the C2 class framework [14] allows the sender of the communication message to include parameters the receiver component does not expect; those parameters are simply ignored by the receiver. It is up to the architect to decide whether such a situation should be permitted in a given architecture.

$$\begin{array}{|l}
\textit{Param\_Name\_Conformance} \\
\hline
Prm\_Nam\_Conf : Int\_Element \leftrightarrow Int\_Element \\
\hline
\forall\, ie1, ie2 : Int\_Element \bullet \\
\quad (ie1, ie2) \in Prm\_Nam\_Conf \\
\qquad \Leftrightarrow \\
\quad (\forall\, p1 : Variable \mid p1 \in ie1.params \bullet \\
\quad\; \exists_1 p2 : Variable \mid p2 \in ie2.params \bullet \\
\qquad p1.name = p2.name \wedge \\
\qquad (\forall\, p3, p4 : Variable \mid \\
\qquad\; p3 \in ie1.params \wedge p4 \in ie2.params \bullet \\
\qquad\quad p3.name = p4.name \\
\qquad\qquad \Leftrightarrow \\
\qquad\quad (p1 = p3 \wedge p2 = p4) \vee \\
\qquad\quad (p1 \neq p3 \wedge p2 \neq p4)))
\end{array}$$

$$\begin{array}{|l}
\textit{Name\_Conformance} \\
\hline
Param\_Name\_Conformance \\
Nam\_Conf : Component \leftrightarrow Component \\
\hline
\forall\, c1, c2 : Component \bullet \\
\quad (c1, c2) \in Nam\_Conf \\
\qquad \Leftrightarrow \\
\quad (\forall\, ie1 : Int\_Element \mid ie1 \in c1.interface \bullet \\
\quad\; \exists\, ie2 : Int\_Element \mid ie2 \in c2.interface \bullet \\
\qquad ie1.name = ie2.name \wedge \\
\qquad (ie1, ie2) \in Prm\_Nam\_Conf)
\end{array}$$

Fig. 6. Name conformance.

4.2.2. Interface Conformance

Name conformance is a rather weak conformance requirement and we have encountered it in practice only as part of the stronger *interface* conformance relationship. Component C_2 is an interface subtype of C_1 if and only if it provides at least (but not necessarily only) the interface elements provided by C_1 with identical direction indicators, and *matching* parameters and results for each interface element. Two parameters belonging to the two components’ interface elements match if and only if they have identical names (*Param_Name_Conformance* schema in Fig. 6) and each

$$\begin{array}{|l}
\textit{Param\_Conformance} \\
\hline
Basic\_Type\_Conformance \\
Param\_Name\_Conformance \\
Prm\_Conf : Int\_Element \leftrightarrow Int\_Element \\
\hline
\forall\, ie1, ie2 : Int\_Element \bullet \\
\quad (ie1, ie2) \in Prm\_Conf \\
\qquad \Leftrightarrow \\
\quad (ie1, ie2) \in Prm\_Nam\_Conf \wedge \\
\quad (\forall\, p1, p2 : Variable \mid \\
\quad\; p1 \in ie1.params \wedge p2 \in ie2.params \wedge p1.name = p2.name \bullet \\
\qquad (p2.type, p1.type) \in Basic\_Conf)
\end{array}$$

$$\begin{array}{|l}
\textit{Interface\_Conformance} \\
\hline
Name\_Conformance \\
Param\_Conformance \\
Int\_Conf : Component \leftrightarrow Component \\
\hline
\forall\, c1, c2 : Component \bullet \\
\quad (c1, c2) \in Int\_Conf \\
\qquad \Leftrightarrow \\
\quad (c1, c2) \in Nam\_Conf \wedge \\
\quad (\forall\, ie1, ie2 : Int\_Element \mid \\
\quad\; ie1 \in c1.interface \wedge ie2 \in c2.interface \bullet \\
\qquad ie1.dir = ie2.dir \wedge \\
\qquad (ie1, ie2) \in Prm\_Nam\_Conf \Rightarrow (ie1, ie2) \in Prm\_Conf \wedge \\
\qquad (ie1.result, ie2.result) \in Basic\_Conf)
\end{array}$$

Fig. 7. Interface conformance.

```
Component CallPlanDB is subtype OnLineDB(int)
  Interface
    prov ip1:  AddNewCust(new_cust:CUST);
    prov ip2:  RemoveCust(cust:CUST_ID);
    prov ip3:  ModifyCust(cust:CUST_ID; new_rec:CUST);
    prov ip4:  AccessCust(cust:CUST_ID) : CUST;
    prov ip5:  EnterCallPlan(c:CUST_ID; p:CALL_PLAN);
  Invariant
    . . .
  Operations
    prov op10:
      Let    id:CUST_ID;
             plan:CALL_PLAN;
      Pre    id  ∈  CustIds  ∧  plan  ∈  CallPlans;
      Post   plan  ∈  CustPlans(CustById(id));
    . . .
  Map
    ip5 -> op10(c -> id, p -> plan);
    . . .
end CallPlanDB;
```

Fig. 8. *CallPlanDB* is an interface subtype of *OnLineDB*.

parameter type of C_1 is a subtype of the corresponding parameter type of C_2 (*contravariance of parameters*, defined in the *Param_Conformance* schema in Fig. 7). The results of two corresponding interface elements match if the result type in C_1 is a supertype of the result type in C_2 (*covariance of result*). For each interface element, the subtype must provide at least (but not necessarily only) the parameters that match the supertype's parameters. Interface conformance rules are formally specified in Fig. 7.

For example, component *CallPlanDB*, shown in Fig. 8, is an interface subtype of *OnLineDB* from Fig. 5. It does not matter what the invariant, operations, and *int_op_map* of *CallPlanDB* are for this relationship to hold. These details have thus been omitted.

4.2.3. Behavior Conformance

Behavior conformance requires that the invariant of the supertype be ensured by that of the subtype. Furthermore, each operation of the supertype must have a corresponding operation in the subtype (the subtype can also introduce additional operations), where the subtype's operation has the same direction indicator as the supertype's, the same or weaker preconditions, same or stronger postconditions, and preserves result covariance.

No constraints are placed on the relationship between the types of the supertype's and subtype's corresponding operation variables. This relationship can vary, but is always an instance of one of the two cases depicted in Fig. 9. Thus, any

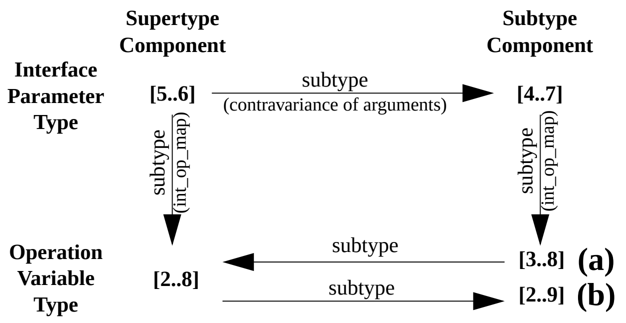

Fig. 9. Contravariance of arguments and the *int_op_map* function do not guarantee a particular relationship between supertype's and subtype's operation variable types (illustrated using integer subranges): (a) supertype component's variable type is a supertype of subtype component's; (b) supertype component's variable type is a subtype of subtype component's.

$$
\begin{array}{l}
\textit{Oper\_Conformance} \\
\hline
\textit{Basic\_Type\_Conformance} \\
\textit{Logical\_Implication} \\
\textit{Oper\_Conf} : \textit{Operation} \leftrightarrow \textit{Operation} \\
\hline
\forall\, o1, o2 : \textit{Operation} \bullet \\
\quad (o1, o2) \in \textit{Oper\_Conf} \\
\qquad \Leftrightarrow \\
\quad (\forall\, v1 : \textit{Variable} \mid v1 \in o1.\textit{vars} \bullet \\
\quad\ \exists\, v2 : \textit{Variable} \mid v2 \in o2.\textit{vars} \bullet \\
\qquad (v1.\textit{type}, v2.\textit{type}) \in \textit{Basic\_Conf} \vee \\
\qquad (v2.\textit{type}, v1.\textit{type}) \in \textit{Basic\_Conf}) \wedge \\
\quad (o1.\textit{precond}, o2.\textit{precond}) \in \textit{Logic\_Impl} \wedge \\
\quad (o2.\textit{postcond}, o1.\textit{postcond}) \in \textit{Logic\_Impl} \wedge \\
\quad (o1.\textit{result}, o2.\textit{result}) \in \textit{Basic\_Conf}
\end{array}
$$

$$
\begin{array}{l}
\textit{Behavior\_Conformance} \\
\hline
\textit{Oper\_Conformance} \\
\textit{Logical\_Implication} \\
\textit{Beh\_Conf} : \textit{Component} \leftrightarrow \textit{Component} \\
\hline
\forall\, c1, c2 : \textit{Component} \bullet \\
\quad (c1, c2) \in \textit{Beh\_Conf} \\
\qquad \Leftrightarrow \\
\quad (c2.\textit{invariant}, c1.\textit{invariant}) \in \textit{Logic\_Impl} \wedge \\
\quad (\forall\, o1 : \textit{Operation} \mid o1 \in c1.\textit{operations} \bullet \\
\quad\ \exists\, o2 : \textit{Operation} \mid o2 \in c2.\textit{operations} \bullet \\
\qquad o1.\textit{dir} = o2.\textit{dir} \wedge \\
\qquad (o1, o2) \in \textit{Oper\_Conf})
\end{array}
$$

Fig. 10. Behavior conformance. *Logic_Impl* is a relation that denotes that the first element in the relation implies the second.

relationship between the variable types is allowed so long as the proper relationships between operation pre- and postconditions are maintained. The rules for behavior conformance are specified in Fig. 10.

Fig. 11 shows an example of behavior conformance. Component *BoundedDB* is a behavior subtype of *OnLineDB* from Fig. 5. The interface and *int_op_map* of *BoundedDB* are unimportant for this relationship to hold. These details have thus been omitted. *BoundedDB* is a behavior subtype of *OnLineDB* because it provides (at least) the same operations as *OnLineDB* and its invariant (number of customers, representing the size of the database), is between zero and one million, inclusive, which implies *OnLineDB*'s invariant (number of customers is zero or greater).

The subtyping relationship that results from the combination of the *Behavior_Conformance* and *Interface_Conformance* schemas (in particular, the *Beh_Conf* and *Int_Conf* relations they define), and the mapping function, *int_op_map*, represents a point in the region depicted in Fig. 3b. This relationship is similar to other notions of behavioral subtyping [2, 3, 11] in that it guarantees substitutability between a supertype and a subtype in an architecture.

```
Component BoundedDB is subtype OnLineDB(beh)
  Interface
    . . .
  Invariant
    0  ≤  #Customers  ≤  1000000;
  Operations
    [OnLineDB operations]
    . . .
  Map
    . . .
end BoundedDB;
```

Fig. 11. *BoundedDB* is a behavior subtype of *OnLineDB*.

```
Component OnLineDB-2 is subtype OnLineDB(int and beh)
  State
    Customers  : set CUST;
    CustIds    : set CUST_ID;
    CustById   : CUST_ID  →  CUST;
    CallPlans  : set CALL_PLAN;
    CustPlans  : CUST  →  set CALL_PLAN;
  Interface
    prov ip1:  AddNewCust(new_cust:CUST);
    prov ip2:  RemoveCust(cust:CUST_ID);
    prov ip3:  ModifyCust(cust:CUST_ID; new_rec:CUST);
    prov ip4:  AccessCust(cust:CUST_ID) : CUST;
    prov ip5:  EnterCallPlan(c:CUST_ID; p:CALL_PLAN);
  Invariant
    0  ≤  #Customers  ≤  1000000;
  Operations
    prov op1:
      Let    c:CUST;
      Pre    #Customers < 1000000  ∧  c  ∉  Customers;
      Post   Customers′ = Customers  ∪  {c};
    prov op2:
      Let    c:CUST;
      Post   c  ∈  Customers  ⇒
             Customers′ = Customers - {c};
    prov op3:
      Let    id:CUST_ID;
             c:CUST;
      Pre    id  ∈  CustIds ∧  c  ∉  Customers;
      Post   c  ∈  Customers  ∧  CustById(id) = c;
    prov op4:
      Let    id:CUST_ID;
      Pre    id  ∈  CustIds;
      Post   result = CustById(id);
    prov op5:
      Let    id:CUST_ID;
             plan:CALL_PLAN;
      Pre    id  ∈  CustIds  ∧  plan  ∈  CallPlans;
      Post   plan  ∈  CustPlans(CustById(id));
  Map
    ip1 -> op1(new_cust -> c);
    ip2 -> op2(cust -> c);
    ip3 -> op3(cust -> id, new_rec -> c);
    ip4 -> op4(cust -> id);
    ip5 -> op5(c -> id, p -> plan);
end OnLineDB-2;
```

Fig. 12. *OnLineDB-2* is a candidate interface and behavior subtype of *OnLineDB*.

The *OnLineDB-2* component, shown in Fig. 12 is a candidate interface and behavior subtype of *OnLineDB* from Fig. 5. *OnLineDB-2* includes the already discussed features from *CallPlanDB* and *BoundedDB*. Additionally, it slightly changed operation specifications for *op1* and *op2* (corresponding, in this case, to interface elements *AddNewCust* and *RemoveCust*). *Op1* will not add a customer to the database if the database is already full. *Op2* does not require that the customer already be in the database; instead, it will check for the customer record and, if found, remove it. For *OnLineDB-2* to be an *int* **and** *beh* subtype of *OnLineDB*, the following must be true (from the schema *Oper_Conformance* in Fig. 10):

- *OnLineDB op1 pre* ⇒ *OnLineDB-2 op1 pre*
  `(c ∉ Customers) ⇒`
  `(#Customers < 1000000 ∧ c ∉ Customers)`
- *OnLineDB-2 op1 post* ⇒ *OnLineDB op1 post*
  `(Customers′ = Customers ∪ {c}) ⇒`
  `(Customers′ = Customers ∪ {c})`
- *OnLineDB op2 pre* ⇒ *OnLineDB-2 op2 pre*
  `(c ∈ Customers) ⇒ true`
- *OnLineDB-2 op2 post* ⇒ *OnLineDB op2 post*
  `(c ∈ Customers ⇒`
  ` Customers′ = Customers - {c}) ⇒`
  `(Customers′ = Customers - {c})`

The first implication is not true. The left hand side (LHS) may be true even if the database is full, in which case the right hand side (RHS) is false. The second implication is true since LHS and RHS are the same. The third implication is true, since *OnLineDB-2*'s *op2* does not have any preconditions. Finally, the fourth implication is true. It has the form (A ⇒B) ⇒B,

$$
\begin{array}{l}
\hline
\mathit{Implementation\_Conformance} \\
\mathit{Behavior\_Conformance} \\
\mathit{Imp\_Conf} : \mathit{Component} \leftrightarrow \mathit{Component} \\
\hline
\forall\, c1, c2 : \mathit{Component} \bullet \\
\qquad (c1, c2) \in \mathit{Imp\_Conf} \\
\qquad\quad \Leftrightarrow \\
\qquad (c1, c2) \in \mathit{Beh\_Conf} \wedge \\
\qquad (c1.\mathit{invariant}, c2.\mathit{invariant}) \in \mathit{Logic\_Impl} \wedge \\
\qquad (\forall\, o1 : \mathit{Operation} \mid o1 \in c1.\mathit{operations} \bullet \\
\qquad\ \exists_1 o2 : \mathit{Operation} \mid o2 \in c2.\mathit{operations} \bullet \\
\qquad\qquad (o2, o1) \in \mathit{Oper\_Conf} \wedge \\
\qquad\qquad o1.\mathit{implementation} = o2.\mathit{implementation}) \\
\hline
\end{array}
$$

Fig. 13. Implementation conformance.

which is false if both A and B are false. However, B in our case will always be true because of the definition of set subtraction: if `c ∈ Customers`, `c` will be removed from the new value of the `Customers` set (`Customers′`); if this is not the case, `Customers` will simply remain the same.

The first implication above is therefore the only one that violates the required relationship. It is because of this that *OnLineDB-2* is not an *int* **and** *beh* subtype of *OnLineDB*. Note that the architect may still decide to use *OnLineDB-2*, particularly since it is so closely related to *OnLineDB*, but must understand that *OnLineDB-2* cannot be substituted for *OnLineDB* in a correctness-preserving manner.

4.2.4. Implementation Conformance

Although useful in practice for evolving components, *implementation* conformance is not a particularly interesting relationship from a type-theoretic point of view. Implementation conformance may be established with a simple syntactic check if the operations of the subtype have identical implementations (both syntactically and semantically) as the corresponding operations of the supertype. Implementation conformance between two types thus also requires a behavioral equivalence between their shared operations, as shown in Fig. 13.

4.3. Type Checking a Software Architecture

In order to discuss type conformance of interoperating components, we must define an architecture that includes those components. There is no single, universally accepted set of guidelines for composing architectural elements. Instead, architectural topology depends on the ADL in which the architecture is modeled, characteristics of the application domain, and/or the rules of the chosen architectural style. We therefore had to make certain choices in specifying properties of an architecture:

- we model connectors explicitly, unlike, e.g., Darwin [12] and Rapide [10];
- we allow direct connector-to-connector links, unlike, e.g., Wright [1];
- finally, we assume certain topological constraints that are derived from the rules of the C2 style [25]: a component is attached to single connectors on its top and bottom sides, while a connector can be attached to multiple components and connectors on its top and bottom.

None of the above choices is required by our type theory. It is indeed possible to provide a definition of architecture that reflects any other compositional guidelines. However, these decisions were necessary in order to formally specify and check type conformance criteria.

$$
\begin{array}{l}
\underline{\quad Architecture \quad}\\
components : \mathbb{P}\ Component\\
connectors : \mathbb{P}\ Connector\\
comp\_conn : Component \leftrightarrow Connector\\
conn\_comp : Connector \leftrightarrow Component\\
conn\_conn : Connector \leftrightarrow Connector\\
Comm\_Link : Component \leftrightarrow Component\\
\hline
\mathrm{dom}\ comp\_conn = components\\
\mathrm{ran}\ comp\_conn = connectors\\
\mathrm{dom}\ conn\_comp = connectors\\
\mathrm{ran}\ conn\_comp = components\\
\mathrm{dom}\ conn\_conn = connectors\\
\mathrm{ran}\ conn\_conn = connectors\\
\mathrm{dom}\ Comm\_Link = components\\
\mathrm{ran}\ Comm\_Link = components\\
\\
\forall\, c : Component;\ b : Connector \mid\\
\quad c \in components \land b \in connectors \bullet\\
\qquad (c, b) \in comp\_conn \Rightarrow (b, c) \notin conn\_comp \land\\
\qquad (b, c) \in conn\_comp \Rightarrow (c, b) \notin comp\_conn\\
\\
\forall\, b1, b2 : Connector \mid b1 \in connectors \land b2 \in connectors \bullet\\
\qquad (b1, b2) \in conn\_conn \Rightarrow (b1 \neq b2 \land (b2, b1) \notin conn\_conn)\\
\\
\forall\, c1, c2 : Component \mid c1 \in components \land c2 \in components \bullet\\
\qquad (c1, c2) \in Comm\_Link\\
\qquad\quad \Leftrightarrow\\
\qquad c1 \neq c2 \land\\
\qquad (\exists\, b1, b2 : Connector \mid b1 \in connectors \land b2 \in connectors \bullet\\
\qquad\quad ((c1, b1) \in comp\_conn \land\\
\qquad\quad (b2, c2) \in conn\_comp \land\\
\qquad\quad (b1, b2) \in conn\_conn^{*})\\
\qquad\qquad \lor\\
\qquad\quad ((c2, b1) \in comp\_conn \land\\
\qquad\quad (b2, c1) \in conn\_comp \land\\
\qquad\quad (b1, b2) \in conn\_conn^{*}))
\end{array}
$$

Fig. 14. Formal definition of architecture.

The formal definition of architecture is given in Fig. 14. Connectors are treated simply as communication routing devices; therefore their definitions are omitted. Two components can interoperate if there is a communication link between them. This means that they are either on the opposite sides of the same connector or one can be reached from the other by following one or more connector-to-connector links (defined by the *Comm_Link* relation).

For example, in the architecture from Fig. 1, there is a communication link between *OnLineDB* and *NOSS_Mgr* components (via a single connector-to-connector link). There is also a link between *Transaction Consistency Checker* and *Customer Support Local Transaction State* components (different sides of the same connector). On the other hand, there is no communication link between *On-Line* and *NOSS* databases: they are attached below the same (top-most) connector; however, the *Comm_Link* relation mandates that they be on different sides of a connector, which reflects C2's communication rules.

Given this definition of architecture, it is possible to specify type checking predicates. As already discussed, components need not be able to fully interoperate in an architecture. The two extreme points on the spectrum of type conformance are:

- *minimal type conformance*, where at least one service (interface and corresponding operation) required by each component is provided by some other component along its communication links; and
- *full type conformance*, where every service required by every component is provided by some component along its communication links.

$$
\begin{array}{l}
\underline{\quad Minimal\_Type\_Conformance \quad}\\
Interface\_Conformance\\
Behavior\_Conformance\\
Architecture\\
\hline
\forall\, c1 : Component \mid c1 \in components \bullet\\
\exists\, c2 : Component \mid c2 \in components \land (c1, c2) \in Comm\_Link \bullet\\
\qquad (\exists\, ie1, ie2 : Int\_Element \mid\\
\qquad\ ie1 \in c1.interface \land ie2 \in c2.interface \bullet\\
\qquad\qquad ie1.name = ie2.name \land\\
\qquad\qquad ie1.dir = req \land ie2.dir = prov \land\\
\qquad\qquad (ie1, ie2) \in Prm\_Conf \land\\
\qquad\qquad (c1.int\_op\_map(ie1),\\
\qquad\qquad\ c2.int\_op\_map(ie2)) \in Oper\_Conf)
\end{array}
$$

$$
\begin{array}{l}
\underline{\quad Full\_Type\_Conformance \quad}\\
Interface\_Conformance\\
Behavior\_Conformance\\
Architecture\\
\hline
\forall\, c1 : Component;\ ie1 : Int\_Element \mid\\
\quad c1 \in components \land ie1 \in c1.interface \land ie1.dir = req \bullet\\
\exists\, c2 : Component;\ ie2 : Int\_Element \mid\\
\quad c2 \in components \land (c1, c2) \in Comm\_Link \land\\
\quad ie2 \in c2.interface \land ie2.dir = prov \bullet\\
\qquad ie1.name = ie2.name \land\\
\qquad (ie1, ie2) \in Prm\_Conf \land\\
\qquad (c1.int\_op\_map(ie1),\ c2.int\_op\_map(ie2)) \in Oper\_Conf
\end{array}
$$

Fig. 15. Type conformance predicates.

They are defined in Fig. 15. The predicates expressing the degree of utilization of a component's provided services in an architecture can be specified in a similar manner [17].

Depending on the requirements of a given project (reliability, safety, budget, deadlines, etc.), type conformance corresponding to different points along the spectrum may be adequate. What would be classified as a "type error" in one architecture may be acceptable in another. Therefore, architectural type correctness is expressible in terms of a percentage corresponding to the degree of conformance (per component or for the architecture as a whole).

4.3.1. Type Conformance and Off-the-Shelf Reuse

Before we can illustrate architectural type conformance with an example, we need to address another issue. Establishing type conformance brings up the question of how much a component may know about other components with which it will interoperate. Although magnified by our separation of provided from required component services, this issue is not unique to our type theory. Rather, it is pertinent to all approaches that model behavior of a type and enforce behavioral conformance.

To demonstrate behavioral conformance between two interoperating components, by definition one must show that a specific relationship holds between their respective behaviors. This relationship is one of several flavors of equivalence or implication, summarized in [29].

Establishing whether two components can interoperate includes matching the specification of what is expected by a required operation of one component against what another component's provided operation supplies. Behavior of an operation is modeled in terms of its interface parameters (in our approach, operation variables) and component state variables. A component may thus need to refer to state variables that belong to another component in order to specify a *required* operation's expected behavior. However, doing so would be a violation of the "provider" component's abstraction. It would also violate some basic principles of component-based development:

- the designer may not know in advance which, if any, components will contain a matching specification for the required operation and, thus, what the appropriate (types of) state variables are. This is particularly the case when using behavior matching to aid component discovery and retrieval. For example, it is not reasonable to expect that a user of the *op1* operation in the *NOSS_Mgr* component from Fig. 5 would know that its behavior is expressed in terms of a function (*Numbers*) that, given an address, returns a set of phone numbers;
- large-scale, component-based development treats an off-the-shelf component as a black box, thereby intentionally hiding the details of its internal state. Having to explicitly refer to those details would require them to be exposed.

Existing approaches to behavior modeling and conformance checking have not addressed this problem. The problem does not apply to component subtyping: the designer must know all of existing component's details in order to effectively evolve it. Thus, those approaches that focus on behavioral subtyping (e.g., America [2], Liskov and Wing [11], and Leavens et al. [3]) do not encounter this problem. Zaremski and Wing [29] do address component retrieval and interoperability. However, their approach makes the very assumption that the designer will have access to a "provider" component's state (via a shared Larch trait [8]). Fischer and colleagues [4, 22] model components at the level of a single procedure. In order to be able to properly specify pre- and postconditions, they include all the necessary variables as procedure parameters. Thus, for example, the stack itself is passed as a parameter to the *push* procedure.

The solution to this problem we propose is based on two requirements arising from a more realistic assessment of component-based development:

- we do not have access to a "provider" component's internal state (unlike Zaremski and Wing's approach), and
- we cannot change the way many software components, especially in the OO world, are modeled (unlike Fischer et al.).

These two requirements result in an obvious third requirement:

- we must somehow refer to a "provider" component's state when modeling operations, even though we do not know what that state is.

This seeming paradox actually suggests our approach. The initial results of this approach are promising and we intend to further investigate its practicality.

We model a required operation as if we have access to a "provider" component's state. However, since we do not know the actual "provider" state variables or their types, we introduce a generic type, *STATE_VARIABLE*, which is a supertype of all basic types. Thus, variables of this type are essentially placeholders in logical predicates. When matching, e.g., a required and provided precondition, we attempt to unify (instantiate) each variable of the *STATE_VARIABLE* type in the required precondition with a corresponding state variable in the provided precondition. If the unification is possible and the implication (with all instances of *STATE_VARIABLE* placeholders replaced with actual variables) holds, then the two preconditions conform.

In the example from Fig. 5, *NOSS_Mgr* requires two services: *AddNewCust*, which is mapped to its operation *or1*, and *RemoveCust*, mapped to *or2*. OnLineDB provides operations with matching interfaces (as required by the type conformance predicates). Thus, to establish type conformance, we must now make sure that the operation pre- and postconditions are properly related. In the interest of space, we do so only for *or1*:

- *NOSS_Mgr or1 pre* $\Rightarrow$ *OnLineDB op1 pre*
  `(c ∉ cust_sv) ⇒ (c ∉ Customers)`
  In this case, `cust_sv` is instantiated with `Customers` and we have an implication of the form A $\Rightarrow$ A, which is obviously true.
- *OnLineDB op1 post* $\Rightarrow$ *NOSS_Mgr or1 post*
  `(Customers' = Customers ∪ {c}) ⇒`
  `(c ∈ cust_sv')`
  In this case, since `Customers` is the only state variable in the provided operation (*op1*), `cust_sv` is again instantiated with `Customers`, and the implication becomes
  `(Customers' = Customers ∪ {c}) ⇒`
  `(c ∈ Customers')`
  This implication is also true (if an item is added to a set, that item is an element of the set).

We have thus established that, at the least, minimal type conformance holds in the architectural interaction between *OnLineDB* and *NOSS_Mgr.*

4.4. Summary

This section defined and demonstrated with examples the major elements of our type theory: multiple subtyping relationships (Section 4.2) and type conformance (Section 4.3). Certain characteristics of our type theory are unique (e.g., separation of interface from behavior) and give rise to seemingly anomalous relationships when considered in isolation (e.g., supertype and subtype operation variable types depicted in Fig. 9). However, the type theory as a whole supplies mechanisms that prevent any such anomalies. For example, the *int_op_map* function constrains the actual use of operation variables with the types of interface parameters through which the variables are accessed. The desired relationship between a supertype's and subtype's operation variables is thus ensured.

5. CONCLUSIONS AND FUTURE WORK

Software architectures show great potential for reducing development costs while improving the quality of the resulting software. Architectures also provide a promising basis for supporting software evolution. However, improved evolvability cannot be achieved simply by explicitly focusing on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures, and ADLs in particular, as tools which also must be supported with specific techniques to achieve desired properties. This paper has outlined such a technique for supporting evolution of software components in a manner that preserves the desired architectural relationships and properties.

This technique is based on the recognition that, unlike PLs, software architectures need not always be rigid in establishing properties such as consistency and completeness. For example, it is not always the case that two components that share a communication link can actually communicate (e.g., due to mismatched interfaces). At the architectural level, this can be detected via type checking and prevented. However, even if such a configuration is allowed to propagate into the implemented system, implementation-time decisions (e.g., communication via implicit invocation) may result in the loss of communication messages [15, 25], but still allow the rest of the system's architecture to perform at least in a degraded mode. Thus, informing the architect of the potential problem and leaving the decision up to the architect is often preferable to automatically rejecting the option.

We have already put many of these ideas into practice in the context of the C2 style and its accompanying ADL. We are currently developing a set of tools to support architectural subtyping, type checking, and mapping of architectural descriptions to the C2 implementation infrastructure [14]. We are also considering several existing theorem provers and model checkers to aid us specifically in establishing component invariant and operation pre- and postcondition conformance: NORA/HAMMR [22], Larch proof assistant (LP) [8], VCR [4], and PVS [20].

A number of issues remain items of future work. These include investigation of the applicability of our type theory for evolving connectors, application of the type theory to other ADLs and across multiple levels of architectural refinement, further research of issues in adapting and adopting legacy components into architectures using the subtyping approach, and automating the evolution of existing components to populate partial architectures.

6. REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Lecture Notes in Computer Science*, volume 489, Springer-Verlag, 1991.

[3] K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. Technical Report, TR# 95-20c, Department of Computer Science, Iowa State University, August 1995, revised March 1997.

[4] B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-Based Software Component Retrieval Tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, New Orleans, LA, USA, December 1994.

[6] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.

[7] C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[8] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[9] D. Lea and J. Marlowe. Interface-Based Protocol Specifications of Open Systems Using PSL. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, Aarhus, Denmark, August 1995.

[10] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.

[11] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[12] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.

[13] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.

[14] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)* and *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.

[15] N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, October-December 1997.

[16] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.

[17] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium*, Los Angeles, CA, April 1996.

[18] O. Nierstrasz. Regular Types for Active Objects. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93)*, Washington, D.C., USA, October 1993.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. To appear in *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, April 1998, Kyoto, Japan.

[20] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. *PVS:* Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, eds., *Computer-Aided Verification (CAV '96)*, volume 1102 of Lecture Notes in Computer Science, July/August 1996, Springer-Verlag.

[21] J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, vol. 3, num. 2, 1992.

[22] J. Schumann and B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of Automated Software Engineering (ASE-97)*, Lake Tahoe, November 1997.

[23] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.

[24] J. M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science, Prentice Hall International, 1989.

[25] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.

[26] A. L. Wolf, editor. *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, October 1996.

[27] A. L. Wolf. Succeedings of the Second International Software Architecture Workshop (ISAW-2). *ACM SIGSOFT Software Engineering Notes*, January 1997.

[28] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, Portland, OR, USA, October 1994.

[29] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, DC, October 1995.