Comprehensive Specification of Distributed Systems Using $\mathcal{I}^5$ and IOA

M. Cecilia Bastarrica
DCC, Universidad de Chile
Av. Blanco Encalada 2120
Santiago, Chile
cecilia@dcc.uchile.cl
phone:(562)678-4362

Steven A. Demurjian
CS&E, Univ. of Connecticut
191 Auditorium Rd., U-155
Storrs, CT 06269, USA
steve@cse.uconn.edu
phone:(860)486-4818

Alex A. Shvartsman
CS&E, Univ. of Connecticut
191 Auditorium Rd., U-155
Storrs, CT 06269, USA
aas@cse.uconn.edu
phone:(860)486-2672
&
LCS, MIT
545 Tech. Sq., Bldg. NE43
Cambridge, MA 02139, USA

Abstract

Low level difficulties in the development of distributed systems that are due to non-standard communication protocols and incompatible components or platforms have largely been solved through standardization and commoditization of protocols and platforms. Distributed systems are being designed at higher levels of sophistication these days, and having an expressive yet usable specification language is a valuable tool.

IOA is a formal language for specifying the semantics of distributed systems. $\mathcal{I}^5$ is a specification framework for architectural definition of distributed systems, also intended as a basis for configuration management. $\mathcal{I}^5$ has five levels that specify mainly the structural characteristics at different levels of abstraction, but $\mathcal{I}^5$ does not address the semantics or dynamics of distributed systems interactions. We explore the integration of IOA and $\mathcal{I}^5$ to create combined specifications that enjoy the benefits of both specification languages: the five different levels of abstraction of $\mathcal{I}^5$ with their structural specification capabilities are enhanced by a semantic specification written in IOA. We show an example of a specification developed using IOA and $\mathcal{I}^5$ in an integrated way. We consider general approaches to such integrated specifications and discuss the possibilities and limitations of integrating IOA and $\mathcal{I}^5$, as well as our future work towards the complete integration.

Keywords: software engineering, software architecture, formal specifications, distributed systems, distributed software engineering.

1. Introduction

Distributed systems are ubiquitous. Computers are connected, and software components running on different computers work together interacting to achieve common goals. Low level technical problems like communication protocols have long been solved. New sophisticated and powerful distributed systems are being developed, and new challenges include the need to design systems at a higher level of abstraction.

Software architecture has become important within the software engineering community. There has been a big effort in developing architectural patterns [4] and several architectural definition languages (ADLs). Examples of these ADLs are Rapide [10], Aesop [6], MetaH [17], UniCon [15], Wright [1], C2 [13], SADL [14], and ACME [7]. All these languages provide formality to architectural specifications, something lacking from box and line diagrams that are still common in professional practice. Distributed systems are the most typical application for ADLs since they are naturally thought of as a set of interacting components. ADLs are precise for specifying distributed systems, but even the simplest systems require a very verbose and detailed specification.

$\mathcal{I}^5$ is a framework for specifying the architecture of distributed systems. $\mathcal{I}^5$ can be considered an ADL. It is fully formalized [3], and it also has many interesting features: it has five integrated levels of abstraction, it includes software and hardware features, and it has a graphical and textual notation. All these features make $\mathcal{I}^5$ a powerful specification framework, providing the designer with a graphical high level language and a textual detailed specification language. $\mathcal{I}^5$ provides a high abstraction level specification framework, but it does not have any means for specifying semantics or dynamics of the communication of distributed systems.

Input/output automata is a formal language designed by Lynch and Tuttle [12] at MIT for specifying the semantics of distributed systems. IOA is a precise language for describing Input/Output Automata and for stating their properties [8]. IOA models distributed systems as a set of automata that have an internal state and may execute input, output and internal transitions. IOA has been successfully used for specifying distributed algorithms [11].

IOA is used to specify semantics of distributed systems at a level of abstraction that does not include the ability to specify system deployment. $\mathcal{I}^5$ is a framework for specifying the architecture of distributed systems but includes no semantics. Our objective is to investigate the integration of the two languages and produce a powerful specification framework taking advantage of the abstraction levels and structure of $\mathcal{I}^5$ and the expressiveness for specifying distributed systems' semantics and dynamics of IOA.

In Section 2 we present an overview of $\mathcal{I}^5$ and IOA by developing a common example in both specification languages. In Section 3 we show how the two specifications can be integrated into a unique more powerful specification of the same example. We analyze the benefits of the integrated specification. We draw some conclusions in Section 4 and discuss the possibility of a complete integration.

2. $\mathcal{I}^5$ and IOA

We introduce input/output automata and $\mathcal{I}^5$ by developing an example. The **GUESS-GENERATOR** example has three types of interacting components, one that chooses a number, another one that generates numbers and tries to guess the chosen number, and a third one that counts the number of times the number generator guesses the number. This simple example highlights some structural and dynamics features that become apparent when specified either with IOA or $\mathcal{I}^5$.

2.1. The $\mathcal{I}^5$ Framework

$\mathcal{I}^5$ is a five level specification framework specially intended for architectural specification of distributed systems [3]. Each level addresses different aspects of the specification in a decreasing level of abstraction including both software and hardware features, and using either graphical or textual notation. The graphical notation is based on customized UML implementation diagrams [5], and the textual notation is based on the Z specification language [16].

The levels of $\mathcal{I}^5$ are *Interface*, *Implementation*, *Integration*, *Instantiation*, and *Installation*, the five *I*s. In any level, the designer can switch between graphical and textual notation, according to his preferences. However, we have found that graphical notation is better for the first three levels and textual notation is more practical for the last two [2] because graphics represent better high level abstractions, but text is more scaleable when more and more detail is added.

Each level of specification in $\mathcal{I}^5$ deals with different concepts and uses a different notation, but they are all related. Figure 1 shows the five specification levels included in a $\mathcal{I}^5$ with their software and hardware parts, and the dependencies among them. We describe each level and show how the **GUESS-GENERATOR** example is represented.

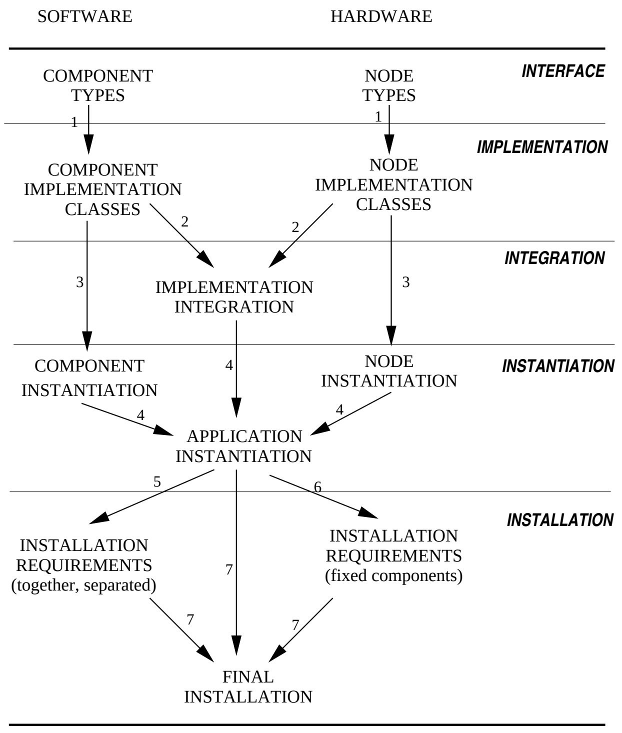
SOFTWARE
HARDWARE
INTERFACE
COMPONENT TYPES
NODE TYPES
IMPLEMENTATION
COMPONENT IMPLEMENTATION CLASSES
NODE IMPLEMENTATION CLASSES
INTEGRATION
IMPLEMENTATION INTEGRATION
INSTANTIATION
COMPONENT INSTANTIATION
NODE INSTANTIATION
APPLICATION INSTANTIATION
INSTALLATION
INSTALLATION REQUIREMENTS (together, separated)
INSTALLATION REQUIREMENTS (fixed components)
FINAL INSTALLATION
1
2
3
4
5
6
7

Figure 1. The hardware and software levels of $\mathcal{I}^5$ and their dependencies.

Interface. This first level defines the component types of the application, and the node and connector types of the target network. For every component type, the *Interface* specification provides a name, a set of interfaces, and a set of calls to interfaces in other component types. Interface inheritance is also specified at this level: a subtype has the same interfaces and calls as its supertypes and it extends them.

In Figure 2, we show the graphical representation of the software *Interface* level of $\mathcal{I}^5$ for the example. There are three component types: **GENERATOR**, **GUESS** and **WINNER**. The **GENERATOR** sends a number to **GUESS**, this one checks if it matches its internal value, and if it does, it send a **true** message back to **GENERATOR** and notifies **WINNER** that there is a winner. The calls and the interfaces are shown in the diagram, but $\mathcal{I}^5$ *has no means to specify the dynamics of the execution just described.*

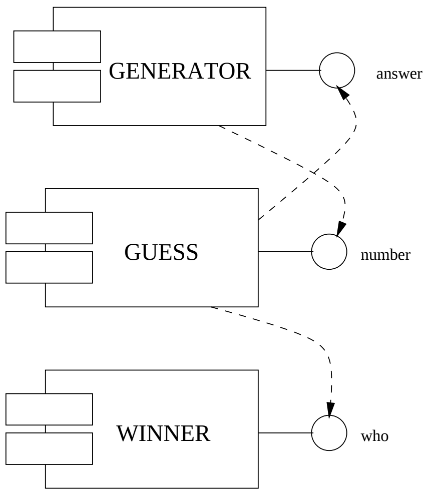

Figure 2. $\mathcal{I}^5$ Software *Interface* Graphical Specification.

Implementation. This level deals with the definition of implementation classes. Each component, node or connector type in *Interface* may be realized by zero or more classes in the *Implementation* (*realization* is the relation between a class and the type it implements). Implementation inheritance is defined at this level: a subclass inherits the implementation of its superclasses; however, it is not mandatory that a class realizing a subtype be a subclass of the class realizing the supertype. We use UML component diagrams for the graphical notation; we use a stereotype for the component classes and we represent them as shaded boxes.

The importance of the *Implementation* level is more apparent when different implementations are provided for the same component type. In the **GUESS-GENERATOR** example, we have two different classes of **GENERATOR**, a **NON-DETERMINISTIC** and a **SEQUENTIAL** generator; these names describe the way the components choose the numbers to guess. The software *Implementation* diagram for the example is shown Figure 3. Notice that the existence of different implementations for **GENERATOR** is clear, but *their internal difference is suggested only by their names.*

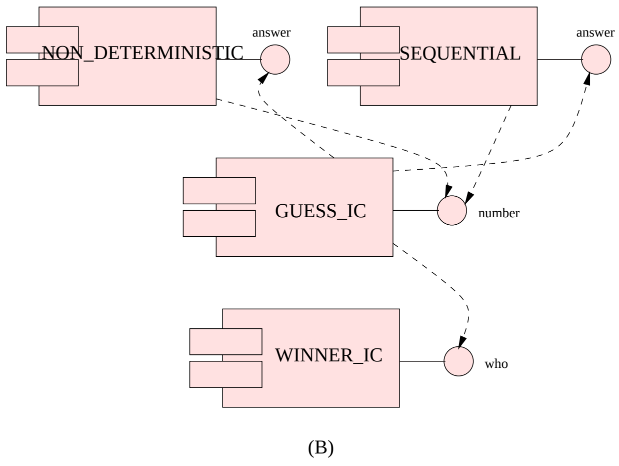

Figure 3. $\mathcal{I}^5$ Software *Implementation* Graphical Specification.

Integration. This level defines the dependencies that exist between component and node classes for deployment: a node class *supports* a component class, meaning that instances of a component class may be deployed to instances of a node class. These dependencies correspond to hardware requirements of the component classes.

Instantiation. In this level, the instance components, nodes and connectors that form part of the actual system are defined. Only instances of the classes defined in the *Implementation* level can be defined and they must follow the same communication patterns. Figure 4 shows the software *Instantiation* of the **GUESS-GENERATOR** example; there are two non-deterministic and three sequential generators and one instance of **GUESS** and one instance of **WINNER**.

Instances are named with an underlined lower case name, a colon and the name of the class they instantiate, as is standard for naming instances in UML diagrams (we do not instantiate types

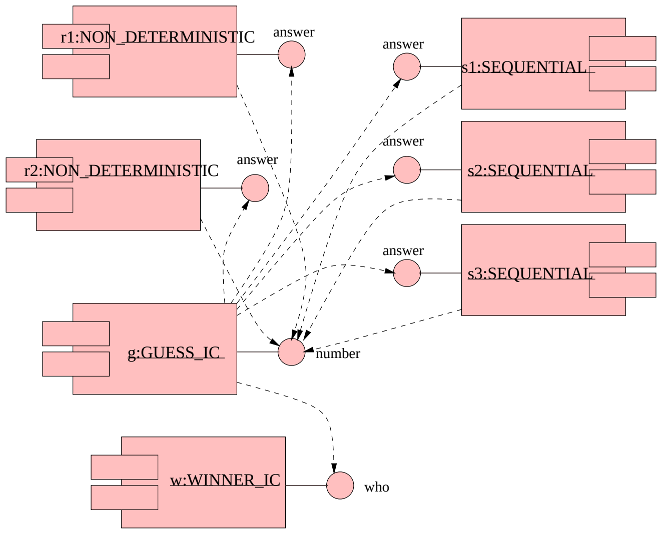

Figure 4. $\mathcal{I}^5$ Software *Instantiation* Graphical Specification.

as is standard in UML, but classes). The interaction occurs as follows: an instance generator sends a number to the guess instance, this component compares the received number with its internal value and if it is the same, it returns a `true` answer to the generator, and otherwise a `false`. Whenever there is a generator that guesses the number, the instance of `WINNER` is notified and `GUESS` chooses another number. $\mathcal{I}^5$ *does not provide means for specifying this dynamics*, and it cannot be shown in the diagram, but all of the possible messages are shown as calls to some interface.

Installation. This last level defines the complete deployment of instance components of the distributed application to instance nodes in the target network. Every instance identified in the *Instantiation* must be part of the *Installation*. Installation requirements such as fixing the location of certain components, or prescribing that two components must be deployed to the same or different nodes, are also defined at this level.

For simplicity of presentation, we make no references to hardware or network elements in our example. Thus we do not show the hardware parts of *Interface*, *Implementation*, and *Instantiation* levels and we do not specify the integration of software and hardware elements in the *Integration* and *Installation* levels.

2.2. Input/Output Automata

IOA is a language for specifying, programming and validating distributed systems [8] described as input/output automata. It has been applied to several real world applications with good success.

An automaton `A` is specified with a signature, `sig(A)`, consisting of the declaration of its input, output and internal transitions; a set of internal states, `states(A)`; a set of start states, `start(A)`; the definition of its transitions as state-transition relations, `trans(A)` (a subset of `states(A)` $\times$ `sig(A)` $\times$ `states(A)`); and an optional task partition.

The signature is the declaration of all of the automaton's transitions. There are **input**, **output**, and **internal** transitions. The definition of the transitions is given as a precondition and an effect; the precondition is a logic expression that enables the transition when it is true (an empty precondition is assumed to be always true). The internal state is defined by a set of state variables; the transitions usually modify the values of these variables as part of their effects; the effects is executed *atomically* to yield a post-state. The task partition is defined to assure fair executions, avoiding starvation of some enabled transitions.

In Figure 5 we show a `GUESS` automaton that receives a number `i` from another automaton `g`, and returns true to `g` if the input value matches its internal value and false if it does not. Whenever there is a `g` that guesses the value, `GUESS` also outputs the identification of the winner. We also include a specification of the `GUESS` automaton using IOA in Figure 6.

The signature of `GUESS` includes the input transition `number` and the output transitions `answer` and `who` shown in Figure 5, and it also defines an internal transition `choose_one`, not present in the figure. The state is defined by the internal `value` to be guessed, the identifier of the `sender` of the number, and two flags indicating when a number has been received, `ready`, and when the number has been `guessed`, respectively. Notice that `value` is initialized nondeterministically to a number between 0 and 10.

The input transition `number` is always enabled (it has an empty precondition), and whenever automaton `g` sends a new number `i`, it is compared to the internal `value`. If it is equal, `guessed` is set to true enabling the `choose_one` and `who` transitions. Whenever a new number is received, the variable `ready` is set to true and `sender` is assigned the identifier of the automaton that sent the number. The transition `answer` gets enabled when a number is received (`ready`); it sends a true or false value to `sender`, depending on the value of `guessed`. The transition `who` informs about the gen-

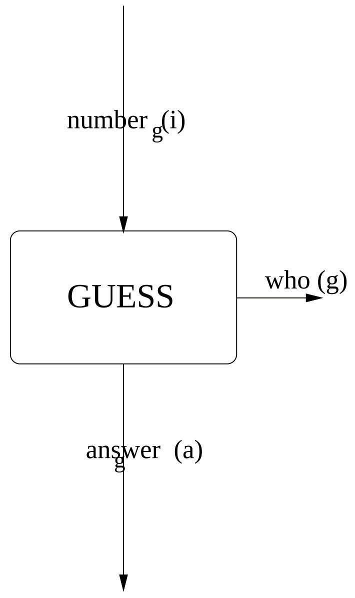

Figure 5. GUESS receives a number and says if it is the internal value.

erator that guessed the number, whenever this happens (`guessed`). The internal transition `choose_one` chooses non-deterministically a new number whenever the old one was guessed (`guessed`).

This simple example shows the main features of the specification of a single input/output automaton. However, more powerful specifications can be built composing different communicating automata or specifying families of automata through parameterization.

2.2.1 I/O Automata Composition

We can compose automata matching input transitions in one automaton with output transitions with the same name in another automaton, and by combining the states of the composed automata. Whenever a transition in one of the automata in the composition is executed, every transition in other automata in the same composition that has the same name is also executed. The combination of an input transition and an output transition with the same name can be considered an internal transition of the composition.

In Figure 8, the automaton WINNER has only one input transition, who, that matches the output transition in GUESS with the same name. Figure 8 also specifies WINNER using IOA. The internal state of the automaton is defined by the array score indexed by Index and containing integer elements; this array is completely

```
automaton GUESS (Index : type )
signature
    input number (i : Int, g : Index)
    output answer (a : Bool, g : Index),
           who (g : Index)
    internal choose_one ()
states
    value : Int := choose i where 0 ≤ i ≤ 10,
    ready : Bool := false,
    sender : Index,
    guessed : Bool := false
transitions
    input
        number(i,g)
            eff : if i = value
                    then guessed := true
                    fi ;
                    ready := true ∧ sender := g
    output
        answer (a,g)
            pre : a := guessed ∧ g := sender ∧ ready
            eff : ready := false
        who (g)
            pre : guessed = true ∧ g = sender
    internal
        choose_one ()
            pre : guessed = true
            eff : guessed := false;
                    value := choose i where 0 ≤ i ≤ 10
```

Figure 6. GUESS receives a number and says if it is the internal value.

initialized to zero. The effect of the who transition is to accumulate 1 to the score of the winning generator.

2.2.2 Parameterized Automata

Input/output automata can be defined with parameters, so a whole family of automata is actually defined with the same specification, one for each value of the parameter. Figure 7 shows a generic automaton GENERATOR; there is actually one automaton for each value of g.

The GENERATOR automaton is parameterized by g, meaning that there is actually one automaton for each value of g, but the set of g's is not specified either.

The semantics of the GENERATOR type is shown in Figure 9. Notice that the input transition answer and the output transition number correspond to the interface and the call in the *Interface* specification in $\mathcal{I}^5$. The IOA specification also provides the semantics of the GENERATOR, that is, whenever an answer is received, the generator is ready to choose another number between 0 and 10 and send it to GUESS.

We specify two different implementations for the

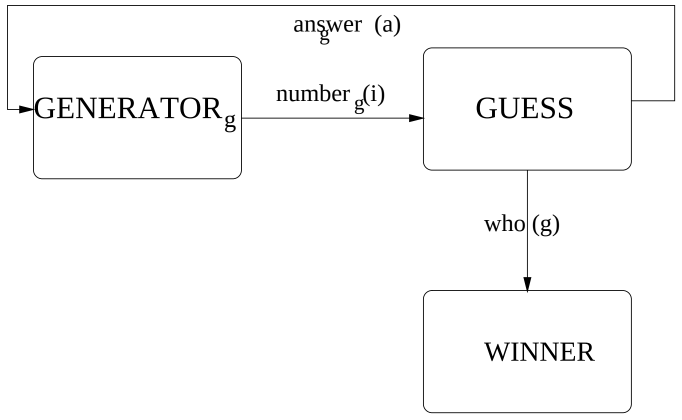

Figure 7. The **GENERATOR** is a parameterized automata.

```
automaton WINNER (Index : type )
signature
    input who (g : Index)
states
    score : Array [Index , Int]
    so that ∀ i : Index (score[i] = 0)
transitions
    input who (g)
        eff : score [g] := score [g] + 1
```

Figure 8. **GUESS** says who is the **WINNER**.

GENERATOR in Figure 10, one that chooses the number non-deterministically and another one that chooses the number sequentially. In Figure 10, we provide the specification of these two kinds of GENERATOR mentioned in Section 2.1. Notice that this differentiation of generators is not present in Figure 7 because it shows only the interface interaction of the automata but not the semantics of this interaction. The specifications of the two different implementations of GENERATOR share their name and signature, but they differ in their internal implementation. The parameter g in the signature is defined as **const**, meaning that the value of this parameter is constant for every instance automaton.

The family of GENERATOR automata have an output transition number that matches the input transition with the same name in GUESS, and an input transition answer that matches the output transition with the same name in GUESS. The answer transition has an identical implementation in both generators, assigning a true value to newtry, meaning that an answer has been received. The implementation of the output transition number is different in both generators: in the

```
automaton GENERATOR (type Index, g : Index)
signature
    input answer (a : Bool, const g)
    output number (i : Int, const g)
states
    newtry : Bool := true
transitions
    input answer (a,g)
        eff : newtry := true
    output number (i,g)
        pre : 1 ≤ i ≤ 10 ∧ newtry = true
        eff : newtry := false
```

Figure 9. The **GENERATOR** type semantics.

non-deterministic generator, the new number i is an integer number chosen non-deterministically between 0 and 10, and in the sequential generator, i is the number following the last one sent. In both cases, newtry is set to false, disabling the number transition so no other number is sent before a new answer is received.

Figure 11 shows the complete specification of the GUESS-GENERATOR example. The set of the generator identifiers Index is defined as an enumerated type. Notice that the set of identifiers corresponds to the instance components defined in Section 2.1.

Notice that the composition corresponds to a high level specification and assumes that GENERATOR automata are all identical and it does not consider the two implementations shown in Figure 10. We can assume we are using the type definition in Figure 9. We can also produce a similar composition at the implementation level by including the two implementations of GENERATOR. Using IOA methodology [12] it is possible to formally prove that the implementation composition correctly implements GUESS-GENERATOR with respect to its external behavior.

3. The Integrated Specification

In this section, we examine the steps that are required to build an integrated specification. The independent specifications using $\mathcal{I}^5$ and IOA presented in Sections 2.1 and 2.2, respectively, have elements that are unique to each language, and most importantly for our purposes, have elements that match to each other. Thus, it is possible to build an integrated specification by using the strengths of both languages. Our proposed integrated specification contains the five abstraction levels of $\mathcal{I}^5$, where each level is enhanced by the specification of the application's semantics using IOA.

```
automaton GENERATOR_SQ (type Index, g : Index)
signature
    input answer (a : Bool, const g)
    output number (i : Int, const g)
states
    value : Int,
    newtry: Bool := true,
transitions
    input answer (a,g)
        eff : newtry := true
    output number(i,g)
        pre : newtry;
              i := choose i where 1 ≤ i ≤ 10
        eff : newtry := false
```

```
automaton GENERATOR_ND (type Index, g : Index)
signature
    input answer (a : Bool, const g)
    output number (i : Int, const g)
states
    value : Int := 1,
    newtry: Bool := true,
    last : Int
transitions
    input answer (a,g)
        eff : newtry := true
    output number(i,g)
        pre : newtry = true;
              if last = 10 then i := 1
              else i := last + 1
              fi ;
        eff : newtry := false;
              last := i
```

Figure 10. Two implementations for GENERATOR: non-deterministic and sequential.

3.1. Matching Elements

Automata in IOA are defined by their name (with zero or more parameters), their signature, their state variables, and the semantics of their transitions. In $\mathcal{I}^5$, component types are characterized by their names, their interfaces and calls; component classes of the same type share their interface but may have different implementations; component instances are identical implementations with a differentiating name. Table 1 details the matching elements in IOA and $\mathcal{I}^5$, this correspondence is critical to understand the construction process of an integrated specification.

The correspondence shown in Table 1 establishes associations between the different modeling elements that are present in IOA and $\mathcal{I}^5$. These associations are critical to allow a software engineer or distributed system designer to utilize the two different specification languages in a complementary process in support of defining a distributed application.

No hardware elements of a distributed system are defined as part of input/output automata. So only the software part of the *Interface*, *Implementation*, and *Instantiation* levels of $\mathcal{I}^5$ can expect to share information with IOA specifications.

```
automaton GUESS-GENERATOR
type Index = enumeration of nd1, nd2, sq1, sq2, sq3
compose
    GUESS (type Index);
    WINNER (type Index);
    GENERATOR (type Index, g) for g : Index
```

Figure 11. Complete IOA Specification.

IOA element	$\mathcal{I}^5$ element
Automaton's name and signature	Component type
Input transition declaration	Interface
Output transition declaration	Call
Internal transition declaration	-
Parameterized automaton	Component type/class
Not parameterized automaton	Component instance
State variables	-
Transitions' semantics	-
-	Hardware elements

Table 1. Correspondence of I/O Automata and $\mathcal{I}^5$ definition elements.

3.2. $\mathcal{I}^5$ + IOA

In this section, we explain the way that the different elements in select levels of $\mathcal{I}^5$ match up with modeling elements in IOA. However, there is one important caveat for the discussion. Recall that unlike $\mathcal{I}^5$, there are no harware platforms or elements of a distributed system which can be explicitly defined as part of input/output automata. Thus, for the purposes of this paper, we concentrate on the software specification levels of $\mathcal{I}^5$ (*Interface*, *Implementation*, and *Instantiation*).

3.2.1 Interface

Generically, a type is defined by a combination of its interface and its semantics. In $\mathcal{I}^5$, the specification of a type is only given by its interface (the interfaces and the calls) with the semantics only suggested by the type's name. Clearly, this semantics is imprecise, and open for misinterpretation. In the definition of an IOA automaton, a name is provided, and input and output transitions are declared as part of its signature. If we consider the component types in the diagram in Figure 2, and the semantic specification of the automata in Figures 5, 8 and 9, we can see the coincidence of types and automata names, interfaces with input transitions, and calls with output transitions. Thus, we can combine the software *Interface* diagram of $\mathcal{I}^5$ and the corresponding specification of each automaton corresponding to its type to obtain an integrated software *Interface* specification. As a result, we begin to augment $\mathcal{I}^5$'s *Interface* with semantics supplied by IOA's automata.

3.2.2 Implementation

In the *Implementation* level of $\mathcal{I}^5$, we must specify the different implementation classes that realize each type identified in the *Interface* level. In Figure 10, we provide two different IOA implementations for **GENERATOR**: **GENERATOR_ND** and **GENERATOR_SQ**. These automata share the same input and output transitions with the type specification of **GENERATOR**. Moreover, it can be shown that these two implementations for the **GENERATOR** automaton type are forward simulations [8] of the **GENERATOR** type specified in Figure 9, meaning that every trace of the implementations is also a trace of the type. For example, there cannot be two consecutive **number** transitions without a **answer** transition in between, and the numbers sent are always integers between 1 and 10. Using this interpretation, these implementations correspond to classes in $\mathcal{I}^5$ terminology.

In the cases of the **GUESS** and **WINNER** automata, we provided a single implementation for each one, so the need for a distinct level of abstraction between $\mathcal{I}^5$ component type and class specification of these automata is not evident.

Combining the concepts of $\mathcal{I}^5$ and IOA in this case yields an integrated specification of the software *Implementation* level of $\mathcal{I}^5$, shown in the diagram in Figure 3, and enhanced with the IOA specifications in Figures 5, 8 and 10.

3.2.3 Instantiation

In the *Instantiation* level of $\mathcal{I}^5$, the actual instance components of the application are identified. The semantics of these instances is identical to the semantics of the classes they belong to, and they are distinguished with a different identifying name. As a result, the IOA specification of the complete system given in Figure 11 is equivalent to the combination of the *Instantiation* in Figure 4 and the integrated *Implementation* specification described in Section 3.2.2.

The progression that combines $\mathcal{I}^5$'s *Interface*, *Implementation*, and *Instantiation* with IOA automata from Figures 5, 8, 9, 10, and 11, yields an integrated specification where the abstraction capabilities of $\mathcal{I}^5$ can be augmented and complemented with the semantic capabilities of IOA.

4. Conclusions

The specification, design, and construction of a distributed application is a difficult task, complicated by the absence of a single model, language, or methodology that can be employed throughout all steps of the process.

In this paper, our proposed integration of $\mathcal{I}^5$ and IOA represents an important first step in supporting this difficult process, where the strengths of one specification language can offset the weakness of the other. We have shown that IOA provides the semantics and dynamics $\mathcal{I}^5$ lacks, while $\mathcal{I}^5$ provides abstractions that differentiate between types and classes, which is not as clearly supported in IOA.

We have also shown that $\mathcal{I}^5$ software *Interface* component types and *Implementation* classes can be augmented with semantics through the specification of corresponding automata using IOA. Forward simulation in IOA can be used to formalize the realization relation between $\mathcal{I}^5$'s types and classes. To accomplish this, we must prove that a class actually implements the semantics specified for the type it realizes: the class and the type must have the same input and output transitions, and every trace of the class must also be a trace of the type. Once this forward simulation is proven, the software engineer can reason using the simpler type specification rather than the more specific implementation. The work presented in this paper is an important step towards a combined specification language that spans multiple steps of the distributed application design and development process.

Our ongoing efforts continue to focus on a complete integration of IOA and $\mathcal{I}^5$. $\mathcal{I}^5$ provides an abstraction of hardware elements that is very important for the specification of distributed systems, especially for defining deployment. Consider, for example, the requirement that the **GUESS** and **GENERATOR** components must be deployed together, but the **WINNER** is a remote printer. This kind of requirements can be specified using the *Integration* and/or *Installation* levels of $\mathcal{I}^5$, and as a result the integrated specification is more expressive. However, there are no obvious analogs of this activity within IOA; rather, we must understand how this process in $\mathcal{I}^5$ can influence and complement IOA.

We are also pursuing the specification of the semantics of interface. There is still no way to specify the meaning of the inheritance in $\mathcal{I}^5$ without using the component types/classes semantics. Inheritance in IOA has been analyzed in [9], and *interface extension* seems to map nicely to $\mathcal{I}^5$'s interface inheritance, as well as *specialization* describes implementation inheritance, but additional work is needed to fully understand the correspondence and its implications.

References

[1] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodoly*, 6(3):213–249, July 1997.

[2] M. Cecilia Bastarrica, Scott Craig, Steven A. Demurjian, and Alex A. Shvartsman. Structural Specification of a Distributed System Using : In *Proc. of the 5 International Conference on Computer Science and Informatics, IC2000*, Atlantic City, NJ, February 2000.

[3] M. Cecilia Bastarrica, Steven A. Demurjian, and Alex A. Shvartsman. $\mathcal{I}^5$: A Framework for Architectural Specification of Distributed Object Systems. In *Proceedings of the 3rd International Conference On Principles Of DIstributed Systems, OPODIS'99*, Hanoi, Vietnam, October 1999.

[4] Frank Buschman, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Son Ltd., August 1996.

[5] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit*. Johen Wiley and Sons, Inc., first edition, 1998.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

[7] D. Garlan, R. Monroe, and D. Wile. ACME: An Architectural Interconnection Language. Technical Report CMU-CS-95-219, Carnegie Mellon University, November 1995.

[8] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A Language for Specifying, Programming, and Validating Distributed Systems. Technical Report User and Reference Manual, MIT Laboratory for Computer Science, Cambridge, MA, December 1997.

[9] Idit Keidar, Roger Khazan, Nancy Lynch, and Alex Shvartsman. An Inheritance-Based Technique for Building Simulation Proofs Incrementally. In *Proceedings of the 22nd. International Conference on Software Engineering, ICSE'2000 (to appear)*, Limerick, Ireland, 2000.

[10] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, pages 717–734, September 1995.

[11] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[12] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI Quart*, 2(3):219–246, 1989.

[13] N. Medvidovic, R. N. Taylor, and Jr. E. J. Whitehead. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28–40, April 1996.

[14] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pages 356–372, April 1995.

[15] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesni. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[16] J. M. Spivey. *Understanding Z*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1995.

[17] S. Vestal. Metah Programmer's Manual, version 1.09. Technical report, Honeywell Technology Center, April 1996.