Structural Specification of a Distributed System Using $I^5$

María Cecilia Bastarrica, Scott Craig, Steven A. Demurjian, Alex A. Shvartsman

Abstract

$I^5$ is an integrated framework for organizing, designing and documenting distributed systems configuration. It is a valuable tool mainly for large and complex distributed systems. We designed $I^5$, a language that supports the definition of distributed systems in an increasing level of detail and we formally specified it in [3]. We present the specification of a real world example, a distributed application for gathering data for a submarine, deployed over a network, showing the way $I^5$ is used in practice.

1 Introduction

- Defining the architecture of a distributed system is difficult because there are many features involved and many possible views.
- $I^5$ proposes an integrated framework where the system is defined in successive levels of abstraction.
- Each level is defined as a refinement of the one before where new details are added in an organized way.
- The $I^5$ language suggests a methodology for designing distributed systems.
- The final specification provides traceability to the system and is a sound basis for future maintenance and configuration management.
- We present the specification of an already developed and deployed system in terms of $I^5$. Rethinking the system in terms of types, classes and instances made it clearer that there is much more homogeneity in this system in terms of functionality (types) or implementations (classes) than previously noticed, when each instance component was thought of as a unique independent entity, and where similarities among different components was just implicit.

The distributed system specified in this document is conceptually based on the Virginia-Class submarine data distribution system, which is made up of several subsystems. In [2] we reported the specification of two subsystems, but we only include the Settings subsystem here. The data to be transferred among subsystems were categorized into data groups and for each data group interfaces were defined based on the CORBA services "Push-Style Communication with an Event Channel" [10]. To prevent polling by clients, servers used Typed Event Channels to distribute aperiodic notifications of changes to available data. Typed Event Channels were also used to distribute periodic updates and aperiodic signals.

There is no standard or widespread comprehensive methodology for designing and/or documenting distributed systems configuration. However, distributed systems tend to be larger and more complex every day, so design, documentation and management of these systems become very difficult. There is a careful design for distributed algorithms, but translating such algorithms to running components is a subtle task. Documenting the result is completely necessary for system evolution and maintenance.

We propose $I^5$, a five level specification language for defining distributed systems, describing both hardware and software parts in an integrated and symmetric way. $I^5$ has two equivalent notations, one textual using Z [11], and another graphical using UML's implementation diagrams [5].

$I^5$ is a specification language for defining distributed systems in five hierarchical dimensions or levels. It is inspired by the ideas in OSF's I4DL [1, 6]. $I^5$'s five levels are: interface, implementation, integration, instantiation and installation, describing the distributed systems' software, hardware and dependencies between software and hardware, in an integrated manner in an increasing level of detail. $I^5$ has a graphical notation based on the UML implementation diagrams [5] and an equivalent textual notation based on Z [11].
**Interface:** Includes the definition of the different *types* of components in the application and nodes and connectors in the network, as well as their dependencies and connections. The *Interface* level includes the inheritance of signatures in a similar way as OMG’s IDL [4, 8, 9].

**Implementation:** *Implementation* refers to the classes that realize the component types and node types defined in the *Interface* level. For software components there may be different realizations for the same functionality mainly because they may be implemented using different programming languages or algorithms, or they may be intended to run on different platforms. Node classes are the different computer classes that form part of the target network.

**Integration:** *Integration* establishes the requirements component classes have with respect to the class of nodes where they need to run.

**Instantiation:** The actual instances of the component and node classes are specified as part of the *Instantiation* level. Each instance is identified with a unique name within the system. In our specification we only address the static configuration.

**Installation:** *Installation* shows where each instance component is to be deployed over the instance nodes of the network identified in the *Instantiation* level. This level did not exist in OSF’s I4DL [6]. Installation requirements such as fixed locations for some components or the necessity of running two components in the same or different node, may also be expressed using this level of specification.

In figure 1 we summarize the specifications in each level and their relationships. Each horizontal level represents a specification level in $I^5$. The meaning of the arrows between the different parts of the specification is described in the figure.

![Diagram](image-url)

**Figure 1:** Definition Languages in $I^5$.

2 **Applying $I^5$**

In this section, we use $I^5$ to specify a distributed system consisting of suppliers, event channels, and consumers from the Settings subsystem. The following interfaces have been defined: Settings.cs, SettingsSignal.ev, and SettingsUpdate.ev. Interfaces ending with "cs" are client/server interfaces. Interfaces ending with "ev" are event interfaces used for "Typed Event Communication" [10]. Event channels and consumers both realize the event interfaces. When a supplier invokes an operation in an event interface realized by an event channel, the event channel invokes the same operation for every consumer that is connected to it.
2.1 Interface

For the software part of the Interface level ($I\_S$) we specify each component type with its type name, supertypes, interfaces provided and interfaces called in other component types. In our example there is no inheritance, so the supertypes are all empty sets. There are three component types: $Sett\_Cons$, $Sett\_EC$, and $Sett\_Supp$, defining the consumer, event channel and supplier, respectively.

As we will see in the following levels of specification, there are different classes of each of these component types, and also several instances of each one, but the interaction structure is already clearly stated at this level.

$I\_S$ provides graphical and textual equivalent specification languages. Figure 2 shows the specification of the $I\_S$ level in the graphical and textual form. The textual specification assumes we already have formally specified each component type, but we are not providing those specifications here.

\[
I\_S
\]

\[
\text{comp\_types} = \{ct : \text{Comp\_Type} \mid ct\.\text{type} \in \{Sett\_Cons, Sett\_EC, Sett\_Supp\}\}
\]

Figure 2: Settings Interface Software Specification.

The Interface level for the hardware ($I\_H$) defines the type for the nodes and the connectors in the network, as well as the way they are combined. The Sonar settings system comprises five different node types: $Cons\_Node$, $EC\_Node$, $Supp\_Node$, $Ethernet\_Hub$ and $ATM\_Switch$, and two connector types: $10\_Base\_T$ and $Fiber$. $10\_Base\_T$ connectors are used to connect $Cons\_Nodes$ to the $Ethernet\_Hub$ and $Fiber$ for all other connections.

For the complete specification of $I\_H$ we put together the node and connector types, as well as the way connectors connect the nodes. A connector between a node type and itself represent the connection between two or more nodes of the same type. Whenever there is more than one connection type between a pair of node types, it implies that the connection between nodes of those types may be one of the connector types shown. Figure 3 shows the graphical and textual specification for the Interface level for the hardware.

\[
I\_H
\]

\[
\text{node\_types} = \{nt : \text{Node\_Type} \mid nt\.\text{type} \in \{Cons\_Node, EC\_Node, Supp\_Node, Ethernet\_Hub, ATM\_Switch\}\}
\]

\[
\text{conn\_types} = \{ct : \text{Conn\_Type} \mid ct\.\text{type} \in \{10\_Base\_T, Fiber\}\}
\]

\[
\text{connects} = \{(10\_Base\_T \mapsto \{Cons\_Node, Ethernet\_Hub\}), (Fiber \mapsto \{Cons\_Node, ATM\_Switch\}), (Fiber \mapsto \{EC\_Node, ATM\_Switch\}), (Fiber \mapsto \{Supp\_Node, ATM\_Switch\}), (Fiber \mapsto \{ATM\_Switch\}), (Fiber \mapsto \{ATM\_Switch, Ethernet\_Hub\})\}
\]

Figure 3: Hardware Interface Specification.

As we can see from the textual specification, all node types and connector types are part of the network and the network is a connected graph.
2.2 Implementation

Component types are refined to component classes. The SettCons is realized in two classes, one intended to run on a Windows NT (SettConsIC2) and the other for a Unix machine (SettConsIC1). The other two component types have just one implementation each. Figure 4 shows the Implementation level of specification for the software ($I_2S$). Component classes are shown as shaded boxes.

Node types are realized into node classes and connector types into connector classes. The ConsNode type is realized in two different implementation classes: a Windows NT (ConsNodeIC2) and a Unix machine (ConsNodeIC1). There is only one implementation for the other node types. There is also only one implementation for each of the connector types.

The specification of the Implementation level for the hardware includes the sets of node and connector classes, as well as the way they are related. Figure 5 shows the Implementation of the network. Implementation classes are shown as shaded cubes.

Figure 4: Software Implementation Level.

Figure 5: Hardware Implementation Classes.
2.3 Integration

I^5 is intended to document the whole process from design to deployment of the distributed system. Instances of each component class are identical copies that run independently, but they all share implementation characteristics that may constrain the kind of machine they could run on. The specification of this relationship between classes of components and classes of nodes constraints the degrees of freedom for the final deployment and guides the deployment process, but it is also a sound documentation that can be used in the future when some components need to be relocated. In our example, each component class may only run on one class of node as show in figure 6, but this could have been different. Also notice that there are no component classes that may run on neither the ATMSwitch nor the EthernetHub, so no instance of the component classes will be deployed to instances of these classes of nodes.

![Figure 6: Integration of Implementation Dependencies.]

All component classes identified in the Implementation level form the domain of the supports relation, and only nodes classes also defined in Implementation are part of its range.

2.4 Instantiation

In the Instantiation level instances of the classes defined in the Implementation level are defined. Class definitions are used as templates for creating identical copies that are the instances. The Instantiation level adds an identification to each actual component, node or connector, but it does not add any functionality or implementation characteristics to the elements. What is also new in this level is the communication between component instances and the connection between node instances, that are specified as they actually happen.

In our example there are two instances of the SettConsIC2: host1: SettConsIC2 and host2: SettConsIC2. Following the UML standard, we name instances with an underlined lower case name followed by the type name (in our case we only instantiate classes). Whenever there is only one instance of a class, UML does not require a particular name to identify the instance, it is enough with the class name underlined; we preferred to give a nn name to instances of the SettConsIC1, SettECIC and SettSuppIC that are unique in the system.

In the specification of each instance component a name is given and the actual calls to other instance components' interfaces is also specified. Figure 7 shows the textual and graphical representation of the complete Instantiation level for the software.

![Figure 7: Instance Components of the Settings System.]

The hardware Instantiation is interesting because there are more than one instance for various node classes,
and they are connected in a non-trivial way. There are two instances of the ATMSwitch: switch1:ATMSwitch and switch2:ATMSwitch, and they are both connected, developing the loop edge specified in I2_H between ATMSwitch and itself. There are also two instances of SuppNodeIC: env:SuppNodeIC and supp:SuppNodeIC, and they are both connected to switch1:ATMSwitch. Both instances of ConsNodeIC2: host1:ConsNodeIC2 and host1:ConsNodeIC2, are connected to the only instance of the EthernetHubIC, nn:EthernetHubIC. Figure 8 shows the graphical and the textual specification of the Instantiation level for the hardware.

### Figure 8: Instance Nodes and Connectors.

#### 2.5 Installation

Installation defines the actual deployment of the system in two ways: the deployment requirements and the actual final installation, i.e. which instance component runs in which instance node. This final installation must satisfy a whole series of constraints imposed by the prior specification levels, as explained in point 7 in figure 1.

In [2] we had the requirements that some of the Settings subsystem component instances must be deployed together with some component instances of some other subsystems, and others separated, but there is no such requirements if we consider the Settings subsystem isolated.

### Figure 9 shows the complete deployment of the distributed application over the network. This is just one possible deployment, there could be another that also satisfies the requirements.
3 Conclusions

Even though the graphical notation is clearer for visualizing the structure of the distributed application, it gets more and more complex as the design process progresses. The textual specification on the other hand, helps very little in visualizing the structure of the system being design, but it gets a little longer but not significantly more complex. Having the possibility of switching between the two types of specification, and having changes made in one reflected in the other adds power to the specification technique since each designer can use the specification form more appropriate for each feature or the one he/she feels more comfortable with.

A complete integrated specification of the software and hardware comprised in a distributed system provides traceability and help in the future configuration management, because it is easy to see the impact a change will have over the system; e.g., adding a new instance node affects only the Instantiation and Installation levels, but adding a new component type affects the complete specification.

UML's features for modeling distributed systems—component and deployment diagrams—have not been fully utilized in practice yet [7], so our integrated language brings motivation for further developing this capabilities in UML. A tool that uses already defined and formalized UML diagrams takes advantage of the potential users training or familiarity with the notation improving usability.

References


