Integrated Notation for Software Architecture Specifications

María Cecilia Bastarrica [1], Sergio F. Ochoa[1], and Pedro O. Rossel[2]

[1] Computer Science Department,
Universidad de Chile,
Blanco Encalada 2120, Santiago, Chile,
{cecilia,sochoa}@dcc.uchile.cl,

[2] Departamento de Computación e Informática,
Universidad Católica del Maule,
Av. San Miguel 3605, Talca, Chile,
prossel@hualo.ucm.cl

Abstract. Currently, there are many ways to specify software architectures that address a wide variety of formality and completeness. Generally, the most formal and complete specifications are the most difficult to understand, compromising usability and applicability. On the other hand, informal specifications are usually easier to use and understand, but several design aspects remain underspecified and there is no way to check for consistency or to reason about the semantics of the specification. This paper presents an integrated notation for specifying software architectures that combines simplicity and formality. This notation involves a specification in three levels of abstraction: a graphical box-and-line diagram, typical from informal specifications that provides the structural information, an interaction behavior specification using Input/Output Automata, completely formal and consistent with the graphical level, and a basis of Larch traits describing the domain specific abstract data types. In order to show the applicability and usability of the proposed notation, this paper describes the use of the notation to specify the architecture of a highly complex mesh management tool. The proposed notation showed to be very helpful in encapsulating the tool complexity and allowing developers to model systems with the appropriate level of abstraction.

Keywords. Architecture definition languages, software architecture.

1 Introduction

Software design involves a great deal of art, but this artistic expression in the absence of rules results in chaotic design [3]. In the last years, software engineering has matured towards well-defined rules to specify and govern the structure and dynamics of software systems [8]. As a result, several patterns and modeling and specification languages were created and validated [23].

Software architecture design is the basis of product design and also one of the most critical parts. Several architectural patterns have been proposed to help design the architecture of software systems [4, 35]. If a software system has a well designed architecture, there is a high probability that the final product has a good overall design. On the other hand, if the product architecture is not well designed there is no possibility at all that the product would have a good design.

However, having a well-designed architecture is not enough to implement, maintain and extend a software product because if it is not well specified it will not be well understood. A good design is as important as its specification. The software architecture community has proposed several architecture definition languages (ADLs) to specify the structure and the dynamics of software architectures. Unfortunately, there are too many different ADLs and there is no clear sign that they will converge to a common standard notation to specify architectures [7] any time soon. Some ADLs focus on structure, others on connectors, some of them are domain specific and others are general, so it is difficult to choose the most appropriate among all the available choices. What is generally true for all ADLs is that formality and usability are qualities that seldom go together.

Specifying a system using a formal ADL is generally difficult and requires trained software architects. Sometimes, the software architecture design may be correct, but its specification is so complex that it cannot be easily understood and shared among the development team. To overcome this difficulty, many designers choose to use informal representations such as box-and-line diagrams or lately the UML, even though it is largely recognized as not an appropriate form of specifying software architecture design [2, 5]. As a consequence, an informal or inappropriate architecture specification weakens its effectiveness as a means for unambiguous communicating and analyzing software systems.

In this paper we present another way of specifying software architectures. We use already developed and validated specification languages but we organize them in a different way such that we take advantage of the formality while not resigning usability. We organize architectural specifications in three abstraction levels: structural, behavioral and domain specific abstract data types. The structural specification uses box-and-line diagrams. This would be semiformal if it were not controlled by the other two levels. The behavioral specification uses Input/Output Automata (IOA) [11, 20, 21]. This level provides an IOA specification for each box in the structural diagram and a combination of an input transition in one automaton and an output transition in the other for each line in the structural diagram. These two levels provide the most important elements in an architectural specification: structure and behavior. But we still have another lower specification level corresponding to the domain specific abstract data types. It is implemented using Larch traits [13, 14, 37]. This formalism allows us to define in all detail the domain specific components that are used in the IOA behavioral definition.

We have applied the proposed notation for the specification of a 3D mesh management tool that is being developed at the Computer Science Department in the Universidad de Chile since last year. We used our notation for specifying what was already programmed and also for specifying new components that still need to be developed. In this way we were able to reason about different features of the product, mainly about coupling between components and scalability issues.

In Section 2 we discuss work done on other ADLs and we compare them with our integrated notation. Following, Section 3 presents a complex mesh management tool that is currently being extended. The software architecture design of this tool was specified using the proposed notation and it is presented in this paper as an example that shows the applicability and usability of our proposal. Section 4 presents the integrated notation showing how each feature is used to specify the software architecture of the mesh tool. Finally, Section 5 presents the conclusions and the future work.

2 Related Work

For the last decade, there has been a big effort in developing ADLs in order to get better architectural design specifications. Examples of these ADLs are Rapide [19], Wright [1], C2 [24], SADL [26], ACME [10], UML [25], and Darwin [22]. Also IOA [12], even though it was not created as an ADL, provides most of the capabilities required from a common ADL. All these languages provide formality to architectural specifications, something lacking from box-and-line diagrams that are still common in professional practice. Formal specifications reduce ambiguities and promote reusability of architectural designs, and allows analysis and simulation. Architectural design is specified in terms of components, connectors and configurations that represent the structure and behavior of a software systems [23].

Rapide is a textual event-based language and toolset for specifying and simulating software architectures. This ADL provides executable features for composing systems out of component interfaces by defining their synchronization and communication interconnections in terms of event patterns. It has the ability to model distributed and dynamic systems. It represents components, and in a minor scale, connectors and configurations.

Wright uses a text-based representation that provides a precise and abstract meaning to an architectural specification, and allows the analysis of both the architecture of individual software systems and families of systems. In particular, work on Wright has focused on the concept of explicit connector types, on the use of automated checking of architectural properties, and on the formalization of architectural styles. Wright does not have a good support for configurations.

C2 is a message-based architectural ADL for building flexible and extensible software systems. A architecture in C2 is seen as a hierarchical network of concurrent components linked together by connectors (or message routing devices)

consistent with a set of style rules. C2 communication rules require that all communication between C2 components be achieved through message passing.

The architectural description language SADL is intended for the definition of software architecture hierarchies that need to be analyzed formally. This ADL can be used to specify both the structure and the semantics of an architecture by using a textual representation.

ACME is a generic ADL that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools. ACME is text-based, but there are some tools that generate a graphical representation of its design.

UML is a graphical language for specifying, visualizing, constructing, and documenting the artifacts of a software-intensive system. It supports multiple views of a system -both structural and behavioral- especially those included in [18]. Today many companies are using UML for architectural description, but formally it is not considered an ADL due to its lack of a formal semantic to represent the designs. It limits the development of validation tools and the architectural analysis. In addition, UML is less expressive than typical ADLs; for instance, UML does not distinguish components and connectors, and does not have a built-in notion of architecture style constraints. UML is useful to specify already designed software architectures, but it is not useful to model it [5].

Darwin is a declarative ADL intended to be a general purpose notation for specifying the structure of software systems composed of a set of software components and their interaction mechanisms. It advocates a constructive style of system design, which leads to a clear separation between program structure, computation and interaction. Darwin has both a graphical and textual representation. It has a solid theoretical background based on π-calculus.

Input/Output Automata is a text-based language that has been developed and applied at MIT for the last 15 years [20, 21]. It is a formalism for specifying asynchronous concurrent and potentially distributed components based on labeled transitions. IOA is not generally considered as an ADL, but it models components, connectors and configurations, therefore, it has most of the characteristics needed to be an ADL [23].

In summary, ADLs like ACME, Wright, and C2, are based on textual representations, which limits readability and understandability of the architectural design specifications. Text-based notations do not help recognizing and reusing high level abstractions as design and architectural patterns. In general, it is natural and efficient to use a graphical representation to aid in the design process [28, 34]. On the other hand, languages like UML and box-and-line that are not formally ADLs, are commonly used as design languages. These languages provides a graphical representation of the architectural design but they have problems avoiding ambiguities on the architectural design specification because they do not provide a formal semantic to represent components, connectors and configurations. It also limits their capability to carry out analysis and validation of architectural designs.

On the other hand, Darwin could provide a good framework for developing specifications of architectural designs. The main limitations of this ADL are that it does not represent connectors and it does not naturally represent hierarchical designs through component abstractions. This is an important limitation for scalability. Similar to Darwin, this paper presents a textual and graphical notation to specify software architecture designs. This notation provides a formal semantics to represent components, connectors, and configurations, therefore it can be considered an ADL. Also, the proposed notation is well conceived to model software architectures in a hierarchical way, which is very helpful for non-experimented software architects.

3 The Application Example: A Mesh Management Tool

There is a group at the Computer Science Department in the Universidad de Chile developing a mesh management tool based on research on mesh management algorithms carried out for the last 15 years. Each algorithm involved is very complex, so the complexity of a tool integrating about 20 different algorithms is even greater. There is a personal issue in the fact that people involved in mesh management research are not very much aware and familiar with good software engineering practices. They have already had a couple of experiences of letting a program grow until it is not possible to fix errors anymore and then the whole effort has aborted. We want a planned development that gives us some warranty that if we follow the plan, we will get what we expected.

Some components of the tool have already been developed such as a 3D tetrahedral mesh visualizer and an interactive refinement component. The generation of a first good quality tetrahedral mesh is currently under development. Figure 1 shows a screenshot of this partial implementation.

The algorithms necessary for generating, managing and refining geometric meshes in three dimensions are highly complex [15, 30]. We need highly competent and expensive people for the development and we need to establish a quality assurance methodology. If we also have the capability of reusing the implementation of all these algorithms in successive versions of the tool, these products will have a better opportunity of commercial success [6].

4 Integrated Software Architecture Notation

Two main goals guide our proposal of an integrated notation for software architecture specification: formality and usability. These two qualities tend to be opposed: the more formality we add into a specification, the less understandable it becomes; the more formality we require from a specification, the more difficult it is to build it. So we want a specification that is formal enough to avoid all ambiguity but in a way that developers do not become overwhelmed with notation. This specification should address structural and behavioral aspects of software architecture design so that we can understand clearly what the system does and

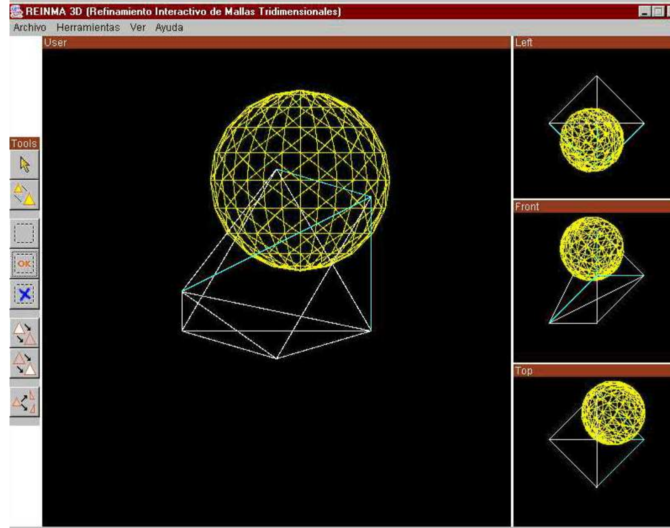

Fig. 1. Mesh tool implementation.

we can analyze its possible behavior, but only if we need to reason about the system.

In order to deal with these requirements we propose an integrated notation that specifies the software architecture design in three different levels of abstraction, as we show in Figure 2. We provide a high level structural specification using box-and-line diagrams, an intermediate behavioral specification using Input/Output Automata, and an abstract data type specification using Larch traits.

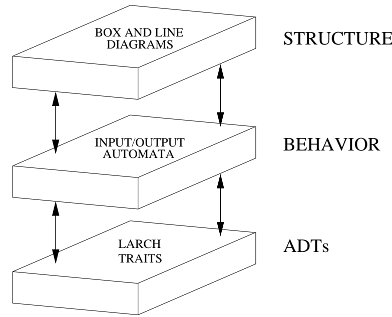

Fig. 2. Three levels of architectural specification

Generally, when people inspect a design, they first identify its structure [27]. It is natural for human beings to think about designs mainly focused on struc-

tures [29]. For this reason, we propose that the top highest level of the software architecture specification is focused on the structure of such design. Once the structure is understood, the behavior is naturally easier to analyze and understand. In summary, people that develop and review designs start with the structure, follows with the system behavior, and only then they can deal with the finer grained details about the design. In general, this way to proceed is natural for human being, and also for software architects.

4.1 Structural Specification

In the uppermost level, a box-and-line diagram is used. Each box corresponds to a software component and each line is an interaction between the components it connects. This kind of diagrams is the most typical informal diagram used for specifying architectures, so people are generally familiar with the notation. Even though it is a limited notation, there are many high level pieces of information that can be addressed very clearly: the type and number of components in the system, which ones communicate and which ones do not communicate, which components communicate with components outside the system and which ones only communicate with internal components, and the configuration of the whole system. All these elements are quite relevant for a software development team. Also, architectural patterns are generally defined using different flavors of box-and-line diagrams [4]; so using this notation makes it easier to recognize and reuse these patterns.

The internal details about the components are almost irrelevant for the structural description of the system. However, developers in charged of implementing or modifying each component, should know about these details, so they should have access to the component behavior specification.

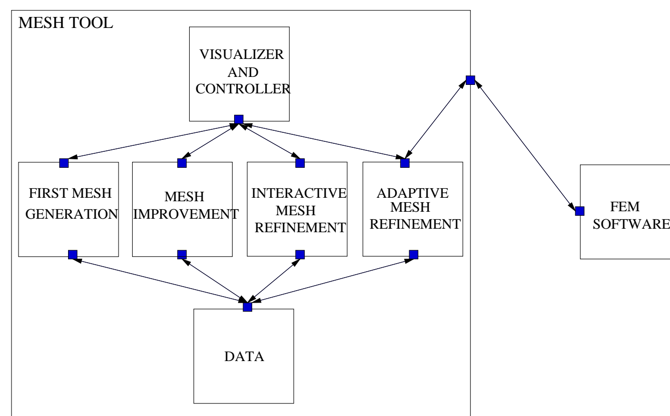

Fig. 3. High level architecture for a mesh tool.

Figure 3 shows the structural box-and-line diagram of the mesh tool. The following components are already implemented: *VISUALIZER_AND_CONTROLLER*, *INTERACTIVE_MESH_REFINEMENT*, and *DATA*. The rest of the components are either under construction or just planned. Table 1 provides a brief description of each component in the system.

Component	Description
VISUALIZER AND CONTROLLER	This components shows the tetrahedral mesh in its current state. It allows visualization operations: rotate user point of view, zoom, contract elements. As a controller, it allows the user to call other functional components [36]. This component holds a simplified version of the mesh including only geometric data.
FIRST MESH GENERATION	The tool needs a tetrahedral mesh to start executing all other components. The first mesh is built starting from a three dimensional surface mesh.
MESH IMPROVEMENT	The main goal of this component is to improve a mesh based on a certain quality criterion. There are many criteria for improving a tetrahedral mesh. The user chooses a criterion and this component will improve all tetrahedra that do not satisfy it [32].
INTERACTIVE MESH REFINEMENT	This component helps the user to improve certain mesh elements that are not good enough. For this purpose, he or she marks the elements to be refined using a refinement algorithm [31, 33].
ADAPTIVE MESH REFINEMENT	The elements that need to be refined depend on the physical phenomenon under study [31, 33]. So this component must carry out the refinement based on the information provided by a FEM (Finite Elements Method) software as is shown in Figure 3.
DATA	Decoupling data structures from algorithms is a very important feature in the mesh tool development philosophy. Data structures must be rich enough to provide the information necessary for each algorithm and sufficiently encapsulated to allow them to see only what other components need.

Table 1. Planned components in the mesh tool

The complete system, the *MESH_TOOL*, can also be seen as a compound box, similarly as the form used in [9]. In the same way, we can specify sub-boxes interconnected as a form of a finer grained specification for complex boxes.

4.2 Behavioral Specification

We specify the behavior of the overall system at the architectural level as the dynamics of component interactions identified in the structural level. For this purpose we use Input/Output Automata.

A system may be first defined at a high level of abstraction capturing only the essential requirements about its behavior, and then be successively refined until the desired level of detail is reached.

The notion of parallel composition, also included in the I/O automaton model, facilitates modular design and analysis of distributed systems. The parallel composition operator in the model allows the construction of large and complex systems from smaller and simpler subsystems and study their behavior in terms of the behavior of its components.

A suite of software tools—the IOA toolkit—is being developed to help in the design, analysis, and development of systems within the I/O automaton framework [17]. The toolkit includes syntax and static semantics analysis, an IOA simulator, a code generator and translators to a range of representations suitable for use with some theorem provers and model checking tools [16].

An I/O automaton is a simple type of state machine in which transitions are associated with named *actions*. The actions are classified as either *input*, *output*, or *internal*. Inputs and output transitions are used for communication with the automaton's environment, whereas internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control, while the automaton itself can control which output and internal actions should be performed. Table 2 describes the essential parts in an I/O automaton definition.

Element	Description
signature	lists the disjoint sets of input, output, and internal actions of an automaton A
states	a (not necessarily finite) set of *states*, usually described by a collection of state variables
start states	a non-empty subset of the set of all states
transitions	triples known as *steps* or *transitions* of the form (state, action, state)
tasks	an optional set which partitions the internal and output actions of A

Table 2. Elements in an IOA definition of an automaton A

An action π is said to be *enabled* in a state s if there is another state s' such that (s,π,s') is a transition of the automaton. Input actions are enabled in every state. That is to say that automata are not able to block input actions from occurring. The *external* actions of an automaton consist of its input and output actions.

We can specify the behavior of the *VISUALIZER_AND_CONTROLLER* as an Input Output Automata as is shown in Figure 4. Only the interactions between the *VISUALIZER_AND_CONTROLLER* with the *FIRST_MESH_GENERATION* and the *INTERACTIVE_MESH_REFINEMENT* have been included in the IOA specification in Figure 4. The interactive refinement is already built and the component that builds the first mesh is currently under construction, so we needed these interactions. As soon as *MESH_IMPROVEMENT* and *ADAPTIVE_MESH_REFINEMENT* start to be built, we will need to evolve the *VISUALIZER_AND_CONTROLLER* IOA specification to include the appropriate interactions.

The *VISUALIZER_AND_CONTROLLER* keeps track of the current mesh; it is defined as part of its internal state as a *TetSet*, that is a set of tetrahedra. This data type is used in the IOA specification but it is defined elsewhere (see Section 4.3). The mesh is initiated as an empty set, so we assume that the first operation should be to generate a mesh. The variable *option* lets us know the current status of the automaton in order to synchronize all transitions; initially *option* is *available*.

```
automaton VISUALIZER_AND_CONTROLLER
    uses TetraSet
    type Status = enumeration of wait_generate, wait_refine, available
    signature
        output generate_mesh()
        input generate_done(new_mesh : TetSet)
        internal mark(ts : TetSet)
        output int_refine_mesh(ts : TetSet)
        input int_refine_done(refset, ext_marked : TetSet)
    states
        option : Status := available
        mesh : TetSet := {}
        marked : TetSet := {}
    transitions
        output generate_mesh()
            pre option = available
            eff option = wait_generate
        input generate_done(new_mesh)
            eff mesh := new_mesh
                option := available
        internal mark(ts)
            pre option := available
                ts ⊆ mesh
            eff marked := ts
        output int_refine_mesh(marked)
            pre option = available
                marked ≠ {}
            eff option = wait_refine
        input int_refine_done(refset, ext_marked)
            eff mesh := mesh - ext_marked ∪ refset
                option := available ∧ marked := {}
```

Fig. 4. IOA Specification for the Visualizer

As we mentioned in Section 4.1, we can see the complete *MESH_TOOL* as a compound automata. The composition is made by identifying all transitions with equal name: whenever an output transition is fired in one of the automata, all input transitions with the same signature (name and parameter list) in other automata are also fired; this is the way of specifying the behavior of the connectors in the architecture.

The set of all interacting components forms the architecture configuration. This is achived by composing all the automata corresponding to components in the system. We provide the compound specification of the mesh tool in Figure 5. Whenever we build a new automaton by composing a set of other automata, we can choose to rename pairs of input and output transitions in the automata set as an internal transition of the compound automaton. In this way, this transition is not an external action anymore and we can better encapsulate automata deciding what is seem from the outside and what is not. In our example, we can deduce that the only external actions of the complete *MESH_TOOL* should be those by which the *ADAPTIVE_MESH_REFINEMENT* interacts with the FEM software.

```
automaton MESH_TOOL
compose
    VISUALIZER_AND_CONTROLLER;
    FIRST_MESH_GENERATION;
    MESH_IMPROVEMENT;
    INTERACTIVE_MESH_REFINEMENT;
    ADAPTIVE_MESH_REFINEMENT;
    DATA
```

Fig. 5. Mesh tool specified as a compound automata

4.3 Domain Specific Abstract Data Types

In order to have a completely formal definition of a system architecture we need to formally define its structure and behavior, as we did in Sections 4.1 and 4.2. However, as has been our motivation from the beginning, in each of these steps we chose to hide some lower level information that could be defined elsewhere and that would only add complexity to the definition at hand. This is the case of the domain specific abstract data types (ADTs). Components defined as automata communicate by sending messages of a certain type, and their internal state is also defined in terms of state variables of a certain type. In complex domains these types may be complex too. We used Larch [14] to specify these ADTs mainly because it has already been used in conjunction with Input/Output Automata [11] and there are many primitives that make it easier to integrate both notations into a unique specification.

The *trait* is the basic unit of specification in the Larch Shared Language (LSL). A trait introduces some operators and specifies some of their properties. Sometimes the trait defines an abstract type. LSL specifications define two kinds of symbols, *operators* and *sorts*. The concepts of operator and sort are similar to "procedure" and "type" in a programming language. Operators stand for total functions from tuples of values to values. Sorts stand for disjoint non-empty sets of values, and are used to indicates the domains and ranges of operators [14].

A Larch specification is formed by four basic parts introduced by the keywords *includes*, *introduces*, *asserts* and *implies*. Table 3 provides a brief explanation of each of these terms.

Figure 6 shows the Larch specification of the *TetraSet* trait. This ADT is used in the *VISUALIZER_AND_CONTROLLER* automaton specified in Figure 4.

The definitions for the included traits *SetBasics* and *DerivedOrders* are assumed to be those provided in [14]. We are also assuming that there exists another trait *Tetra* that defines a tetrahedron as an ADT. The set operations included in the *introduces* part of the trait are the only ones that we need, that means that a *TetSet* is formed by elements *Tetra* and thus only elements of the sort *Tetra* can be members of a *TetSet* ($\in$), and that union ($\cup$) and set difference (-) are operations between *TetSet*s. In the *asserts* part, the axioms that rule the defined operations are stated. Finally, in the *implies* part we include

Clause	Description
includes	This clause allows the reuse of traits when specifying more complex ADTs. Operators are specified in a separate trait that is included by reference using the *includes* clause.
introduces	Declares a set of operators (function identifiers), each with its *signature*(the sorts of its domain and range). Every operator used in a trait must be declared.
asserts	Generaly, it represents the *body* of the specification. It contains equations (two terms of the same trait, separated by = or ==) between terms containing operators and variables. This equations constrain the operators.
implies	It is useful for theory containment (that a specification has intended consequences). It enables specifiers to include information they believe to be redundant, either as a check on their understanding or to call atention to something that a reader might otherwise miss.

Table 3. Clauses in a Larch definition of a trait

certain statements for clarifying notation, provided that we are using included definitions that are not necessarily at hand for the user.

5 Conclusions and Work in Progress

The architecture of a system defines its high-level structure as a collection of interacting components. Many software architects use informal box-and-line diagrams to describe architectures. Unfortunately, informal diagrams and descriptions are highly ambiguous. Consequently, it is virtually impossible to answer with any precision most of the questions that arise during system development. Recognizing the deficiencies of using ad-hoc and informal notations to describe architecture, the software engineering research community has developed architecture definition languages. ADLs have well-defined semantics and tools for parsing, compiling and analyzing specifications of software architecture designs. Most of them are text-based. Also, the more formal and complete these specifications are the more difficult to specify and understand these architectural designs become, compromising usability and applicability of the ADL. The software architects need an ADL that is complete and formal enough to avoid ambiguities in the architectural design specifications, but also easy to use and understand so that it does not become an obstacle to specify, extend, or modify a software system.

In order to deal with these challenges, this paper presented an integrated notation for specifying software architectures that combines simplicity and formality. This notation involves a specification in three levels of abstraction: a graphical box-and-line diagram to specify structure, an IOA specification to represent behavior, and a basis of Larch traits to describe the domain specific abstract data types. Box-and-line diagrams are quite popular and most developers understand what they mean. However it is also well known that they do say almost nothing about the behavior. For that reason the proposed notation includes a IOA specification for each box-and-line component, in order to represent their behavior. IOA is a formal modeling language that has been used for many

```
TetraSet: trait
    includes
        Tetra,
        SetBasics,
        DerivedOrders (TetSet, ⊆ for ≤)
    introduces
        __∈ __: Tetra, TetSet → Bool
        {__}: Tetra → TetSet
        __∪ __, __− __: TetSet, TetSet → TetSet
    asserts
        ∀ e: Tetra, s1, s2: TetSet
            e ∈ (s1 ∪ s2) == e ∈ s1 ∨ e ∈ s2;
            e ∈ (s1 − s2) == e ∈ s1 ∧ e ∉ s2;
            s1 ⊆ s2 == s1 − s2 = { }
    implies
        AbelianMonoid (∪ for ∘, { } for unit, TetSet for T),
        JoinOp (∪, { } for empty),
        MemberOp ({ } for empty),
        PartialOrder (TetSet, ⊆ for ≤)
        TetSet generated by { }, {__}, ∪
        ∀ e: Tetra, s1, s2: TetSet
            s1 ⊆ s2 ⇒ (e ∈ s1 ⇒ e ∈ s2)
        converts
            ∈, ∉, {__}, ∪, −, ⊆
```

Fig. 6. Larch Specification for TetraSet

years to specify concurrent systems behavior. IOA has a natural integration with Larch traits, helping encapsulating required ADTs details.

The integrated notation is able to model components, connectors, and configurations through a formal specification in order to represent the structure and behavior of software systems. It does not avoid the difficulty of building a completely formal specification, but it organizes it in a way that not everybody needs to be involved with the technical details. All the semiformal diagrams that are shared among the development team can be proved to be sound and consistent with the expected structure and behavior.

In order to show the usability and applicability of the proposed notation, this paper presented the architecture design specification of a complex mesh management tool that has been partially implemented. In the future we intend to create a tool to simulate and analyze architectures specified with the integrated notation. Further, it is needed to formalize the hierarchical box-and-line specification in conjunction with IOA composition to enhance this integration and take more advantages of it.

Acknowledgements

The work of María Cecilia Bastarrica has been partially funded by project I-01-2/2001 of the Dapartment of Development and Research of the Universidad de Chile.

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. Hedley Apperly. *The Component Industry Metaphor*. Addison Wesley, 2001. In Component-Based Software Engineering, chapter 2, George Heineman and William Councill editors.
3. Marteen Boasson and Hollandse Signaalapparaten. The Artistry of Software Architecture. *IEEE Software*, 12(6):13–16, November 1995.
4. Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Son Ltd., August 1996.
5. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures. Views and Beyond*. SEI Series in Software Engineering. Addison Wesley, 2002.
6. Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, first edition, August 2001.
7. Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida, 2002.
8. Amnon H. Eden. Directions in Architectural Specifications. In *Software Architecture Recovery and Modelling, in: the 8th Working Conference on Reverse Engineering (WCRE)*, Stuttgart, Germany, October 2001.
9. C. Gane and T. Sarson. *Structured Systems Analysis*. Prentice-Hall, 1979.
10. D. Garlan, R. Monroe, and D. Wile. ACME: An Architectural Interconnection Language. Technical Report CMU-CS-95-219, Carnegie Mellon University, November 1995.
11. Stehen J. Garlan, Nancy A. Lynch, and Mandana Vaziri. IOA: A Language for Specifying, Programming and Validating Distributed Systems. Technical report, MIT Laboratory for Computer Science, December 1997.
12. Stephen J. Garland and Nancy A. Lynch. The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems. Technical Report MIT/LCS/TR-762, MIT Laboratory for Computer Science, Cambridge, MA, August 1997.
13. J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(5), 1985.
14. John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag Texts and Monographs in Computer Science, 1993.
15. Marc Halpern. Industrial Requirements and Practices in Finite Element Meshing: A Survey of Trends. In *Proceedings of the 6th International Meshing Roundtable'97*, Park City, Utah, October 1997. Sandia National Laboratories.
16. IOA. IOA Language and Toolset, 2002. `http://theory.lcs.mit.edu/tds/ioa/`.
17. Dilsun Kirli Kaynar, Anna Chefter, Laura Dean, Stephen Garlan, Nancy Lynch, Toh Ne Win, and Antonio Ramírez-Robredo. The IOA Simulator. Technical Report MIT-LCS-TR-843, MIT Laboratory for Computer Science, July 2002.
18. P. Krutchen. Architectural Blueprints - The "4+1" View Model of Software Architecture. *IEEE Software*, 12(6):42–50, November 1995.

19. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, pages 336–355, April 1995.
20. Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
21. Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI Quart*, 2(3):219–246, 1989.
22. Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3–14, October 1996.
23. Nenad Medvidovic and Richard Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.
24. Nenad Medvidovic, Richard Taylor, and E. James Whitehead Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28–40, April 1996.
25. S. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Addison Wesley, 2002.
26. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pages 356–372, April 1995.
27. K. Mullet and D. Sano. *Designing Visual Interfaces: Communication Oriented Techniques.* Sunsoft Press (Prentice Hall), 1995.
28. J. Nosek and I. Roth. A comparison of formal knowledge representation schemes as communication tools: Predicate logic vs. semantic network. *International Journal of Man-Machine Studies*, 33:227–239, 1990.
29. Joseph Novak. *Learning, Creating, and Using Knowledge: Concept Maps as Facilitative Tools in Schools and Corporations.* Lawrence Erlbaum Associates, 1998.
30. National Science Foundation Information Technology Research (NSF/ITR). Adaptive Software Project, 2001. `http://www.erc.msstate.edu/ jcollins/ITR/-index.html`.
31. María Cecilia Rivara. Design and Data Structure of Fully Adaptive Multigrid, Finite-Element Software. *ACM Transactions on Mathematical Software*, 10(3):242–264, September 1984.
32. María Cecilia Rivara. New Longest-Edge Algorithms for the Refinement and/or Improvement of Unstructured Triangularions. *International Journal for Numerical Methods in Engineering*, 40:3313–3324, 1997.
33. María Cecilia Rivara and Cristian Levin. A 3-D Refinement Algorithm Suitable for Adaptive and Multi-Grid Techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.
34. D. Roberts. *The Existential Graphs of Charles S. Peirce.* The Hague, Mouton & Co. N. V., 1973.
35. D. Schmidt, M. Stal, H. Rohner, and F. Buschmann. *Pattern Oriented Software Architecture. Vol. 2. Patterns for Concurrent and Networked Objects.* John Wiley & Sons, 2000.
36. Felipe De Toro. Interfaz Orientada al Refinamiento Interactivo de Mallas Tridimensionales, 2002. Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Departamento de Ciencias de la Computación.
37. Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, October 1997.