ODOG

Framework for Programming Concurrent Platforms

Version 1.0

February 2008

1. Introduction

ODOG is an open-source software framework for developing executable specifications composed of concurrent message passing components. There are other similar commercial and open-source projects available[1]. However the ODOG framework was written from scratch, which specific goals and requirements.

Borrowing from the terminology of the *Linda* programming model, an executable specification is composed of atomic components described in some (not necessarily programming) language, called the host language, and composite components which are a composition of other components described by a coordination language. One of the goals of ODOG is to seamlessly support several host and coordination languages.

The syntax of the coordination languages is a graph, where the nodes are components and edges are communication paths between components. Hence, a coordination language may have a graphical notation. The semantics of a coordination language specifies concepts, rules and restrictions for the composition of components. In particular, how components exchange data and notions around the "execution" concept. A particular semantics for a coordination language is also called an **interaction semantics** (ISem).

The usefulness of an infrastructure such as ODOG comes from the fact that individual languages are appropriate for a set of problems that they were designed to address, and that systems, in particular embedded systems, have a mix of necessities. Hence, one needs to "glue" different design techniques under the same design flow. Developing a new coordination language means designing its semantics, and once it is done, incorporating it in ODOG is easy. Hence, it can be used to rapidly testing new ideas regarding implementation of design languages.

1.2 Objectives

It is not the intent of the ODOG framework to applicable under any methodology (i.e, a miracle), but rather to provide an adequate support for experimenting with different solutions based on concurrent components and executable specifications. Besides allowing for the testing new ideas, it should be used by designers in useful experiments. Hence, it must have a good usability and necessary features. The main goals for ODOG are:

1. Allow "fast" execution of specifications;
2. Implement important ISem and support the addition of new ones;
3. Support for various host languages;

1 see National Instruments Labview,

4. Have a validation flow, including debuggers, static rule checking, dynamic rule checking and links to external verification tools;
5. Be usable within existing design flows;
6. Provide tools for creation of correct-by-construction components.

An important decision of the framework was how to implement the coordination language syntax and the various ISems. Other infrastructures use a library-based approach, that is, enhance a programming language with methods for tasks such as creation of the graph that describes the composition of components, data communication, execution control etc. These methods are used by the atomic components and when compiled together with the library, an executable code for the specification is created. I believe that the main deficiency with this approach is that if one wants to use the specification as a starting point for the implementation of the system described, it will be necessary to translate all the library into the desired target. Sometimes, it may be even necessary to do translations between languages, a very problematic task.

Instead of the library-based solution, the syntax of a coordination language in ODOG is described by an abstract tree-like structure, that can be easily recorded in XML. All the syntactic elements of a specification, like communication points, parameters of components, connections, hierarchy, etc, are described independently of any language. In other words, atomic components are written in the desired (implementation) language, and by using a template together with a code-generator, all the necessary extra code for implementing the semantics of the coordination of components is produced. There are three major benefits of this approach:

- the result is a specification that can be directly deployed, without any additional translation;
- no unnecessary code is produced, therefore no extra overhead on execution time;
- by having a language independent syntax, several different host languages can be used;

1.3 The ODOG framework

The following sections of this document will describe the components of the of the framework. They are:

1. Graph structure for describing the syntax of coordination languages;
2. Static rule checker;
3. Support for code generation;
4. GUI front-end for editing specifications;

5. Host languages;
6. Interaction semantics
7. Platforms

In companion with this manual, the *Getting Started* document illustrates the use of the framework with examples.

2. Syntax

This section describes the main syntactical elements of a specification. These elements capture the structure of atomic and composite components. As will be seen, they are related forming an acyclic graph structure. A persistent representation for this syntax can therefore be realized in XML. Common to all syntactic elements is the name field, which is used for identification.

2.1 Interface

The name of the component, together with data ports and attributes forms its **interface**. The interface provides a separation between the internals of the component and the "outside world" of the specification (or eventually the environment).

A data port is by where all data is received and send by a component. A data port must have a type, and may have attributes. Data ports are classified as being either input (receive data) or output (send data).

An attribute is a tag that can be associated with some other element, in order to add information to it. As mentioned, components and ports may have attributes. The other attributable elements are version (atomic and topology), connection and method.

Every attribute is classified as:

- visible or invisible;
- with or without data;
- static or dynamic.

The visibility of an attribute is defined with respect to the internals of the component. An attribute tagged as visible can be seen by the host language. An invisible attribute can only be accessed during code generation.

An attribute may have an associated data. If it is classified as static, its value cannot be changed during execution of the specification. Otherwise it can. Attributes can contain a default value. The valid classifications for an attribute are: visible, with data, static or dynamic; invisible, with or without data, static.

The interface of a component provides the information regarding its relationship with the environment where it is supposed to operate. Behavior must still be specified to produce a useful component. For an atomic one, a host language is employed. For a composite component, this is accomplished with a composition of other components. This composition is also called a **topology**.

The first decision when creating a new component is how its interface should be. An entire specification can be created containing only components described by their interfaces. Of course, for execution, it is necessary behavior. This separation between interface and behavior is analog to information hiding in object oriented environments.

2.2 Atomic component element

Such component is represented by a distinct syntactical element. It contains the interface elements plus versions elements. The version element is used to capture the behavior of the component, and other useful informations. A component may have one or more versions. However, when code generation is performed one must be choosed.

2.3 Atomic component version

Instead of describing the behavior directly by an artifact in the chosen host language, it is advantageous to break it into a number of elements. These elements are grouped in a version. It is composted of:

- data port;
- attribute;
- value;
- method.

The interface of a component may be further extended by a version by listing additional data ports and attributes.

For the attributes that are tagged as with data, a value element specifies its data value. It overrides the default value of the attribute, if present. For the same value element, more than one attribute can be selected. All of them will have the same value.

A method is an element that contains an artifact in the host language for describing the behavior of the component. There are four possible methods: *init*, *compute, finish* and *fixpoint*. The init method is called only once at the beginning of the execution of the specification. The finish method is analog to it, but called when execution terminates. The compute and fixpoint methods may be called several times, depending of the ISem associated with the topology where the component is instantiated. The fixpoint is always called after the compute method. At least, the init, compute and finish methods are supported by any ISem. The fixpoint may or may not be allowed.

2.4 Topology

The topology element represents the composition of other components. It is composed of:

- component instance;
- connection;

- exported port;
- attribute;
- version.

A component instance is a reference to another component, that should be inserted in the respective topology. A component instance has an extra name field for its instance name. Only the interface of each instantiated component is available for the topology element.

A connection is a relation containing exactly one output port and one or more input ports. It defines a particular data path of the topology. Component instances, together with topologies partially forms the behavior of the component.

An exported port is a port of some component instance contained in the topology that is tagged to be seen at the composite component interface that the topology defines. Therefore, either a data or another exported port can be referenced. It inherits the data direction of the referenced (data) port.

2.5 Topology version

The elements of a topology version are:

- component instance;
- connection;
- attribute;
- exported port;
- value;
- version definition;
- method.

The behavior of a topology may be extended by a version by adding extra component instances and connections. Its interface may also be incremented, with new attributes and exported ports.

The value element have the same function as in the atomic version. However, the attributes that can be associated to a value located at the topology version are the ones defined in the topology element, the current topology version, and all attributes of the components instantiated in the topology and the current topology version.

Version definition is an element that associates for a contained component instance, one of its version. Therefore, in order to fully determine the behavior of a topology, one version definition element must exist for each contained component instance.

A method is analog to the one contained in the atomic version, but only the init and finish methods are allowed.

2.6 List and relations of syntactical elements

The following two tables contains each element of atomic and composite components, respectively. For each element, the list of contained elements is given. As can be seen, this relation is analog to an acyclic graph.

Element	Contains
Atomic Component	Data port, Attribute, Version
Data Port	Attribute
Attribute	Attribute Classification, Value
Attribute Classification	
Version	Data Port, Attribute, Value, Method
Value	
Method	Attribute

Element	Contains
Topology	Component Instance, Attribute, Exported Port, Connection, Version
Component Instance	Value
Attribute	Attribute Classification, Value
Attribute Classification	
Exported Port	
Connection	Attribute
Version	Component Instance, Attribute, Exported Port, Connection, Value, Method, Version Definition
Value	
Method	Attribute
Version Definition	

The value element determines the associated data expression of an attribute. However, for a component with behavior fully determined, there are three places where a value can reference the same attribute: at a component instance, version or the default value. To decide which value element should be used, a precedence relation is imposed: a value of an instance has higher priority, followed by one of a version of that component and the default value has the lowest priority.

2.7 XML representation and examples

The XML representation follows the relations described in the previous section. The DTD for both atomic and composite components can be found in the *dtds* directory of the ODOGWorkspace.

The following code represents the simplest atomic component possible, one without any element.

```
<atomicComponent name="AATest1">
</atomicComponent>
```

Another component with all elements is illustrated by the following code.

```
<atomicComponent name="AATest22">

<dport name="input" isInput="true" isOutput="false">
  <portType>
    <attribute name="type">
    <attrClassification visible="false" hasData="false" static="false"/>
    </attribute>
  </portType>
</dport>

<dport name="output" isInput="false" isOutput="true">
  <portType>
    <attribute name="type">
    <attrClassification visible="false" hasData="false" static="false"/>
    </attribute>
  </portType>
</dport>

<version name="ver1">
  <dport name="inputX" isInput="true" isOutput="false">
    <portType>
      <attribute name="type">
      <attrClassification visible="false" hasData="false" static="false"/>
      </attribute>
    </portType>
  </dport>

  <dport name="outputX" isInput="false" isOutput="true">
    <portType>
      <attribute name="type">
      <attrClassification visible="false" hasData="false" static="false"/>
      </attribute>
    </portType>
  </dport>

 <value type="String" valueExpr="integer"
associatedAttribute="AATest22.input.type"/>
```

```
 <value type="String" valueExpr="double"
associatedAttribute="AATest22.output.type"/>
 <value type="String" valueExpr="double"
associatedAttribute="AATest22.ver1.inputX.type"/>
 <value type="String" valueExpr="double"
associatedAttribute="AATest22.ver1.outputX.type"/>
 <value type="integer" valueExpr="1"
associatedAttribute="AATest22.ver1.atribute1"/>
 <value type="integer" valueExpr="2"
associatedAttribute="AATest22.ver1.atribute2"/>
 <value type="integer" valueExpr="1"
associatedAttribute="AATest22.ver1.m1.atribute1"/>

  <attribute name="atribute1">
  <attrClassification visible="false" hasData="false" static="false"/>
  </attribute>

  <attribute name="atribute2">
  <attrClassification visible="true" hasData="false" static="false"/>
  </attribute>

  <method name="m1" language="l1" codeURL="sd">
    <attribute name="atribute1">
    <attrClassification visible="false" hasData="false" static="false"/>
    </attribute>
  </method>
</version>
</atomicComponent>
```

This component has two data ports (input and output) on its interface. One version, named ver1 is defined, containing, two extra data ports (inputX and outputX), two attributes (attribute1 and attribute2) a method (m1) and seven value definitions. Note that the type of data transmitted by a data port is represented by a predefined attribute (portType). Also, when a value makes a reference to an attribute, it employs its full name. The full name of an element is the concatenation of all its containers, up to the component element, and its name.

The following examples illustrates a topology containing three component instances (produtor, consumidor1 and consumidor2), where the output port of one component (produtor.out) is connection to an input port of both components (consumidor1.in and consumidor2.in), via two distinct connections (c1 and c2).

```
<topology name="Top">
  <compInstance instanceName="produtor" compName="Producer"
libraryURL="ProducerConsumer.xml">
  </compInstance>
  <compInstance instanceName="consumidor1" compName="Consumer"
libraryURL="ProducerConsumer.xml">
  </compInstance>
  <compInstance instanceName="consumidor2" compName="Consumer"
libraryURL="ProducerConsumer.xml">
  </compInstance>
  <attribute name="ISEM">
    <attrClassification visible="false" hasData="true" static="true"/>
  </attribute>
  <attribute name="Toplevel">
    <attrClassification visible="false" hasData="false" static="false"/>
  </attribute>
  <connection name="c1">
    <portRef completePortName="Top.produtor.out"/>
    <portRef completePortName="Top.consumidor1.in"/>
  </connection>
  <connection name="c2">
    <portRef completePortName="Top.produtor.out"/>
    <portRef completePortName="Top.consumidor2.in"/>
  </connection>
  <version name="DEsim">
    <defVer name="d1" instanceName="Top.consumidor1" versionName="DEsim"/>
    <defVer name="d2" instanceName="Top.consumidor2" versionName="DEsim"/>
    <defVer name="d3" instanceName="Top.produtor" versionName="DEsim"/>
    <value type="string" valueExpr="DE" associatedAttribute="Top.ISEM"/>
  </version>
</topology>
```

3. Rule Checking

Just like in any programming environment, the description of a component may contain flaws. For instance, a topology with component instances that are only partially defined or an attribute with several values referencing it in the same version are example of syntactical inconsistencies.

Code cannot be generated for a specification containing erroneous components. One solution would be hardwire code inside the implementation of the code generation to detect all possible error situations and report it to the designer.

However, there is a better solution that exploits the fact that the syntax of the specification is language independent, based on a graph structure. One can code every possible error situation as rules[2], using some logic formalism, and than check if the rule is satisfied by the graph structure. The two major advantages of this approach are:

1. It is very easy to add new checks. Instead of searching the source code of the framework and adding extra code to handle the potential erroneous situation, it is only necessary to code the respective rule in a text file;
2. The user of the framework can too add its rules. Those rules would impose constraints on its components, like ensuring that some value is within a certain range, or that one component has at least one connection to every data port, etc.

Therefore, one of the tools of the ODOG framework is a rule checker, that reads a rule and determines automatically if a given component satisfies it. It is a static rule checker, since the verification is done at design time. Dynamic rule checking is also possible, but is not currently supported. Only components that pass the rule checking tool can be input to the code generating tool.

3.1 Syntax and Semantics

A rule is a logic expression over the elements of the graph structure. The atom of a rule is a element with its associated attributes or a literal. Each different syntactic element have a list of available attributes. The following table list them, together with the type name of each element. All syntactic elements have the name and fullName attributes.

Element	**Type**	**Attributes**
Data Port	dport	isInput, isOutput
Exported Port	exportedPort	

2 not all error conditions are possible or efficient to write as rules.

Atomic	atomicComponent	
Attribute	attribute	
Attribute Classification	attributeClass	visible, hasData, static
Topology	topology	
Component Instance	compInstance	compName, libURL, instanceName, localInstanceName, fullInstanceName
Atomic Version	atomicVersion	
Topology Version	topologyVersion	
Value	value	type, valueExpr, isDefault
Method	method	language
Connection	connection	
Version definition	defVer	versionName, instanceName

For an atomic component, an atomicComponent node always exists and has a zero in-degree. Likewise, the topology node for a topology. The definition of each attribute of an element is:

Element Attribute	**Definition**
name	The name of the element.
fullName	The concatenation of names of all elements starting from the first component element (atomic or not) up to the container of the attribute.
visible	the state of visibility of an attribute.
hasData	the state of presence of data of an attribute.
static	the state of the accessibility of the data of an attribute.
isInput	if the associated data port is a receiving port.
isOutput	if the associated data port is a sending port.
compName	the name of a component to be instantiated.
libURL	the library containing the component to be instantiated.
instanceName	the name of the component instance
fullInstanceName	similar to fullName, but instead of the component's name as the last element, uses instanceName

localInstanceName	the concatenation of the topology's name that contains the instance, the version's name if the instance is within a version, and the instanceName. It is different from fullInstanceName, since the full name includes all elements up to the very first component.
type	a string denoting the type of the value.
valueExpr	an expression denoting how the value is obtained.
language	one of the host languages supported by the framework.
versionName	the name of the version selected.
instanceName (within a defVer)	the fullInstanceName of the compInstance being defined.

The attributes visible, hasData, static, isInput and isOutput have possible values “true” or “false”. The valueExpr attribute may have any value from the available types of the framework. All the other attributes have any string as possible values.

There are two means to access the value of an element attribute within a rule:

1) by giving the full name of the element;
2) by associating a variable with a particular element.

The first case is done by the writer of the rule. The second one is done by the tool, when encountering path quantifiers. The syntax for accessing an attribute is:

node full Name OR variable name '[' attribute Name ']'

Consider the example of component AATest2 shown in section 2.7. The sub-expression *AATest22.ver1.inputX[isInput]* is accessing the attribute isInput of the data port inputX.

Comparison expressions can be build using atoms and comparison operators. The usual operators are available: =, !=, <, <=, >, >=. Note that the last four operators make sense only for the *valueExpr* attribute of a value. Using again component AATest22, the expression *AATest22.ver1.outputX[isOuput] = “true”* is testing if the port outputX is tagged as a data sending port.

Comparison expressions can be glued by conditional operators. The common logical operators are available: && (and), || (or), => (imply), <=> (if-only-if), ~ negation.

The syntactical elements are grouped in a graph structure, where each edge between nodes represents a containment relation, that is, the source node contains the sink node. A rule contains to extra operators, named as path-operators, for exploring this relation:

-> : is a “has edge” operator;

->* is a “has path” operator.

The syntax is:

node full Name OR variable name ['->' OR '->'] node full Name OR variable name*

For *AATest22*, the expression *AATest22 -> AATest22.ver1* means that a node with full name *AATest22* contains a node with full name *AATest22.ver1*.

The explicit use of the full name of elements in expressions limits its usefulness, since for every possible specification, the rules would need to be rewritten. A better idea is to use a variable, and bound it to a specific node. Currently, there are two ways to declare and bound a node to a variable: the universal quantifier and the existential quantifier.

The syntax of the universal quantifier is:

'PT.'nodeType'.'varName'(' expression ')'

nodeType is one of the possible types of elements listed in the above table. *varName* is an identifier for the variable. For the universal quantifier expression, all the nodes of the type *nodeType* of the graph will be bound to the variable *varName*, one at a time. The universal quantifier expression is true only if for every possible assignment of the variable, the *expression* is evaluated to true.

The syntax of the existential quantifier is analogous:

'EX.'nodeType'.'varName'(' expression ')'

The only difference to the universal quantifier is that the expression is evaluated to true if it is found at least one node binding to *varName* that evaluates *expression* to true.

As an example, the expression *PT.dport.d(d[name] != “porto”)* means that no data port of an atomic component can be named as *porto*.

For examples of rules, see the *OdogRules* directory of the odogWorkspace.

3.2 Types of rules

Although any valid rule within the ODOG framework follows the syntax and semantics described above, it is useful to classify the rules according to different tasks. Currently there are the following groups of predefined rules:

- syntax;
- code generation;

All these rules are shipped under the OdogRules directory of the odogWorkspace.

4. Code Generation support

The ODOG framework maintains code generation as simple as possible. Currently, no parsing and analysis of host language code is necessary. Neither translation between host languages. For atomic components, the designer implements the behavior of the methods in any of the supported host languages. The code generating tool reads those peaces of code, performs some text transformations on them, and together with predefined parameterized templates of code, will produce the final executable for the desired platform.

A platform is defined as the target execution environment of the specification. Two attributes must be present in the toplevel[3] component of a specification: Toplevel (indicates that the component is the toplevel one) and ISEM. The ISEM attribute will specify the interaction semantics for the respective composite component, and must be present in at least the toplevel component. Other composite components may also have this attribute, identifying the desired ISEM. This mechanism allows for specifications with multiples ISEMs.

The input to the code generation is a checked composite component, together with the desired version for it. The tool generates a module for each component instance, for each composite component with a distinct ISEM, a main file and a *Makefile*.

4.1 Template code generation

The code generation mechanism relies on a template that has two types of elements:

1) character data;
2) parameters.

A character data is a set of ASCII characters. Parameters are variables with values attributed by the code generation tool. A template is described in XML.

The template code generation is used throughout the ODOG framework. For instance, the code associated with a topology governed by a certain ISEM is a template. Its *compute* and *fixpoint* methods are predefined in this template. Templates for the available code-generators can be found under the *CodeGenerators* directory of the odogWorkspace.

3 toplevel is an composite component that is not contained in any other component of the specification.

5. Graphical environment

A text editor is sufficient to create components and build a specification, since they are a set of text files. However, a GUI for editing specifications is of great utility, for two main reasons:

- it allows for host languages that are not necessarily textual;
- creation and editing is simplified by avoiding syntax details.
- a GUI provides an environment for management of components and specifications, organized in projects and libraries.

An important philosophy behind the ODOG framework is to relief the designer from burdens and complexities associated with the creation of components and specifications, that arise are at component development. For atomic components, one must understand the host language, how to employ it in a component, how to ensure that its behavior is correct and well constructed. For composite components, it may be necessary to understand the interaction semantics to a great level, in order to avoid undesirable behaviors. The ODOG framework tries to provide several tools to alleviate the designer from those technical burdens, and to concentrate on the design itself. Currently, two groups of tools are planned: component creation and specification analysis.

The first group act when the designer is developing the component. The objective of such tools are simplification of the process and/or ensuring correctness. For example, starting from a finite state machine, possibly captured by some graphical tool, an atomic component can be automatically generated. The second group acts after the specification is code generated, and returns useful information to the designer. A typical example is a debugger.

The GUI environment should be the hook for all those tools.

5.1 Meta-artifacts and artifacts

A component or a rule is named an artifact within the GUI environment. The property of an artifact is that is can be input to the code generation, that is, an artifact is ready for use. Although static rules are not code generated, they must be checked before code generation. Also, dynamic rules should be inserted in the generated code. An artifact provides the hook for the actual description of the component, plus extra information, like its version or a textual message used for commentaries or reports.

A project is a collection of artifacts forming a specification. A library is also a collection of artifacts, but not necessarily forming a specification, that is, it should not be input to code generation.

An artifact is a final product, meaning that once it is finished, it is ready for producing the desired code. As mentioned in the previous item, component creation tools are an important objective of the ODOG framework. It is possible that some of those tools do not work with completed artifacts, but with artifacts with only some abstract information of the component or rule. After some generation process, the real artifact is produced. Therefore, a meta-artifact is an object that after some transformation, will produce an artifact. The FSM example of the previous section is a meta-artifact. Meta-artifacts can be placed within libraries and projects, but for code generation, at least one artifact must have been generated.

5.2 User interface

Please refer to the *Getting Started* document available at http://odog.sourceforge.net for how to use the GUI environment and its capabilities.

6. Component's host language

Depending on the host language and the type of the component artifact, different techniques should be used in order to allow the component to interact with its surrounding.

General purpose languages usually don't directly fit the data exchange model of components. Hence, a technique based on abstract methods is employed for those. These methods are akin to procedures, and their code is filled during code generation with code for data transmission.

For domain specific languages, it is made a mapping between the appropriate syntactic elements of the language and the component. For example, Esterel[4] is a language that is based on processing of input events and production of output events. Those events flow through ports. Therefore, if Esterel is used as a host language, the framework should map the Esterel's ports with the component's ports.

Currently the ODOG framework supports the ANSI-C as a host language.

6.1 Abstract methods for general purpose languages

The data communication methods, their meaning and prototypes are:

*void send(char *connection, void *data, size_t length)*

Used to send data through one connection associated with an output port. The contents of the *data* pointer, whose length is specified by the *length* parameter, will be copied to all destination ports associated with *connection*. The contents of *connection* must be the name of a connection associated with an output port of this component. In order to facilitate the use of the data communication methods, two auxiliary methods are provided:

*int numberOfConnections(char *port)*

*char *nameOfConnection(char *port, int number)*

The *numberOfConnections* method returns the number of connections associated with the port with name specified by the *port* parameter.

The *nameOfConnection* method return the name of the connection associated with *port* and whose number is given by the number parameter.

*void sendDelayed(char *connection, void *data, size_t length, double instant)*

Similar to the *send* method. The *instant* parameter indicates a future moment in time

4 www.esterel-technologies.com

where the data will be available to the destination. Therefore, in order to be different than the *send* method, the component using the *sendDelayed* method must be governed by an Isem that has a notion of time evolution.

*void sendAll(char *port, void *data, size_t length)*
*void sendAllDelayed(char *port, void *data, size_t length, double instant)*

Just as the *send* and *sendDelayed* methods respectively, but delivers the data to all connections associated with the output port *port*.

*void receive(char *connection, void **data, size_t *length)*

Receives data from the given connection associated with an input port. The *data* vector is allocated by the framework, so after processing, a *free(data)* should be called.

char canReceive("connection", int size)

Returns a value equals to zero if *connection* can supply *size* data elements. The *connection* should be associated with an input port. Otherwise returns a value greater than zero.

char canReceiveAll("port", int size)

Similar to the *canReceive* method, but checks all the connections associated with *port*.

char canSend("connection", int size)

Returns a value equals to zero if *size* data elements can be send through *connection*. Otherwise returns a value greater than zero.

char canSendAll("port", int size)

Similar to the *canSend* method, but checks all the connections associated with *port*.

6.2 Types

The data port and value elements of a component have a field to specify its type. Those types should be given in the host language that the component will be generated, since this field will be propagated during code generation. The framework checks if every port in every connection has the same type.

6.3 Examples

The following code illustrates the use of some of the abstract methods of section 6.1.

```
void init() {
    scheduleMe(1.0);
}

static char msg[45];

void compute() {

    sprintf(msg, "tempo = %f", currentTime());
    sendAllDelayed("out", msg, sizeof(msg) + 1, currentTime() + 0.35);

    scheduleMe(currentTime() + 0.4);
}

#include <stdio.h>
void finish() {
    printf("Produtor encerrou\n");
}
```

This component has one output port named *out*. At the initialization of the specification, the component requests to be executed once at instant of time 1.0. At each execution of the component, a string message is sent into the future (0.35 from the moment of each execution) to all connections associated with port “out”. The component requests to be executed at each moment of time, multiple of 0.4. When the execution terminates, if this happens, the component prints a text message.

7. Interaction semantics

7.1 Discrete events (DE)

The *Discrete Events*[5] interaction semantics associates to each connection between data ports the communication of **events** between the source and destination components. An event is a pair of a data value and a real number, called the *timestamp*. The purpose of the *timestamp* is to establish a total order on all the events generated during execution. In other words, the *timestamp* represents a time instant.

An atomic component under the DE semantics has its *compute* method called at any time instant where events are present on at least one of its input ports. The component may or may not produce new events, or alter its internal state, as a consequence of the execution of its *compute* method. At every occasion that a new event is produced, its *timestamp* must be equal or greater than the *timestamp* of all events that triggered that execution of the compute method. If an event with a *timestamp* smaller than the current instant of time is generated, a fatal error is reported and execution is aborted.

Conceptually, a new event is not sent directly to the ports of the components connected to the output port that generated the event. It is rather placed on a queue, where the events are ordered based on their timestamps. The same queue is used for all components. Hence, the time evelution in the DE semantics is global. The selection of which component should have its *compute* method executed next is based on the destination of the first event on the queue. If there are more than one event for this component at the head of the queue, all are removed and placed on the respective input ports. If after the *compute* method of the selected component there are still events on its input ports, they will be discarded.

A potential conflict situation that may arise is the presence of events to different components with the same *timestamp* at the head of the queue. In this case, the decision of which component should be executed is done based on a priority value. Each component has a distinct priority value. This value is computed from the topological order of the graph of the topology where the components are instantiated. The nodes of this graph are the atomic (or composite components with different ISems) components, and edges are connection between data ports. In order to be able to compute the topological order, the graph obtained from the topology must be acyclic. Note that it is still possible to have cyclic connections on topologies, if it is guaranteed that the events produced along every possible cycle will always be into the future. This information can only be provided by the designer. In order to do so, an attribute named *delayed* (invisible, without data, static) must be associated with the respective output port. It is not associated with a connection, because when a component is created, it is not known where it will be used, that is, how many connections there will be on a data port.

The execution of a topology governed by DE is controlled by a scheduler that finds at

5 J. Banks, et al, **Discrete event System Simulation**, ISBN 0131446797, 1996.

every instant of time, which is the next component to execute. The execution of the specification stops when the global event queue is empty, or when a particular instant of time is reached.

The compute method of an atomic component under the DE ISem should have the following pseudo-code:

1. For each input port or connection, check the presence of events with the *canReceive* method. It is guaranteed that at least one will have new data;
2. Read all the events available with the *receive* method;
3. Do some processing based on the new events, and/or the component state;
4. If desirable, produce new events with the *send* method.

The semantics of the DE ISem is deterministic. Note that zero delay cycles are not allowed, that is, connections forming cycles that produce events with the same *timestamp*. Unfortunately, there is no adequate way to prevent a designer for introducing such cycles. Therefore, it is possible that flawed specifications when executed, don't produce outputs since they keep looping on the same time instant.

In addition to the methods described in section 6, the DE semantics provide specific methods to the atomic component:

scheculeMe(double instant) : requests the execution of the atomic component at *instant*. If multiple calls to this service are made prior to reaching the least value of instant specified by them, only the last one will be efective.

abortSchedule() : invalidates the previous call to the *scheduleMe()*.

currentTime() : returns the current value of time.

7.2 Dataflow (DF)

This semantics breaks the execution of a component in a series of **firings**. A component is said to be able to fire when at least on data value is available in every input port. In order to avoid loosing data, each connection is associated with a (conceptually infinite) queue. At any moment, more than one component may be able to fire. The scheduler implementing the semantics chooses with component to fire. The dataflow semantics has been extensively studied in the past[6].

In ODOG, a dynamic version of the dataflow semantics is adopted:

- the component should inform the sample rate of each input port;

6 Please, refer to W. M. Johnston, J.R.P. Hanna and R. J. Miller, **Advances in Dataflow Programming Languages**, ACM Computing Surveys, Vol 36, N 1, 2004.

- at each firing, the component may change the sample rate of any port.

With this two conditions, any computable function can be captured using DF. However, it is impossible to determine if the execution will be bounded, that is, if an upper bound on the capacity of the data queues is finite, and deadlock-free. Deadlock is a condition that, although there are non-empty queues, no component is able to fire with the available data.

As in the DE semantics, cycles in the topology represent a problem. It is easy to see that a cyclic condition avoids the firing of the components in the cycle, hence, deadlock. The solution is during initialization, generate data somewhere in the cycle, so that when execution starts, some components may fire. The *delayed* attribute is used to mark the output ports that will generate data during the initialization phase.

Under the DF semantics, the designer has the following semantic specific methods:

void *setSampleRate(char *port, short value)*: sets the sample rate of *port*.

int *getSampleRate(char *port)* : returns the sample rate of *port.*

void stopExecution(void) : asks for immediate termination of the execution. The finish methods of all components will be called before exiting.

The typical *init* method of an atomic component should set the initial values of each sample and if required, generate initial data. The typical *compute* method should read the required data, based on the current set of sample rates, perform some computation, generate output, and if necessary, change some input sample rates. Note that a sample rate equals to zero, means that the respective input will not be considered in the next firing.

Bellow is an example of the flexibility of the dynamic version, by showing how to implement a selection component: based on the value of a control input, read data from the respective data input, and send it to the output.

```
void compute() {
   size_t len;
   char *ctrl;
   double *data;

   switch(state) {

       case WAIT_CONTROL: {
           receive(nameOfConnection("control", 0), &ctrl, &len);
           if(*ctrl == 0) {
               setSampleRate("input1", 1);
               state = WAIT_INPUT1;
           }
           else {
               setSampleRate("input2", 1);
               state = WAIT_INPUT2;
```

```
            }
            setSampleRate("control", 0);
            free(ctrl);
        } break;

         case WAIT_INPUT1: {
             receive(nameOfConnection("input1", 0), &data, &len);
             sendAll("output", data, len);
             free(data);

             setSampleRate("input1", 0);
             setSampleRate("control", 1);
             state = WAIT_CONTROL;
         } break;

         case WAIT_INPUT2: {
             receive(nameOfConnection("input2", 0), &data, &len);
             sendAll("output", data, len);
             free(data);

             setSampleRate("input2", 0);
             setSampleRate("control", 1);
             state = WAIT_CONTROL;
         } break;
    }

}
```

7.3 Synchronous (SR)

7.1 Semantics

The SR semantics draw its name from the *synchronous hypothesis*: for any set of input events, the system reacts in zero time. This hypothesis has been used extensively for hardware design, and also for embedded software. Its major advantage is to provide a deterministic concurrent model. The version implemented in the ODOG framework is the one described in Edwards[7].

The SR semantics provides the notion of an **instant**: every set of new events that the system must react to constitutes an instant. At each instant, the *compute* method of the components are executed in such a way that the outputs of the system are generated. Therefore, it is expected that the system converge to this output set at every possible instant. It is imperative then to established necessary and sufficient conditions to obtain this convergence in finite time. One definition and one condition on the atomic components are required for convergence:

Definition: at most one data value can be associated[8] to each connection. Thus, a connection

7 S. A. Edwards, **The Specification and Execution of Heterogeneous Synchronous Reactive Systems,** *PhD Thesis, University of California, Berkeley, 1997.*

8 this can actually be relaxed to multiple data items, but not currently implemented in ODOG.

can be in one of three states: *unknown, known and absent, known and present*. The first state means that whether the connection has or has not data is not known. With the exception of the system's inputs, every connection is at the unknown state at the beginning of an instant. The second state means that it is known that in the particular instant, no data will be generated in the respective connection. The third state indicates that data was generated in the current instant.

Condition: every atomic component must implement a monotonic function. A monotonic function is one that given a set of “more defined” inputs, will generate “more defined” outputs. By more defined I mean that the difference between two sets is that the more defined can only change some previously unknown inputs to known, and no known input may change back to unknown. As a consequence, in SR, whenever an output is set to be known, it cannot be changed back to unknown.

As in the DF and DE ISems, cycles of components represents a problematic situation, since deadlock may arise. The solution is to classify atomic components in two categories: Strict and Non-Strict. The first type require that all of its input connections be in a known state prior to allowing its *compute* method to be called. The second type can be called several times during an instant, independent of the state of its input connection.

The scheduler of the SR ISem constructs a graph based on the components and their connections, and whenever a Non-Strict component is found, no edges are placed going to this component. In order to be able to execute the specification, the resulting graph cannot have any cycles. The order of calling the *compute* methods is then a topological order of this graph. At the end of each iteration of this order, if any new event was generated, a new iteration is run. The instant terminates when during an iteration, no new event is produced. If all components are implemented using monotonic functions, it is guaranteed that the number of iterations is finite.

7.2 Component development under SR

The *receive* method under SR has a variation when the data transmitted is an scalar value. In this case, the semantics described in 6.1 is changed:

*void receive(char *con, void *data, size_t *len)* : the data parameter is no longer a pointer to a vector of void. It is a pointer to the scalar type being communicated.

Therefore, pay attention to the type of the input connection when using the *receive* method.

Since the connections under SR may have three instead of two states, the designer has the following specific methods for setting/testing the *known and absent* state:

*void setAbsent(char *con)* : set the state of the connection specified by *con* as being known and absent in the particular instant.

*void setAbsentAll(char *port)* : set the state of all connections associated with the input port *port* as being known and absent in the particular instant.

*char isAbsent(char *con)* : returns 0 if the state of connection *con* is known and absent. Otherwise returns 1.

*char isAbsentAll(char *port)* : returns 0 if the state of all connections associated with input port *port* is known and absent. Otherwise returns 1.

The *canReceive* and *canReceiveAll* methods will return 1 only when the state of an input connection is different than *unknown*. The *canSend* and *canSendAll* methods will return 1 only when the state of an output connection is *unknown*.

Strict components are monotonic by definition. However, it is possible to implement non-monotonic functions with the Non-Strict components. Bellow is a fragment of code exemplifying this situation:

```
...
if(isKnown(“preferred”, 0)) {
    if(canReceive(“preferred”, 0)) {
        sendAll(“output” ...);
    }
    else
    if(isKnown(“alternate”, 0)) {
        if(canReceive(“alternate”, 0)) {
            sendAll(“output” ...);
        }
        else {
            sendAll(“output” ...);
        }
    }
}
...
```

In the above code, there are two input ports: *preferred* and *alternate*. Whenever the state of *preferred* is known, if it has data, some action will be taken. If it is absent, then the *alternate* input is verified for the presence of data. In order words, the *preferred's* state must always be checked first, and has precedence over the *alternate's* state. The above code is monotonic. What could not be done is to check both preferred and alternate at the same time, since both would change the state of output. Therefore, one must **never** write a component based on the order of arrival of events.

Non-Strict are useful not only to avoid deadlock, they can speed-up the convergence of a system. Think of logic gates: whenever a controlling value is available at an gate's input, output can be produced irrespective of the other inputs. In order to tag a component as Non-Strict, it is only necessary to add an attribute named *nonstrict* (invisible, static, without data) to it.

The method *fixpoint* is allowed under the SR ISem. Its purpose is to allow the designer to update the state of a component. Since the number of times of each component's compute method is called is unkown, it is only safe to update any state when convergence is found, that is, when the instant is finished. Therefore, *fixpoint* is used for this purpose.

8. Platforms

A platform is a set of software libraries, hardware information and models, that can execute code. Therefore, the target of any code generation is a platform. ODOG currently have platforms:

- host : generates code for running under a Linux/GNU machine;
- multicore : generates code for a multicore machine running LINUX/GNU;

Depending on each platform, an ISem and/or host language may be available or not. Also, the actual code generated for the same specification can differ greatly depending on the target platform.

8.1 Host

This platform is mainly used for testing purposes. Together with code for all atomic and composite components instances, a Makefile and a main.c file are generated. The Makefile includes an application dependent file (*AppMakefile*) for referring to external resources. The main file is used to start the execution. Depending on the ISem, it has different arguments:

- DE : *startime* and *stoptime* : for specifying the time interval of execution;
- SR : *numberOfIterations* : gives the number of maximum instants to be processed;
- DF : none, runs forever.

8.2 Multicore

This platform can generate code only for the DF ISem. The code generator produces code using the posix threads API, therefore automatically using all processor cores from a multicore chip. No input from the designer is necessary.