

Towards a Formal Semantics of a Composition Language

Nigel Jefferson, Steve Riddle
School of Computing Science,
University of Newcastle upon Tyne, UK
{n.p.jefferson, steve.riddle@ncl.ac.uk}

Abstract

Although several composition environments exist that are built on top of object-oriented languages, they fail to supply the level of abstraction required to specify compositions of components. There is therefore a need for pure component-based languages in order to allow the composition developer to focus on the composition from a clear viewpoint, free of any obscurities imposed by existing programming languages that essentially operate at the individual component level. In this paper we make a clear distinction between a composition language and a composition representation. A composition language is any language that allows the specification of a piece of software in terms of its composition whereas a composition representation is the abstract, general, architectural description of a composition. This position paper sets out to formally express the basis for a composition representation. The definition of an abstract representation is necessary in order to derive the formal semantics of a composition language. We believe that this semantic definition should be the initial step in the construction of a high level component-based language.

1 Introduction

In much the same way as the need for customizable reuse of software fueled the growth and development of object-oriented programming languages over module-based languages, the same driving force for component-based solutions is leading to object-oriented languages being transcended by component-based composition languages.

Although several composition environments exist that are built on top of object-oriented languages, they fail to supply the level of abstraction required to specify compositions of components. Most focus mainly on special application domains and do not enforce a clear separation of components [8]. A recognised problem with reuse in object-oriented programming is that in order to take advantage of customizable software reuse using inheritance it is often necessary to know details about the internal workings of the methods that are being extended [15]. This implies that the object-oriented paradigm focuses on ‘white box reuse’ and is therefore unsuitable for the reuse of ‘black box’ software components.

There is therefore a growing need for pure component-based languages in order to allow the composition developer to focus on the composition from a clear viewpoint, free of any obscurities imposed by existing programming languages that essentially operate at the individual component level. To be pure, a component-based language must provide sufficient abstraction and rigour, and so must have a precise vocabulary and semantics.

1.1 Definitions

Component-based applications are in general described in terms of a *composition* of components. In this paper we make a clear distinction between a composition *language* and a composition *representation*.

- A composition language is any language that allows the specification of a piece of software in terms of its composition.
- A composition representation is the abstract, general, architectural description of a composition.

A composition representation can be viewed as the ‘internal’ representation of a composition that is created through the execution of a composition language program. Furthermore, it is conceivable that a composition language would allow run-time modification of the internal composition representation. Therefore we argue the need for a clear distinction between the two, as each should possess its own abstract syntax and semantics. The semantics of a composition language program would define transitions between different composition representations and the semantics of a composition representation would describe the behaviour of an actual application. Getting the internal representation correct is therefore a vital step in determining the vocabulary and semantics of the associated composition language.

There have been many definitions of what constitutes a component. We follow the following definition of a component provided by Szyperski [14]:

“A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

To express the internal representation of a composition, we adopt trends utilized in the design of ADLs (Architectural Description Languages) and follow the ‘pipe-and-filter’ architectural style whereby each component is described in terms of sets of inputs and outputs. In this style a component essentially ‘filters’ data from one pipe (connected to an input) into another pipe (connected to an output) [13].

1.2 Formal Semantics

This position paper sets out to formally express the basis for a composition representation. The definition of an abstract representation is necessary in order to derive the formal semantics of a composition language. We believe that this semantic definition should be the initial step in the construction of a high level component-based language.

A precise description of a programming language, such as that afforded by a formal definition, should be a prerequisite for its implementation and use. A formal language description can be used to formulate laws of equivalence and provides a high level of rigour when designing programs. This is of particular importance when dealing with languages and programs that are involved in the creation of safety critical applications [2].

Conventionally a formal language definition is specified in terms of its abstract syntax and semantics. The role of the abstract syntax is to reduce the syntax of the language constructs and specify them in pure terms by abstracting away from irrelevant notational details. The semantics are in turn divided into two distinct sections: static semantics and dynamic semantics. The static semantics (or context conditions) cover the *elaboration* of a program and determines whether it is ‘well formed’ and correctly typed. The dynamic semantics relate to the actual *execution* of a program and determines its meaning in terms of its declarations and value information in the state.

Two main methods for expressing language semantics are in common use: denotational and operational. In denotational semantics, a meaning function is defined which maps the elements of the programming

language to some understood set of denotations. Operational semantics are defined via state transitions which represent a hypothetical machine that interprets programs written in the programming language. Lucas provides a more complete description of the different methods [6]. Formal semantics can be found for many programming languages [10, 5].

1.3 Related Work

Much research has been done into ADLs and several such languages exist, for example Darwin [11] and Rapide [7]. Formal semantics have also been specified for component behaviour in architectural descriptions [13].

Recent research has resulted in the definition a number of composition languages, some of which (for example PECOS [1] and CL [3]) specify semantics formally. Of these, CL has most influenced the content of this paper, though we chose to express our language using a state-based modeling technique rather than the CSP process algebra used to define the behaviour of CL.

1.4 Roadmap

The remainder of this paper is structured as follows. First we describe a straightforward, abstract composition representation; from this starting point a number of refinements are made and additional properties specified. Upon the completion of a refined representation, we provide an example static semantic rule - a complete semantic definition is beyond the scope of this paper. The static semantic definition is an important step as if a composition can be shown to be well formed, then it would therefore be possible to show (through the use of additional context conditions) that a well formed composition language program will produce a well formed composition.

The semantics of a particular composition are of course significant, and so it is important to be able to show the dynamic semantics of a particular composition created as a result of a well formed composition language program. The specification of such semantics is included as an item of further work.

2 Basic Composition Representation

In this section we specify an abstract representation usable for the architectural description of component assemblies. The structure and alphabet presented in this section is intentionally basic in order to clearly express the required constructs. The representation is specified in VDM-SL (Vienna Development Method Specification Language) [4], a formal modeling language that uses well-understood mathematical entities such as sets, mappings and functions, as well as basic types such as integer and boolean.

2.1 Ports

A component provides interfaces to the surrounding assembly and environment, these are modeled in terms of *ports*. Each port acts either as a ‘source’ or ‘sink’ for data; a sink port represents an interface to the component and a source port represents some type of response¹ from the component.

$$\begin{aligned} \text{Port} &:: tp : \text{SOURCE} \mid \text{SINK} \\ &\quad in : \text{DataInterface} \end{aligned}$$

Ports are defined purely in terms of their type (source / sink) and data interfaces². A port’s data interface is specified as a sequence of parameter types:

¹The term ‘response’ implies that a source port must be linked somehow to a sink port and that the source port merely supplies the data resulting from a response to the communication with the sink port. This is only one possible scenario; others are discussed later.

²Other constraints may include the use of contracts; these are considered later.

$DataInterface = DataType^*$

$DataType = Data\text{-}set$

In the definition above, the definition of type *Data* is not significant and is therefore represented as a token³ type. *DataTypes* are simply represented as a set of *Data* in order to maintain an appropriate level of abstraction. Therefore for a data value to belong to a particular type, it must be a member of the set defined by the corresponding *DataType*. An accurate list and explanation of potential types would depend on the environment in which the component operates.

2.2 Components

The main elements of a composition language should be components:

$Component :: \begin{array}{l} ports : PortMap \\ behav : Behaviour \end{array}$

$PortMap = PortId \xrightarrow{m} Port$

This abstract record type represents a component as a collection of ports and their associated behaviour. These ports can take many forms depending on the nature of the component and the surrounding environment. For example, if the component in question is a pre-compiled and dynamically loadable library of functions, the sink ports might represent the input parameters to those functions and the source ports might represent the values returned from those functions.

The collection of ports is represented in this case by a mapping of *PortIds* to *Ports*. Using such a mapping it is therefore possible to access a particular port by referring to its id; this method is used in the definition of the component's behaviour. The definition of *PortId* type is not important and it is therefore represented as a token type.

A component's behaviour is defined in terms of 'computations'. A computation links one or more sink ports to one or more source ports and represents an internal component computation:

$Behaviour = Computation\text{-}set$

$Computation :: \begin{array}{l} sinks : PortId\text{-}set \\ sources : PortId\text{-}set \end{array}$

This merely represents the static behaviour of a component in terms of responses to requests. This is a straightforward depiction which ignores (for the sake of simplicity) the possibility that a sink port need not be connected to a source port and visa versa; this prospect is explored in Section 3.7.

It should also be noted that no attempt is made at this stage to characterize the actual functionality provided by a component through the use of a particular port or set of ports; one method of specifying such functionality is through the use of contracts, which is discussed in Section 3.4.

2.3 Connectors

Connectors are used to provide a means of modeling communication channels between the components:

$Connector :: \begin{array}{l} c_1 : ComponentId \\ p_1 : PortId \\ c_2 : ComponentId \\ p_2 : PortId \end{array}$

³A token type consists of an infinite set of distinct values, called tokens; they are used where a type definition is not important. Tokens cannot be used for anything other than equality and inequality comparisons.

This representation simply models a connector as an association between two different component ports (port p_1 on component c_1 and port p_2 on component c_2)⁴. The *ComponentId* type is used in the same way as *PortId* (see Section 2.1) and is properly introduced in Section 2.4.

2.4 Applications

So far we have only considered the representation of components and connectors; these are meaningless without a top level specification of the application created as a result of the composition:

Application :: *Assembly*

Where an assembly is described as follows (*ComponentId* is defined as a token type):

$$\begin{aligned} \textit{Assembly} &:: \quad c : \textit{ComponentId} \xrightarrow{m} \textit{Component} \\ &\quad \textit{top} : \textit{Connector-set} \end{aligned}$$

An assembly denotes a particular collection of components c composed as described by the topology of connectors specified in *top*.

3 Refined Composition Representation

The language introduced in Section 2 has many limitations, for example it only allows a flat composition and doesn't provide for the hierarchical composition that might be expected from a component based system. This section takes the basic representation and introduces many new concepts (such as hierarchical composition). This representation also seeks to 'flatten' and reduce the specification in order to produce a concise representation that can easily be checked by context conditions and updated by composition language statements.

3.1 Hierarchical Composition

We model hierarchical composition by allowing an assembly to be composed of components and other assemblies. This can be easily accomplished by redefining *Component* as follows:

$$\textit{Component} = \textit{ComponentDesc} \mid \textit{AssemblyId}$$

Where *ComponentDesc* indicates a component description (see Section 3.7) and *AssemblyId* is a token type referring to another assembly (properly introduced in Section 3.8).

3.2 Bridges

With the addition of hierarchical composition, it is necessary to include constructs to represent the boundaries between assemblies; these are represented by *bridges*. A bridge is conceptually the same as a port in that it acts as either a source or sink and has a particular data interface. Semantically however they are quite different; a bridge must translate data from one environment such that it is legal in the next. The semantics of bridges are beyond the scope of this paper however and as such they are treated in the same fashion as other ports, nevertheless it is important that the reader be aware of this distinction. The representation of bridges is covered in Section 3.5.

⁴It is possible to use an invariant to model the requirement that a connector should only connect a source port to a sink port, we choose however to model such constraints using context conditions (see Section 4).

3.3 Environments

An environment type defines the services that are available to components, the low level semantics of connectors and the representations of available data types. A component can only be composed into an assembly associated with a particular environment type or one of a set of environment types as it will most likely make use of specific environment services and data types. An environment type could for example relate to a specific platform with a particular operating system or hardware architecture and as such a particular component may have been compiled to run on such a platform.

Therefore it is possible to envisage an environment represented by the following record type:

$$\begin{aligned} Env &:: \text{alph} : \text{Data-set} \\ &\quad \text{res} : \text{Resource-set} \end{aligned}$$

Where *alph* represents the alphabet allowed by the environment in terms of a set of legal Data and *res* represents the resources made available by the environment. A resource can be viewed as a port providing an interface with the surrounding assembly environment; the representation of resources is covered in section 3.5.

3.4 Assertions

As mentioned above, at this level of abstraction it is not strictly possible to completely describe the functionality of a component, however, we can augment the information we already know (the data interface of each port) with contractual and functional meta-data. The use of contracts as design-time and run-time verification tools is well documented [9]; here we can use contracts by specifying assertions in the form of pre and post conditional predicates over the data passed to and from the interface of a particular source or sink port.

$$\text{Assertion} = \text{Data}^* \rightarrow \mathbb{B}$$

An *Assertion* for a particular port is simply modeled as a predicate over the data interface of that port. An assertion associated with a sink port is a pre-condition on the input data which must be satisfied before data can be passed to that port. An assertion associated with a source port is a post-condition that states that the port will only produce data that satisfies the assertion.

3.5 Ports

With the addition of contractual information it is necessary to redefine the representation of Port. Furthermore we can take this opportunity to incorporate new information into the definition:

$$\begin{aligned} Port &:: \text{di} : \text{DataInterface} \\ &\quad \text{ast} : \text{Assertion} \\ &\quad \text{use} : \text{MANDATORY} \mid \text{OPTIONAL} \\ &\quad \text{pl} : \text{BRIDGE} \mid \text{RESOURCE} \mid \text{ComponentId} \\ &\quad \text{tp} : \text{SOURCE} \mid \text{SINK} \\ &\quad \text{mx} : \mathbb{B} \\ &\quad \text{cm} : \text{ASYN} \mid \text{SYN} \end{aligned}$$

Where *DataInterface* is defined in Section 2.1. The *use* field provides information on its use (whether it is mandatory or optional), the value of *mx* determines whether it is mutexed, and its communication method (*cm*) states whether it uses asynchronous or synchronous communication.

As resources and bridges can (conceptually at least) be viewed as ports it makes sense to combine all three constructs into the same type. At the same time we can remove the ports from the component description and specify them at the assembly level; this allows for the trivial addition of resources and bridges and simplifies the definition of the context conditions. The port's placement (*pl*) states whether it is a bridge, a resource, or in the case of a component port it specifies the affiliated component.

3.6 Entry Points

Thus far we have covered the architectural description of the composition but have not considered its functionality, i.e. what it actually *does*. In order to maintain the abstract representation we have achieved so far, it is not strictly possible to express the semantics of a component (and therefore an assembly), instead we will focus on what we will refer to here as *entry points*. An entry point signifies a point within the composition at which execution can begin. The nature of the entry point (i.e. what instigates the execution) is not relevant; it may be time or event triggered, but for now all that matters is that it exists at all.

At first glance a resource could be construed as serving the same purpose as an entry point, but an entry point differs from a resource in one very important way: a resource (like a port) corresponds to a real software interface of some kind; and an entry point corresponds to some action or set of criteria that *triggers* some computation within the composition. An entry point can therefore be associated with any source port contained within a composition.

The following definition of an entry point merely refers to the set of ports p to be triggered in assembly a :

$$\begin{aligned} \text{EntryPoint} &:: a : \text{AssemblyId} \\ & \quad p : \text{PortId-set} \end{aligned}$$

3.7 Component Descriptions

The new port definition means that a component description need only be specified in terms of its behaviour:

$$\begin{aligned} \text{ComponentDesc} &:: \text{Computation-set} \\ \text{Computation} &:: en : \text{PortId} \mid \text{ENTRYPOINT} \\ & \quad ex : [\text{PortId-set}] \\ & \quad th : \mathbb{B} \end{aligned}$$

This definition also allows a computation to be defined that does not necessarily have to link a sink port to a source port. Now a computation can be initiated as a consequence of an entry point, or it may result in NULL which indicates that the computation will only result in an internal state change.

3.8 Assemblies

The definition of an assembly need only bring together the information previously presented on ports, components and environments. The definition of a connector is simplified significantly as a result of the new port description; a set of connectors is now represented by a mapping from source ports to sink ports:

$$\begin{aligned} \text{Assembly} &:: cm : \text{ComponentMap} \\ & \quad tm : \text{ConnectorMap} \\ & \quad pm : \text{PortMap} \\ & \quad al : \text{Data-set} \\ \text{ComponentMap} &= \text{ComponentId} \xrightarrow{m} \text{Component} \\ \text{ConnectorMap} &= \text{PortId} \xrightarrow{m} \text{PortId} \\ \text{PortMap} &= \text{PortId} \xrightarrow{m} \text{Port} \end{aligned}$$

3.9 Applications

The top level definition of an application holds all the assembly information, the set of application entry points, and finally specifies the root assembly (rt) which operates at the highest level of the application.

$$\begin{aligned} \text{Application} &:: \text{rt} : \text{AssemblyId} \\ &\quad \text{am} : \text{AssemblyMap} \\ &\quad \text{tr} : \text{EntryPoint-set} \end{aligned}$$

$$\text{AssemblyMap} = \text{AssemblyId} \xrightarrow{m} \text{Assembly}$$

4 Static Semantics

The internal representation described in Section 3 by itself would afford a lot of flexibility to a composer wishing to make use of it; the application developer could design a meaningless composition, or worse still an impossible composition that could never be realized in reality. The role of context conditions is to state what qualifies as a ‘well formed’ composition, in essence it is a hierarchical set of predicates that constitute what a well formed Application is from the top down.

Context conditions can be viewed as the set of guidelines a compiler might follow when parsing a program before translating the code into binary form. Therefore they not only state what constitutes a meaningful composition (for example ensuring that a contractual pre-condition can be passed), but also ensure consistency (for example that a connector’s data interface is compatible with its connectives).

Because a full complement of context conditions is beyond the scope of this paper, here we present an example in order to clarify the idea. The example is expressed using Plotkin’s rule notation [12], where the conclusion under the line can be inferred given that all the hypotheses above the line are true.

4.1 Auxiliary Types

In order to execute the well formedness predicates it is necessary to make use of the auxiliary type *AppInfo* that stores the useful top level application information. The well-formedness checks are constructed as a set of recursive hierarchical rules that check through the composition from the top down so *AppInfo* is used to pass that information down the rules.

$$\begin{aligned} \text{AppInfo} &:: \text{am} : \text{AssemblyMap} \\ &\quad \text{tr} : \text{EntryPoint-set} \end{aligned}$$

4.2 Context Condition Example

For our example we present the top level rule for what constitutes a well formed application:

$$\frac{\begin{aligned} \text{appinfo} &= \text{mk-AppInfo}(\text{am}, \text{tr}) \\ \text{rt} &\in \mathbf{dom} \text{ am} \\ \forall t \in \text{tr} \cdot \text{wf-EntryPoint}(t, \text{am}) \\ \forall id \in \mathbf{dom} \text{ am} \cdot \text{wf-Assembly}(\text{am}(id), \text{appinfo}) \\ &\quad \text{wf-Hierarchy}(\dots) \end{aligned}}{\text{wf-Application} \quad \text{wf-Application}(\text{mk-Application}(\text{rt}, \text{am}, \text{tr}))}$$

This rule states that an application is well formed if the root assembly exists, all assemblies and entry points are well formed, and the hierarchy is well formed (there are no cycles in the assembly hierarchy for example).

5 Conclusions

This paper has presented a basis for the definition of an abstract composition representation, as a first step in defining the semantics of a component-based language. This work is being carried out in the context of the DOTS (Diversity with Off-The-Shelf components) project, which investigates methods of selecting and composing OTS components in order to take advantage of potential design diversity between components with similar functionality. The resultant focus on use of pre-existing black-box components requires a high-level language for component composition.

We have used VDM-SL to specify the abstract representation, and provided a static semantic rule as an example of the types of rule necessary to show that a composition is well-formed.

5.1 Future Work

This paper establishes an abstract representation of a component composition, the next stage is a specification of the full static and dynamic semantics. Once this is accomplished the formal specification of a composition language can be defined. It is our intention that the composition representation act as the composition language state that is updated by the execution of composition language statements.

The creation of such a language allows for research into many interesting topics such as new methods of black-box component reuse, exception handling, arrays of components, and the semantics of bridges and connectors.

References

- [1] Thomas Genssler et al. *PECOS in a Nutshell*, September 2002.
- [2] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems*. Cambridge, 1998.
- [3] James Ivers, Nishant Sinha, and Kurt Wallnau. A basis for composition language cl. Technical Report CMU/SEI-2002-TN-026, Software Engineering Institute, Carnegie Mellon University, September 2002.
- [4] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.
- [5] Gerwin Klein, Tobias Nipkow, David von Oheimb, Leonor Prensa Nieto, Norbert Schirmer, and Martin Strecker. *Jave source and bytecode formalisations in Isabelle*. Bali, 2002.
- [6] Peter Lucas. Main approaches to formal specifications. In Dines Bjørner and Cliff B. Jones, editors, *Formal Specification & Software Development*, chapter 1. Prentice-Hall, 1982.
- [7] David C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Methods Workshop IV*, July 1996.
- [8] Markus Lumpe and Jean-Guy Schneider. Wcl 2001 workshop summary. In *1st Workshop on Composition Languages*, September 2001.
- [9] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.
- [10] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [11] Department of Computing. *The Darwin Language*. Imperial College of Science, Technology and Medicine, 3d edition, September 1997.
- [12] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

- [13] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1st edition, 1996.
- [14] Clemens Szyperski. *Component Software, Beyond Object Oriented Programming*. Addison-Wesley, 1998.
- [15] Roel Wuyts and Stéphane Ducasse. Composition languages for black-box components. <http://www.iam.unibe.ch/scg/Archive/Papers/Wuyt01c.pdf>, February 2002.