PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 42, NO. 1, PP. 25-32 (1998)

LOGICAL COMBINATORS FOR SYSTEM CONFIGURATION

Gillian HILL

Department of Computer Science, City University
and
Department of Computing, Imperial College,
of Science, Technology and Medicine
London, Great Britain

Received: September 30, 1997

Abstract

System configuration describes the construction of complex engineering systems from their component parts. The configuration language is at a meta-level to a specification language and expresses the horizontal structuring of specifications and modules by extension and parameterization; it also expresses the implementation, of both specifications and modules during the development of a software system.

The logic chosen for system configuration is many-sorted first-order logic which possesses the Craig interpolation property. Configuration is expressed precisely within the logical framework by the operation of combinators on recursively defined configured objects of sorts in the set {specification, module}; each configured object is a named theory presentation. Properties of commutativity between the combinators are illustrated by equivalent paths in the three-dimensional development space for configuration. The actual building of configured objects is expressed by constructing diagrams within a categorical workspace that is based on the structure of a KZ-doctrine.

Keywords: system, specification, modules.

1. Introduction

We have presented our logical approach to system construction in [1,3] and [4]. In particular we have contrasted our proof-theoretic approach with the model-theoretic to systems theory. Our own approach is simple and natural for software engineers to use because it does not rely on building a model of a system during the configuration of that system. A further advantage of systems configuration is that it involves keeping the history of system construction in the textual specifications of configured objects; this history is expressed in the formal diagrams that represent the specifications in the configuration workspace. The details of the construction of the categorical workspace for configuration are given in [2] and [5].

The aim of this paper is to demonstrate the flexibility of the system configuration by presenting the properties of commutativity between the

combinators of the configuration language. Initially the properties are illustrated by forming equivalent paths in the three-dimensional workspace. Examples are then given of textual specifications that express the different ways of configuring configured objects that have the same underlying structure. Finally we give the formal diagrams for these specifications and explain why these diagrams are equivalent in the categorical workspace.

The paper is arranged as follows. In Section 2, an intuitive view of system configuration is presented and the combinators in the configuration language are identified. Commutativity between the combinators that operate on the recursively defined system components is illustrated in the three-dimensional workspace. The formal diagrams are constructed within a categorical workspace in Section 3 and equivalent diagrams are shown to represent the commutativity between the combinators. Conclusions are drawn in Section 4.

2. The Development Space for Configuration

The *development space* for each system configuration allows the history of the configuration to be expressed diagrammatically.

2.1. The Three-Dimensional Development Space

The set of configured objects that describe a particular final configured system object are represented in a three-dimensional development space for that configuration. Objects within the development space are connected to other objects by arrows that lie along the three dimensions of the space, as shown in *Fig. 1*.

The arrows form a series of connected steps from the top left hand corner of the back face of the development space to the bottom right hand corner of the front face. Because each system configuration can be made in several different ways, with the history of each configuration recorded by different specifications, there will be several different series of connected steps within the development space for each system. The arrows that form the steps represent operations to configure objects by the single application of one of the high-level combinators, which are part of the configuration language.

Structure is added to a configured object within the space by horizontal development steps; configured objects are implemented by vertical development steps. The specification of each configured object that represents a component part of a system is configured on the back face of the development space; the module is created from a specification by a step in the third dimension to the front face of the space.

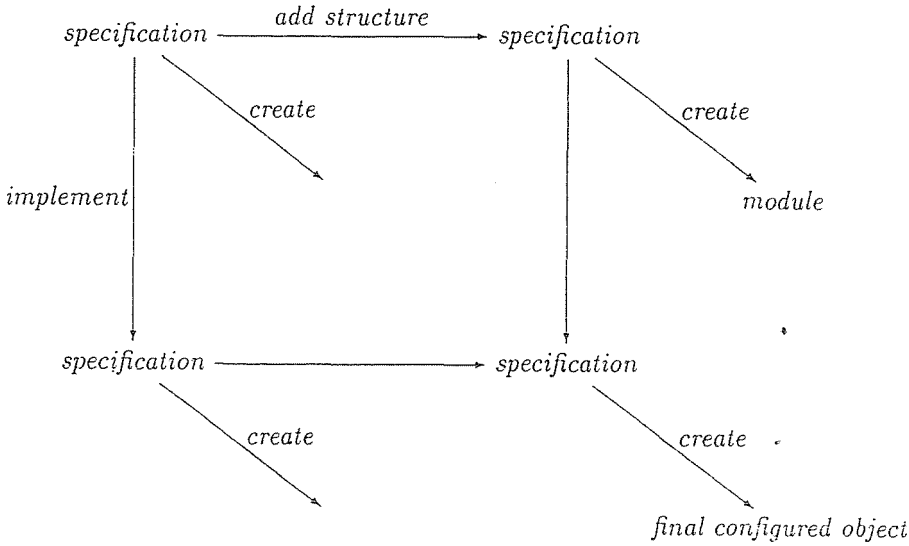

Fig. 1. A three-dimensional development space

The development space for each configured system has a recursive structure that reflects the recursive definition of the final configured object that represents the system. Each configured object that represents a component part of the system has a development space that is nested inside the space for the configuration of the final system. Modules, created in the third dimension of a development space, may themselves be structured and implemented within specifications on the back wall of the 'larger' space within which a more complex specification is configured. Each development space is shaped as a cube, therefore, which is without a front face.

2.2. The Commutativity of the Combinators

The rules for commutativity are presented informally and illustrated by informal diagrams in the three-dimensional development space. Finally an example is given of the commutativity between the high-level combinators for extension and the creation of modules.

The following properties of commutativity lead to flexibility within the configuration development space

create ; extend = extend ; create
create ; parameterize = parameterize ; create
create ; implement = implement ; create
extend ; parameterize = parameterize ; extend

extend ; implement = implement ; extend
parameterize ; implement = implement ; parameterize

These commutative properties are expressed as equalities in *Fig. 2* by forming commutative diagrams along the appropriate axes in our three-dimensional development space.

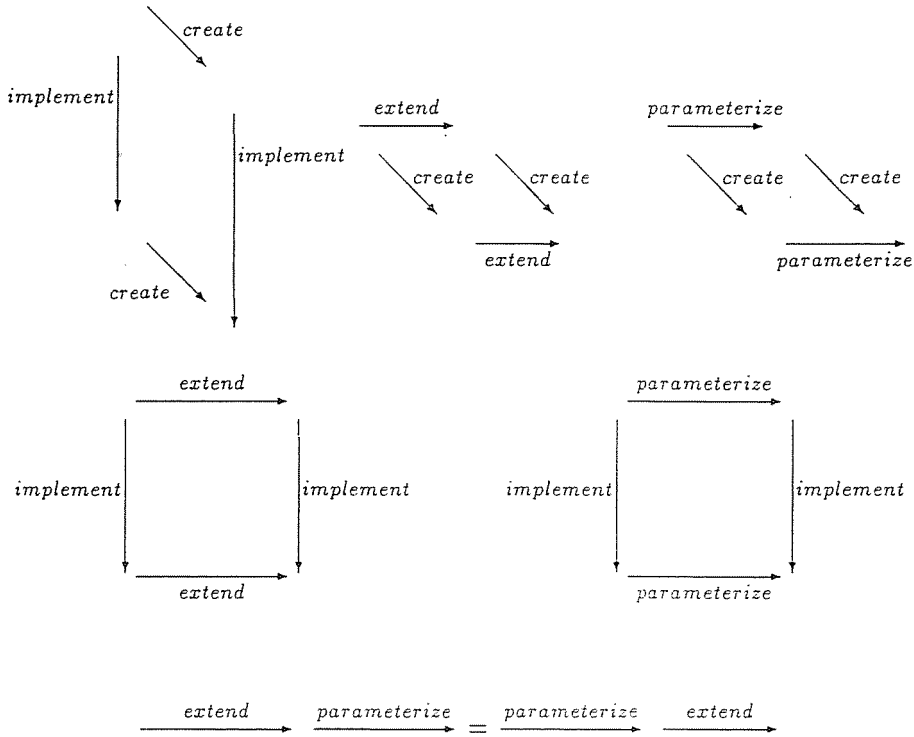

Fig. 2. Commutative properties of the high-level combinators

Only the create combinator is not defined on both sorts of object, and only the combinators for extension and parameterization are defined along the same axis of the development space. Commutativity between extension and parameterization is expressed as equality between horizontal paths in the development space.

Example 2.2.1

We extend a door by a window and then create the module instance of the extended object. A module with the same structure is then configured by first creating modules from the included specification and then extending the door module by the window module.

First the sequence *create ; extend* gives

spec door1_withwindow1 is
create (**spec** door, λ) = **module** door1 ;
create (**spec** window, λ) = **module** window1 ;
module door1 *ext. by* **module** window1 *to*
module door1_with_one_window1
endspec

Then the alternative configuration *extend* ; *create* gives a module with the same structure

spec door_with_window_module is
spec door_with_window is
spec door *ext. by* **spec** window *to* **spec** door_with_window
endspec
create (**spec** door_with_window, λ) = **module** door_with_window1
endspec

The difference between the two configurations is shown by the different histories that are recorded in the two specifications. In each specification, however, the door is extended by *including* the window in the door. Information about fitting the window in the door is contained in the object that is the extension.

Example 2.2.2

Commutativity can also be shown in the configurations of a module instance of a door extended by two windows. The configuration *create* ; *extend* gives

spec door_with_two_windows is
create (**spec** door_with_window, λ) = **module** door_with_window1;
create (**spec** window, λ) = **module** window1 ;
module door_with_window1 *ext. by* **module** window1 *to*
module door1_with_window1_window1
endspec

Alternatively *extend* ; *create* gives

spec door_with_two_windows_module is
spec door_with_two_windows is
spec door_with_window *ext. by* **spec** window *to*
spec door_with_two_windows
endspec
create (**spec** door_with_two_windows, λ) = **module**
door_with_two_windows
endspec

The **module** window1 is uniquely named in **spec** door_with_two_windows and is used to structure a more complex module,

in that same specification, within which it is in scope. Clearly each of the two specifications above expresses the structuring of a module instance of a door extended by two windows. The expression of the creation of uniquely named module instances of these specifications could be given in some more structured specification.

As a final alternative the sequence *create ; extend* will configure a door with two windows directly from the specifications for a door and a window.

spec door1_with_two_windows is
create (**spec** door, λ) = **module** door1 ;
create (**spec** window, λ) = **module** window1 ;
create (**spec** window, **module** window1) = **module** window2 ;
((**module** door1 *ext. by* **module** window1 *to*
module door1_with_window1)
ext. by **module** window2 *to* **module**
door1_with_window1_and_window2)
endspec

The module instance of this specification would be configured in some outer specification by the expression.

create (**spec** door1_with_two_windows, λ) =
module door1_with_two_windows1

3. Formal Diagrams Represent Configuration

The specification and the module instances of each configured object are represented in a categorical workspace of formal diagrams. The specification and the module instances of primitive objects are represented in this workspace by a diagram with a single node. The specification of a configured object is represented by a diagram that expresses the entire history of the configuration of the object. Each module instance of a configured object is represented by a diagram with a single node that has lost the history of the configuration of the object. In this way the semantics of configuration, expressed by formal diagrams, mirrors the syntax of the configuration language expressed by textual specifications. We illustrate the commutativity of the combinators by constructing the categorical diagrams of the specifications in our examples. There is no need to explain the structure of the workspace as a KZ-doctrine.

Example 3.0.1 (from Example 2.2.1)

$$door1^m \longrightarrow door1\_with\_one\_window1^m$$

is the diagram that represents **spec door1_withwindow1.**

$$door^s \longrightarrow door\_with\_window^s$$

is the diagram that represents **spec door_with_window.** A module instance of this specification is then created and is represented by the singleton diagram $door\_with\_window1^m$. The specification diagrams are defined to be equivalent in the categorical workspace because they have the same colimit objects.

Example 3.0.2 (from Example 2.2.2)

$$door\_with\_window1^m \longrightarrow door\_with\_window1\_window1^m$$

is the diagram that represents **spec door_with_two_windows.**

$$door\_with\_window^s \longrightarrow door\_with\_two\_windows^s$$

is the diagram that represents **spec door_with_two_windows.** A module instance of this specification is then created and is represented by the singleton diagram $door\_with\_two\_windows1^m$. The specification diagrams are defined to be equivalent in the categorical workspace.

4. Conclusions

We have focused on the properties of the combinators for our language for system configuration, and have illustrated the property of commutativity between the high-level combinators of *create* and *extend*. The commutativity can be shown in the three-dimensional development space, and in the categorical diagrams that are equivalent in the categorical workspace.

Acknowledgments

Tom Maibaum, of Imperial College, has made many helpful and constructive comments on this work. Steve Vickers, also at Imperial College, made the valuable suggestion that the categorical workspace could be structured as a KZ-doctrine.

References

[1] HILL, G.: Category Theory for the Configuration of Complex Systems. In: T. Rus, M. Nivat, C. Rattray and G. Scollo, editors, *Algebraic Methodology and Software*

Technology, Entschede, 1993, pages 193–200. *Proceedings of the Third International Conference on Algebraic Methodology and Software Technology*, University of Twente, The Netherlands, 21–25 June 1993, Springer-Verlag, 1994. Workshops in Computing series.

[2] HILL, G.: Constructing Specifications and Modules in a KZ-Doctrine. In: C.L. Hankin, I. Mackie and R. Nagarajan, editors, *Theory and Formal Methods '94*, pp. 219–236. *Proceedings of the Second Imperial College*, Department of Computing, Workshop on Theory and Formal Methods, Imperial College Press, distributed by World Scientific Publishing Co. ISBN 1-86094-003-X, September 1995.

[3] HILL, G.: The Configuration of Complex Systems. In: Tuncer I. Ören and George J. Klir, editors, *Computer Aided Systems Theory–CAST '94 Selected Papers*, pp. 46–64. *Fourth International Workshop*, Ottawa, Ontario, Canada, May 1994, Springer-Verlag ISBN 3-540-61478-8, May 1996.

[4] HILL, G.: A Logical Approach to Systems Construction. In: Rudolph Albrecht and Franz Pichler, editors, *Proceedings of EUROCAST '95 LNCS 1030*, pp. 30–47. Fifth International Workshop on Computer Aided Systems Theory, Innsbruck, May 1995, Springer-Verlag, January 1996.

[5] HILL, G.: A Categorical Workspace for System Configuration. In: S. Jourdan, A. Edalat and G. McCusker, editors, *Advances in Theory and Formal Methods of Computing*. Proceedings of the Third Imperial College Workshop, Christchuch, Oxford 1–3 April 1996, published by Imperial College Press, distributed by World Scientific Publishing Co., 1997.