

Composing Dynamic Hyperslices

Ruzanna Chitchyan, Ian Sommerville

Computing Department, Lancaster University, Lancaster, LA1 4YR, UK
{rouza | is}@comp.lancs.ac.uk

Abstract. This paper describes a composition mechanism for the dynamic hyperslices model outlined in [1]. This mechanism composes primary concerns, directly aligned with requirements and designs (decomposed in accordance with the Hyperspaces approach), maintaining each concern unchanged as a first class entity all through its lifetime. Consequently this mechanism allows for dynamic change and re-configureability of the resultant systems.

1. Introduction

Recent work in the field of Software Engineering has resulted in a set of approaches collectively termed Aspect-Oriented (AO) Techniques [2]. AO introduces yet another level of decomposition: separation of crosscutting properties in software, and brings to light the notion of *concern*. A *primary concern* – a matter of interest in a software system that cannot be decomposed into smaller meaningful parts, can now be identified as the atomic unit of any software artefact¹. It has been shown that it is the inappropriate separation of these concerns [3] that results in monolithic designs [4] and code, which are difficult to understand, maintain and reuse.

Some research has already been carried out on modelling concerns [5], mapping them to design [6] and implementation units [7].

Our work on the Dynamic Hyperslices model [1] follows this line of research, aiming to preserve the unchanged primary concerns as first class entities all through the life cycle of software that concerns form part of – from concern modelling to software run-time. This path leads to flexibility in concern manipulation, reuse and maintenance, as well as dynamic change and re-configureability of the resultant systems. However it shifts the complexity of development and maintenance into concern composition. In view of the increasing importance of composition, we have proposed [1] to distinguish it as a separate *developer-related* concern (i.e. a concern which arises due to specific development-related activities carried out by software developer).

In the present paper we provide some detail on the composition mechanism of our Dynamic Hyperslices model: section 2 describes the background work upon which the Dynamic Hyperslices model is based, section 3 briefly outlines the model followed by the outline of the composition mechanism in section 4 and analysis of possible change scenarios and correctness of composition in section 5. We conclude with brief summary and future work in section 6.

¹ Conceptual concern, not programming language expressions, such as variable declaration etc.

2. Background

As software becomes ever more closely integrated in our everyday life, on one hand costs of interruptions for maintenance and change of some systems become disproportionately high (e.g. safety critical systems), on the other hand demand for higher adaptability of systems grows (e.g. mobile and disappearing computing). We suggest that dynamically composeable systems could provide a solution for these and other similar problems by providing for dynamic changeability and context-sensitive re-configureability. One such approach – A Model for Dynamic Hyperslices – is discussed below.

In developing the Dynamic Hyperslices model we draw on the concern decomposition mechanism of the Hyperspaces approach, message interception and manipulation ideas of the Composition Filters approach and component integration mechanism of Connectors. The present section provides a brief description of these technologies.

2.1 Hyperspaces

This approach [8], [9] proposes to use a set of modules each of which address a single concern (called hyperslice). Hyperslices can overlap, i.e. a given unit in a hyperslice can appear, possibly in a different form, in other hyperslices and dimensions of concerns². All the concerns of importance are modelled as hyperslices, which are then composed into hypermodules (i.e. compound hyperslices with a composition rule specifying how the hyperslices must be composed) or to a complete system. At the composition stage issues such as overlapping are resolved via composition rules.

Composition is based on commonality of concepts across units: different units describing the same concept are composed into a single unit describing that concept more fully. To compose one needs to match units in different hyperslices that describe the same concepts, reconcile their differences and integrate the units to produce a unified whole. Composition rules specify the relationships between composed hyperslices.

In HyperJ [10] (a composition tool developed for OO instantiation of Hyperspaces) composition-related concerns are not treated as first class entities. Although hypermodule composition is specified in a separate composition file, it is only a transitory unit. Consequently, when the elementary concerns are composed, they get contaminated with properties of the composite concern³.

In the Dynamic Hyperslices approach we differ in composition from HyperJ (see section 3) but we maintain the decomposition principles of Hyperspaces. We also clearly define two types of concerns: user and developer-related, both of which are to be treated as first class entities all through the software development and maintenance process.

² Dimensions of concerns are ways of decomposition, such as for instance per object classes, per viewpoints, per features etc. This concepts stem from the multi-dimensional decomposition approach, for which Hyperspaces approach is an instance.

³ More discussion available in [1]

2.2 Composition Filters

The Composition Filters (CF) model [11] extends the Object-Oriented model in a modular and orthogonal way. Since behaviour in the OO model is implemented by exchanging messages between objects, the CF model proposes to use a set of *input and output filters* for message interception and manipulation. By wrapping these filters around the objects, CFs are able to manipulate object behaviour without directly invading object implementation.

The CF model is very well suited to implementing concerns that lend themselves to modelling through message-coordination and introduction of actions executed before or after executing a method (e.g. intercept message, put record of message arrival in Log file, execute message).

Our Dynamic Hyperslices utilise the message interception and manipulation capabilities of Composition Filters. Filters form part of our *composition connectors* (see section 3).

2.3 Architectural Connectors

The concept of connectors originates from the area of software architecture [12] [13]. Connectors were proposed to facilitate component integration by catering for specific features of interactions among components in a system. The current work in this area argues for giving connectors a first class entity status because they contribute towards the better understandability of system architecture [14] through localising information about interactions of components in a system; capturing the design decisions and rules of interactions amongst components; handling incompatibilities between components and so on. In [15] the idea of connectors as run-time entities is discussed.

Unlike the previous work on connectors, the composition connectors in our model are connectors for hyperslices (sections 3&4), i.e. not (necessarily) for complete object classes or (OO) components. Our connectors don't simply match provided/required services, or specify roles for connected components, but rely on dynamically updateable composition strategy to build up functionality of coarser-grain components (e.g. object classes) from primary hyperslices⁴, as well as carry out the communication between the member hyperslices at run time.

3 Brief Outline of the model for Dynamic Hyperslices

The model for Dynamic Hyperslices [1] intends to provide a composition mechanism that will allow all the primary concerns, decomposed in accordance with the Hyperspaces approach, to endure in the composite concerns after composition.

The model:

⁴ Thus, the composition strategy in the connectors can be perceived as a kind of “merger algorithm” for producing higher order artifacts. Here the “merger” is performed through run-time message manipulation within connectors, without physically merging the hyperslices.

- Uses the Hyperspaces decomposition approach in separating concerns into single-minded hyperslices (or primary concerns).
- Requires that an additional dimension for Composition concerns is specified in each Hyperspace-type decomposition. This additional dimension contains connector-concerns. At the composition stage the connector concerns are used to compose other concerns.
- Utilises a composition connector to integrate any primary/composite concerns. Consequently, any interaction between other concerns will be channelled through a set of connectors.
- Provides connectors with capability to reflect upon their immediately connected concerns, while still keeping these internals hidden from all other connectors and hyperslices.

Figure 1 below provides a high-level diagram for the model. In this model composition concerns are first class entities (depicted as ovals) which also retain hyper-slice integration information. All *user-related* concerns used in the system are retained unchanged at runtime (depicted as squares with solid borders) and all interactions between the concerns are resolved through their connectors (depicted as solid arrows). This model is aimed to be open and extensible, as concerns can be added/updated/removed by simply adding/updating them and their respective connectors.

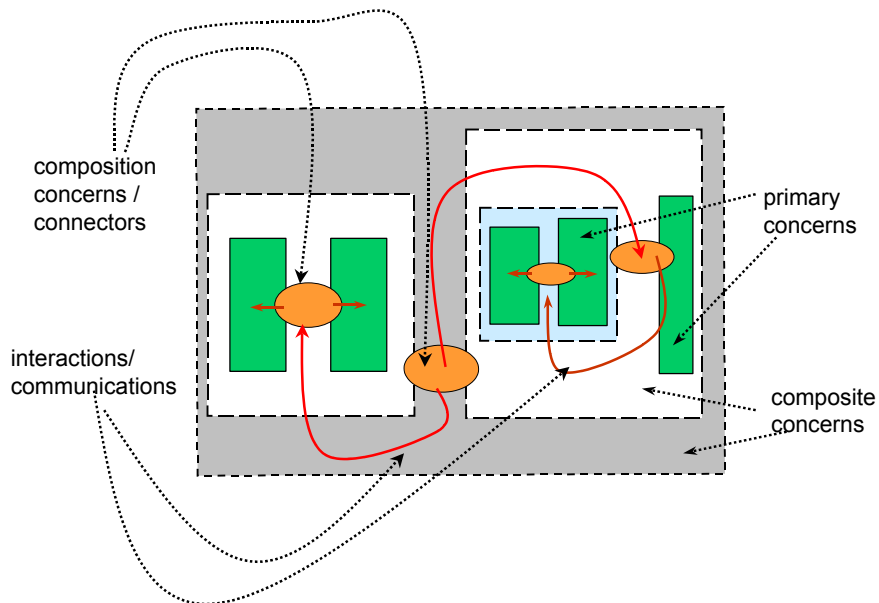


Fig. 1. An outline of the Dynamic Hyperslices model

4 Connectors

Our connectors decouple hyperslices in a hypermodule and promote reuse of individual concerns and their compositions. Since at any level each of the primary concerns remains intact, changing requirements can be easily mapped onto a composite hyperslice by modifying the out-of-date primary concern. In the process of primary component change only its immediate connectors will be affected, other parts of the model will automatically adjust to changes, if necessary. Similarly a composite part of a hyperslice can be updated/replaced with change introduction localised in its immediate connectors. This locality attribute of our architecture arises due to structuring it around dynamically updateable composition strategies and hiding all (levels of) hyperslices, except for the directly participating ones, from composition. A connector structure is depicted below in Figure 2. It consists of the following elements:

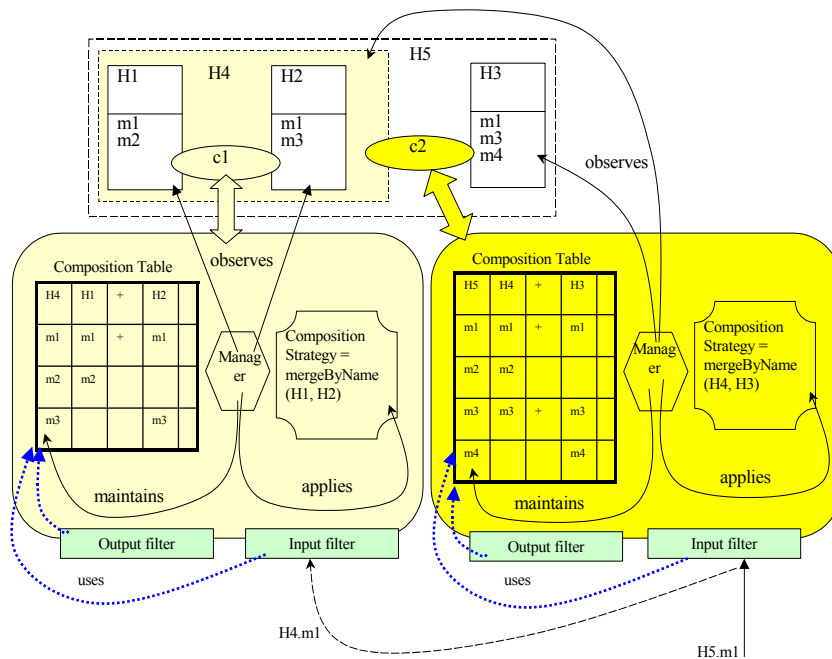


Fig. 2. Elements of a Composition Connector

Sets of input/output filters: these are used to intercept incoming/outgoing messages sent to the hyperslice and manipulate the message in accordance with filter specification. Filters are similar to those defined in the Composition Filters approach in that they can manipulate or substitute the target and selector of the intercepted message. Thus, for instance, a certain filter type (lets call it dispatch filters as in CF) will be able to re-direct the intercepted message to a different hyperslice or put it through for execution to the initially intended target.

Composition table contains the records of the subjects that constitute the composed hyperslice. The first column in the table displays the hyperslice public interface. Each of the following columns contains the references to the hyperslices immediately connected by the present connector (top row of the table) and the elements of the connected hyperslices that contribute to the corresponding unit of interface in the resultant hyperslice (all but top row in the table). The composition operators between the column elements represent the operations required to compose the individual hyperslice elements into that of composed hyperslice. The operators are applied in accordance with the Composition Strategy specification.

Composition Strategy is the specification of how exactly should the constituent hyperslices be composed. For instance the *mergeByName* strategy is used in the example shown in Figure 2, stating that all elements with same names in constituent hyperslices should be merged into one element of the resultant hyperslice.

Connector Manager: as suggested by its name, this element is responsible for overall “management” of the connector. It uses the Composition Strategy and the public interfaces of the contributing hyperslices to fill in the Composition Table. It also keeps the other elements under observation and updates the composition table content when any of the contributing constituents gets updated.

5 Consistency Preservation and Correctness Checks

5.1 Change Scenarios and Consistency Preservation

The following set of scenario analyses illustrates how our composition model will deal with some possible changes in the constituent hyperslices. All scenarios are based on the case illustrated in Figure 2.

Scenario 1: *The implementation of one/several methods in constituent hyperslices is changed, but their interface is maintained.* No change is required to any part of any connector, the updated method will be used when a call is directed to it.

Scenario 2: *The interface of a method has changed - method $m1$ in $H1$ has been renamed to $met1$ - but $met1$ is still to be part of $m1$ in $H4$, i.e. the composition has not changed.* In order to maintain the consistency of the Composition Strategy (since it did not change) the $c1$ Connector Manager adds a new clause to Composition Strategy, indicating that $met1$ is an equal name for $m1$. Then it updates the Composition Table in the connector. The second row of the Composition table will change from $[m1|m1|+|m1]$ to $[m1|met1|+|m1]$, indicating that the $m1$ in public interface of $H4$ consists of the merge of methods $met1$ in $H1$ and $m1$ in $H2$. No further change is required.

Scenario 3: *Method $m2$ is deleted from $H1$.* The Connector Manager of $c1$ removes $m2$ from its composition table. The Connector Manager of $c2$ detects that the interface of $H4$ has changed and updates its composition table by first removing $H4.m2$, then since $H5.m2$ has no constituent parts, it is also removed. Thus $m2$ is removed from the interface of $H5$.

It should be noted that subtractive changes to the hyperslices’ interfaces will be guarded by use counters. Before a subtractive change use counters of respective

hyperslices will be checked to verify that the items marked for deletion are not currently in use. If they are – the change will be postponed till use counter is reduced to 0. Principles of pertinence from [16] could also be beneficial here.

Scenario 4: *Method m7 is added to H1*. The Connector Manager of c1 adds *m7* to its composition table, then c2 Connector Manager updates its composition table with *m7*, thus *m7* appears in the interface of H5.

In all the above scenarios changes introduced to the primary hyperslice are either localised within the immediate connectors of these hyperslices, or automatically propagated to the coarser granularity connectors by respective Connector Managers. Consistency preservation is also a task of the respective Connector Managers, supported by a set of composition rules.

5.2 Factors Facilitating Correctness Checking

Several properties of this model facilitate correctness checking of compound hyperslices yet allowing for good changeability (discussed above) of its constituents:

- *Reduced complexity of checking*: smaller single-minded concerns reduce the complexity of the specification, design and implementation, and verification tasks;
- *Checking individual concerns*: since the composition does not affect the concerns in any way, each concern can be checked and tested against its own specification initially, independently of its composition context. This helps to assure correctness of constituents in the composite.
- *Incremental correctness check*: since the larger modules are built by incrementally composing individual concerns, incremental verification of composition is also supported, easing the testing of larger composite units.
- *Direct mapping of change*: as traceability of artefacts at different levels is preserved (due to use of the Hyperspaces decomposition), change in any of the requirements will be directly reflected in its design/implementation concern, and will be easier to trace and validate.

All of the above factors could facilitate correctness checks in our model; however the precise techniques for checking need to be developed.

6 Summary & Future Work

We have observed that simplification of software development through new software decomposition techniques (such as those provided by Aspect-Oriented paradigm) tends to move complexity into the composition process. Consequently, we are working to produce a model to separate composition concerns themselves into first class entities and simplify the composition process. In this paper we have discussed our model for Dynamic Hyperslices, which closely follows the spirit of AO, and outlined its composition approach.

There are a number of open issues in our model that need to be addressed, for instance we are working on providing a clear structure for *Connector Manger* elements, refining the composition mechanism, defining techniques for verifying correctness of composition, and developing a meta-model for the Dynamic Hyperslices Model.

Some of our future work will include implementation of a system that realises this model, investigating ways of incorporating domain-specific knowledge into the composition process.

References

1. Chitchyan, R., I. Sommerville, and A. Rashid. A Model for Dynamic Hyperspaces. in Workshop on Software engineering Properties of Languages for Aspect Technologies: SPLAT (held with AOSD 2003). 2003.
2. Elrad, T., R. Filman, and A. Bader, Theme Section on Aspect-Oriented Programming. Communications of ACM, 2001. **44**(10).
3. Tarr, P.L., et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. in Proc. 21st International Conference on Software Engineering (ICSE 1999). 1999: ACM.
4. Clarke, S., et al., Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code, in Proc. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). 1999. p. 325-339.
5. Sutton, S.M. and I. Rouvellou. Modeling of Software Concerns in Cosmos. in International Conference on Aspect-Oriented Software Development. 2002: ACM.
6. Clarke, S., Composition of Object-Oriented Software Design Models, in School of Computer Applications. 2001, Dublin City University.
7. Clarke, S. and R.J. Walker, Mapping Composition Patterns to AspectJ and Hyper/J, in Workshop on Advanced Separation of Concerns (ECOOP 2001). 2001.
8. Ossher, H. and P. Tarr, Multi-Dimensional Separation of Concerns using Hyperspaces. 1999(21452).
9. Hyperspaces. 2003, IBM Research; <http://www.research.ibm.com/hyperspace/>.
10. Tarr, P.L. and H. Ossher, Hyper/J user and Installation Manual. 2000: IBM Research.
11. Bergmans, L. The Composition Filters Object Model. in RICOT symposium: Enabling Objects for Industry. 1994.
12. Garlan, D. and M. Shaw. An Introduction to Software Architecture. in Advances in Software Engineering and Knowledge Engineering. 1993. New Jersey: World Scientific Publishing Company.
13. Allen, R. and D. Garlan, A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology (TOSEM), 1997. **Volume 6**(Issue 3): p. 213 - 249.
14. Balek, D., Connectors in Software Architectures (PhD Thesis), in Faculty of Mathematics and Physics. 2002, Charles University: Prague.
15. Ducasse, S. and T. Richner. Executable Connectors: Towards Reusable Design Elements. in ESEC '97. 1997: LNCS.
16. Benatallah, B. and Z. Tari. Dealing with Version Pertinence to Design an Efficient Schema Evolution Framework. in International Database Engineering and Applications Symposium (IDEAS). 1998. Cardiff, Wales, UK: IEEE Computer Society.