Applying a Grid Based Approach to End User Component Composition

Erlend Stav
Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), N-7491 Trondheim, Norway
erlend.stav@informatics.sintef.no

ABSTRACT

The idea of quickly and easily creating software by composing and configuring existing components is appealing. Experience from end user programming can probably be applied to create component composition environments that can be used by end users. Ideas on how a grid based end user programming approached are presented, along with directions on how the ideas can be validated.

Keywords
End user programming, component composition, component software, software composition

1. INTRODUCTION

The hypothesis behind this work is that experience from the field of end user programming can be used to create environments that enable end users to compose applications from software components. There are many approaches to end user programming. This paper focuses on the grid based approaches used in products like Stagecast Creator\(^1\) and AgentSheets\(^2\), and presents some initial thoughts on how these can be used applied for component composition. The goal of the paper is to bring up questions more than to provide answers.

The next section presents components and objects, while component composition is presented in section 3. Section 4 gives a brief introduction to end user programming environments, with a focus on grid based environments, and is followed by some initial thoughts on how these can be applied to component composition. The final section outlines future work.

2. COMPONENTS AND OBJECTS

Software components have received a lot of attention the last years, both in the academics and the software industry. In publications and software documentation the term “component” is often not defined, and thus is not a very precise term. Recently the definition made by the Workshop on Component Oriented Programming at ECOOP 1996, and referred in Szyperski’s influential book [19], has been adopted by many authors:

A software 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.

A unique property of software that makes software components different from components in other disciplines is described in [19]. “Rather than delivering a final product, delivery of software means delivering the blueprint for products”. This cause some problems when we try to compare components used in other disciplines to software components, and both “plans” and “products” are frequently referred to as components. Usually a software component allows multiple instances of objects to be created from it. To keep the terminology precise these instances should not be called components, even in the special case where only a single object instance can be created from the component.

Components contain classes and static objects (class objects). The interface of a component is defined by the classes and static objects it provides.

The commercial arena is currently dominated by three base technologies:

- Microsoft’s COM-based technologies [12]
- OMG’s Object Management Architecture with CORBA [20] and CORBA Components
- SUN’s Java approaches (including JavaBeans [8] and Enterprise JavaBeans [18])

All of these approaches are based on object-orientation. By many of its proponents, class-based object orientation where initially considered “the solution” to software reuse. Components in addition require dynamic linking, and emphasis is on polymorphism instead of inheritance and delegation [15]. Classes and inheritance are still used for implementation within the components.

---

\(^1\) Stagecast Creator is a trademark of Stagecast Software, Inc.

\(^2\) AgentSheets is a trademark owned by the Regents of the University of Colorado, licenced to AgentSheets, Inc.
3. COMPONENT COMPOSITION AND CONFIGURATION

The idea of quickly and easily creating software by composition and configuration of existing components is appealing. The analogy of building with Lego blocks is popular. However, in reality things are unfortunately much more complex. If components are created with no common ground to guide their design, it can not be expected that will easily collaborate. [10] points out that reuse is not as pervasive in object oriented development, as early proponents assumed it would be. Part of the problem is that component are usually not perfectly co-ordinated.

The last decade a lot of effort has been put into dealing with this problem. [9] proposes Contracts as an approach to specify behavioural compositions in object-oriented systems. Contracts provide a large grain abstraction that helps understand and reuse software. Contracts supports refinement and inclusion as mechanisms for expressing complex behaviour in terms of simple behaviour.

In [2] superimposition is proposed as a component adaptation technique. A set of requirements for adaptation techniques is described; they should be transparent, black-box, composable, configurable and reusable. Well-known techniques like copy-paste, inheritance and wrapping are analysed, and each are found to support only some of these requirements. In superimposition both a set of reusable components and a set of reusable component adaptation types are used, where an adaptation overrides some behaviour of the component.

In [21] a component framework is defined to be set of rules to be obeyed by components in a certain environment. These rules ensure that the components can coexist and collaborate in a single environment. The framework can exists as only a set or rules, or can implement management of shared resources and provide for communication be. [19] further argues for the importance of component frameworks.

[3] proposes a set of requirements for component composition languages. Among the requirements listed is support for composition operations for binding communication channels, creating higher level component aggregates, macro expansion of parameterised components and recursive component composition. Also the language should allow "glue code" to deal with compositional mismatches, and it should support component frameworks.

Although there are notable differences between the COM/CORBA/Java-based component technologies, their focus when it comes to composition and configuration is quite similar, and can be summarised as:

- **Properties.** The objects created from the components have properties that can be edited by the component user to adjust e.g. appearance and behaviour.
- **Methods.** The objects have sets of methods that can be called.
- **Events.** Events are frequently used to allow clients of the objects to react to situations of interest.
- **Connections.** Compared to events, connections usually allows closer and more detailed interactions through an interface consisting of a set of methods.

Some research has been done on visual approaches to component composition. Some examples of this are [5], [7] and [17].

Most of the current commercial development tools, e.g. Borlands JBuilder, allows visual composition only of the user interface, although there are examples, e.g. IBM’s Visual Age, that allows visual composition also of non-visual parts of the application.

4. END USER PROGRAMMING

[13] contains a wealth of information on the field of end user programming. Some of the highlights are:

- people are able to learn fairly complex formal systems when they have a motivation for doing it. Examples of this are base-ball scoring sheets and knitting recipes
- end user programming languages do not necessarily need to be visual to be successful
- domain specific approaches work well

Among the examples of successful approaches presented in the book, spreadsheets are one of the foremost.

There are numerous approaches to end user programming. Among the approaches that has been successfully applied are:

- grid based approaches
- programming by example
- spreadsheets
- scripting languages
- flowchart-based approaches
- macros

Experience from many of these approaches may be of interest to component composition. In the rest of this paper the focus is on grid based programming by example approaches. A good collection of examples on programming by example is found in [4]. Other approaches may be studied in future work.

4.1 Grid Based End User Programming Approaches

Stagecast Creator is a grid based end user programming environment. The user creates an application by placing characters on a page that is divided into a grid of equally sized cells. Each character has one or more appearances that can be edited, and a set of rules that decides the behaviour. The rules are organised as an ordered list.

The rules of the characters are created "by example". The definition of a rule usually starts by selecting the character and
optionally a set of neighbour cells that is part of the context for
the example. The contents of these cells when the rule is created
are part of the precondition for the rule to trigger. The user can
then demonstrate the effects of the rule by modifying the
situation. This typically involves changing the content of one or
more of the involved cells, changing the appearance of one or
more characters, or setting some other variable. The effects can
later be viewed and edited further as a list of actions. Figure 1
shows how a rule for moving an actor to the right on the grid is
created in Stagecast Creator.

The execution of the application is divided into time steps. For
each step the rules of each actor are examined, and the first rule of
each actor which matches the preconditions (if any such is found)
is executed.

The basic principles of Agentsheets are very similar to Stagecast.
The tool is slightly less visually oriented. Instead of selecting an
area of neighbour cells, conditions like “See a” and “Next to” are
used to check on the content of the neighbour cells. Instead of
demonstrating the effects of the rule, the user inserts a set of
actions like “move”, “new” or “erase”. Figure 2 (from [16])
shows an example application in Agentsheets that allows
prototyping of a voice dialog. In this case the agents are built
within the tool, but in a component based environment some of
the agents could be instances of objects defined in a component.

One of the reasons these tools seem to be successful is that the
grid based model is very simple to relate to and understand. The
application is nothing more or less than the grid and the
characters/agents (called actors from now on) contained in the
cells. There is not any hidden, more complex “reality” that causes
unexpected and unexplainable behaviour, although in more
complex situations the application can show emergent behaviour
not expected by the user as a result of interaction between the
rules of multiple actors.

Another success factor is probably an easily understandable
execution model. Somewhat simplified, the execution is divided
in steps, and each actor uses the first rule in its list that matches
the precondition (although it is also possible to include more
event-based tests like checking for a pressed key, or doing
something with e.g. each second).

Figure 1 - Creating a rule in Stagecast Creator

Figure 2 - Voice dialog example from Agentsheets
A simple UML model of the basic object types involved in a grid based environment is displayed in Figure 3 (this is a user’s view and not an attempt to describe the actual implementation of the two tools described here).

Given that the grid based environments seem to work well with end users: how can we apply some of the techniques to component composition? This is the topic of the next section.

5. GRID BASED COMPONENT COMPOSITION

This section presents some initial thoughts on how techniques from grid based end user programming can be applied to component composition. The ideas presented here are not in any way tested or validated, and are just meant as an initial exploration of possibilities.

What might it mean to introduce components into an environment similar to the grid based environments presented earlier? The most obvious use of components in this environment would be to let the components introduce new actor types with some predefined behaviour and properties. In the model of Figure 3 these would be objects of type ActorClass, from which actor instances could be created.

Components could also introduce frameworks applicable to specific domains. Such frameworks could come with wizards, or support templates like the ones described in [1], that would quickly set up an initial application structure. Such a scenario could potentially add objects of all the types described in Figure 3.

In the applications developed with Agentsheets and Stagecast the grid is usually used both as UI and logic of the applications. When used for component composition, an abstraction is introduced, as it is not likely in the general case that a visual presentation of the grid is used at runtime (except for debugging purposes).

In grid based programming the content of neighbour cells are frequently used when describing rules. Although used for a more abstract purpose in component composition, it is natural to exploit the interpretations the environment affords. Some natural mapping could be:

- Move objects together so they become neighbours in example –> create connection when rule triggered
- Move objects apart so they are no longer neighbours in example –> remove connection when rule triggered
- Add object to example –> create instance when rule triggered
- Remove object from example –> remove instance when rule triggered

When new actor types are introduced by components, these can introduce new mappings to the environment. A container actor could e.g. cover multiple cells, and allow another actor to be added or removed from its content by moving the other actor into or out its boundary.

A grid based approach naturally has some limits to the number of neighbours an object can have (4 or 8 depending on whether diagonals are included, as long as all objects cover a single cell). This may not be a serious problem in reality, as the number of connections for each object is usually limited. One possible way to avoid this limitation altogether is to treat different concerns separately; e.g. by keeping a set of grids that together describe the whole situation, and where the system finds the composed behaviour.

One property of the grid based environments that may not be useful to component composition, is the time-step based execution model. An alternative could be to use an event-based approach.

The traditional ways of composition and configuration of components can also be integrated into a grid based environment:

- **Properties** – simple to handle with this approach. Initial values can be edited with property sheets. Changes to properties can be done part of the actions of triggered rules (this is already used in the mentioned tools). The property change can be demonstrated during rule creation, by recording the editing from a property sheet.

- **Events** – can be handled similar to preconditions for rules (as keyboard/mouse handling is done in Stagecast), or used as triggers for rules instead of the time-step based approach. The current tools usually describe the event handling rules on the object where the event originates.
• **Methods** – messages can be sent as part of the action-list of the rules, quite similar to how it is done in the tools now. New actor types introduced by components can present their own set of public methods.

• **Connections** – already mentioned above. Could be automatically created when the objects are moved next to each other. It may in some cases be more than one kind of connection that can be established between two neighbour objects. Selecting and specifying the details of the connections when they are created, e.g. using a wizard or dialog, can solve this. The details become part of the resulting actions in the rules. Default "counter"-actions for the event that the two objects are move apart from each other could also be created as part of the same process.

6. **FUTURE WORK**

As already mentioned the ideas presented in the previous section are just an initial exploration of the possibilities when applying grid based techniques to component composition. A first step to check the ideas and develop them further would be to decide on some example applications and attempt to express these. If the result from this is promising, the ideas can be tested further by making prototypes (paper or software) and test it on the target user group.

Among the broader questions and challenges for further investigation is the following:

• Who should perform component composition? What are the limits for what an end user can do?

• Design and development of frameworks that enable rapid creation of suitable domain-specific environments

• Is creation of domain-specific component-composition environments at all worthwhile, or is the cost to great?

Recently some new approaches have been proposed to address weaknesses in the traditional object oriented technologies. One of the new approaches that has emerged is aspect-oriented programming. [11] claims that object-orientation is good at expressing the base functionality of a system, but some other design decisions are difficult to cleanly capture in code. These aspects (e.g. synchronisation, error handling, constraints and optimisation) cross-cut the base functionality of the system, resulting in tangled code. The proposed solution to this is to code the aspects separate from the base functionality, and use a "weaver" to merge the code into a solution.

One of the goals of object-orientation is to achieve simplicity through separation of concerns. The concerns are decomposed into classes, but some concerns like features and aspects do not map well to classes, and causes scattering and tangling of code. [14] criticises this "tyranny of the dominant decomposition" that object-orientation is one example of, and proposes the Hyperspace approach to address multi-dimensional separation of concerns.

A challenge here is to see if any of these approaches can be successfully combined with a component based end user programming environment. One use of this could be to allow end users to compose applications by combining features.

7. **REFERENCES**


