A Checking Mechanism for Visual Language Environments

Kai HERRMANN, Ulrich HOPPE, Niels PINKWART
Institute for Computer Science and Interactive Systems
Faculty of Engineering, University of Duisburg-Essen
47048 Duisburg, Germany
Phone: (+49) 203 305 11 273
{herrmann, hoppe, pinkwart}@informatik.uni-duisburg.de

Abstract. This paper presents CCM *(Checker for Cool Modes)*, a checking system which enables the user to specify and check requirements of a potentially complex graph structure. We have used it in a collaborative visual language environment called "Cool Modes" which is used mainly in distributed learning/teaching scenarios. CCM supports the learning process through incremental feedback which is automatically generated by comparing the student's solution to a given target constellation. The feedback messages as well as the target constellation can be specified interactively by teachers or educational designers. All the information produced in and used by CCM is stored in XML and processed in Java. Yet, no specific knowledge of these technologies is needed to use the system.

1. Introduction

In phases of classroom exercises, when students have to solve problems in *parallel* individually or in small groups, the teacher is often confronted with a lot of questions at the same time. Particularly questions like "Look! Is this correct?" draw much of the attention of the teacher. In some cases, a computer could answer these questions just as well. Thus, a computer system that liberates the teacher from answering these "standard questions" would help him to focus on more substantial tasks like, e.g., the support of weaker learners. The checking mechanism introduced in this paper offers this feature. During his *lesson preparation*, the teacher develops an ideal solution of the task he wants to give to the students. CCM then creates a first version of a multi-level feedback automatically. After that the teacher can (and usually has to) edit this feedback in different ways. During the *school lesson*, the checker then supports the students autonomously by giving feedback about their mistakes and advances, without any further need of teacher intervention.

More general, CCM is a domain-independent analysis tool: it checks objects of visual languages with respect to different parameters. These are user-defined and easily expandable; they vary from the geometric orientation of components to the number of marks in a Petri Net place. Although there are some existing checking mechanisms, ranging from simple multiple choice test and crossword puzzle checking [1] to more sophisticated applications like CAD systems checking [2], all of these are domain-specific. In opposite to this, CCM is domain-independent and works with environments ranging from System Dynamics simulation to learning material in the domain of Jewish rituals.

As mentioned before, the checker is implemented as a plug in for the Cool Modes environment. Cool Modes ("COllaborative Open Learning and MODEling System") [3] is a platform and tool environment designed to facilitate co-constructive activities. It offers a

shared workspace environment allowing the co-learners to synchronously and jointly elaborate external graph representations based on visual languages. The main difference between Cool Modes and other comparable tools like Sepia [4] or Belvedere [5] is a plug-in approach for flexibly adding externally defined semantic structures of specific visual languages without a priori assuming a given specific domain semantics for the overall system. This allows for integrating multi-purpose structuring tools, e.g. for discussion and argumentation, with specialized domain-related functions, especially modeling languages, in the way proposed by van Joolingen [6] for "collaborative discovery learning" in science education. Existing examples for domain-specific Cool Modes extensions are, e.g., a Petri Net Simulator and a System Dynamics environment.

Cool Modes and similar collaborative-interactive environments exploit the interaction principle of direct graphical manipulation as opposed to command or dialogue driven interaction. This gives users a maximum of choice and initiative but is difficult to analyze on the part of the machine. Mühlenbrock and Hoppe [7] have proposed an analytic approach based on action primitives, whereas Constantino-Gonzalez and Suthers [8] have focused on the analysis of a problem state in a specific task domain (ER modeling). Our CCM implements a light-weight approach which exploits general state information in Cool Modes, namely geometry and abstract graph structure of an object-oriented representation. This implies that it works with a variety of different representation languages.

CCM is a *state oriented* analysis-tool. It only takes into account the *current* state of the workspace and does not consider the history that led to that state. This way, it supplements the action-based analysis tools already developed in the COLLIDE group [7]. Although the Cool Modes framework inherently supports collaborative work, CCM was not specially developed for distributed scenarios. One of the future goals will be the integration of a user awareness component to support distributed learning more efficiently.

2. Parameters of the Checking Mechanism

The *Checker for Cool Modes* enables teachers or educational designers to precisely define conditions that a workspace content (more precisely: the node structure of a JGraph) has to fulfill. A JGraph is the fundamental data structure in Cool Modes and contains both an abstract graph and its visual representation. CCM works with the nodes and edges of a JGraph and checks conditions that have to be fulfilled by them, beginning with things like their position on the screen up to more domain-specific constraints. Currently, there are five different condition types that can be checked. These can be grouped into two categories: One group that can work with all Java objects that extend the `java.awt.Component` class, and a second group where the nodes and edges must fulfill special conditions.

2.1. Standard Checking Mechanisms

First, we explain the four standard checking conditions which work for all types of nodes.

- *Checking the existence of nodes:* For the existence check, CCM uses the Java class name as identifier, i.e. it counts instances of certain Java classes present on the screen. E.g., assumed that for a special task, a Petri Net must have six places and five transitions, CCM would produce negative feedback messages as long as not enough (or too many) of these elements are available in the JGraph.
- *Checking the absolute positions of nodes:* All nodes in a JGraph have their absolute horizontal and vertical coordinates. CCM uses this information to check whether the nodes are arranged correctly (within a certain tolerance). Imagine a card game

where the cards have to be placed at special points of the workspace marked by a background image. The checker then checks all `Card` nodes on the screen and verifies their position. If a card is misplaced, CCM presents a message to the user.

- *Checking relative positions of nodes*: It is possible to express complex constraints about the relative arrangement of nodes. This feature e.g. allows checking a puzzle not only against an explicitly declared position, but also against *any* position on the screen. Also "inaccurate" declarations ("node A has to be somewhere *near* node B") are possible by defining a tolerance value.
- *Checking edges*: This mechanism examines if different nodes of the JGraph are connected to each other. It is also possible to distinguish between different edge types and it is possible to make restrictions concerning properties of the edge (e.g. an edge weight). We can for example use the edge check together with the existence check to describe the arrangement of transitions and places in a Petri net.

2.2. Extended Checking Capabilities

The checking conditions presented so far work with *all* nodes of a JGraph. They use features provided by the `java.awt.Component` class, which are inherited by all nodes of a JGraph. This way it is possible to examine some fundamental attributes of the JGraph's visual appearance. However, the possibilities offered by nodes that implement a special interface, named `Checkable`, are even more interesting. For nodes of this type, there is a flexible system to examine arbitrary user-defined properties. In contrast to the basic features, the extended features make it possible to check not only the *appearance*, but also aspects of the semantic structure encapsulated in a JGraph.

The `name` parameter is a concrete example for this extended feature. It is used by all nodes that implement the `Checkable` interface: Identifying nodes by using their Java class name, as it is possible by using the basic checking features, is too inaccurate for many applications, because it is impossible to distinct between different instances of the same class in this approach. Therefore the `Checkable` interface offers the possibility to name each node individually and, by this, to distinguish between instances of the same class if

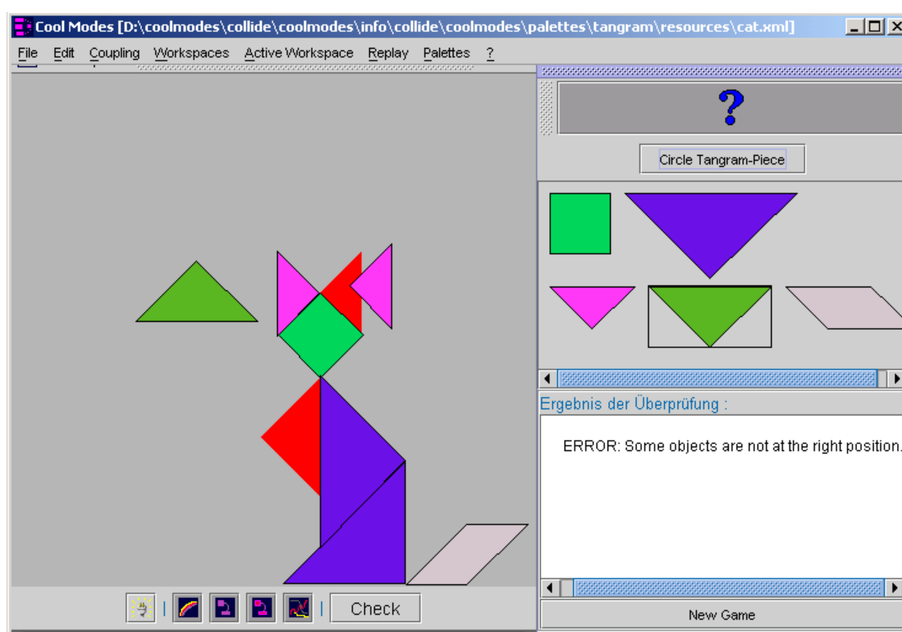

fig. 1: Screen shot showing the Tangram game for Cool Modes using CCM.

desired. The standard checking mechanisms are already designed to use this extended naming feature if a node implements `Checkable`.

2.3. Combining Single Checking Conditions to Create More Complex Structures

The conditions described in the previous sections can be combined to complex structures by Boolean operators. Additionally, we have included "fuzzy" conditions like "2 to 3 of the following 5 conditions have to be fulfilled".

In the Tangram game, which was implemented for Cool Modes (fig.1), a typical checking structure consists of three AND structures. The first one checks whether the correct pieces are *existent* on the workspace; the second one checks their relative position, the third one checks the turn angle of the pieces. These three AND structures are combined by a superior AND, because a solution is correct only if all needed pieces are existent *and* at the right position *and* placed with the right angle. Alternative solutions for a given Tangram puzzle, if applicable, would be combined by an OR structure at a higher level.

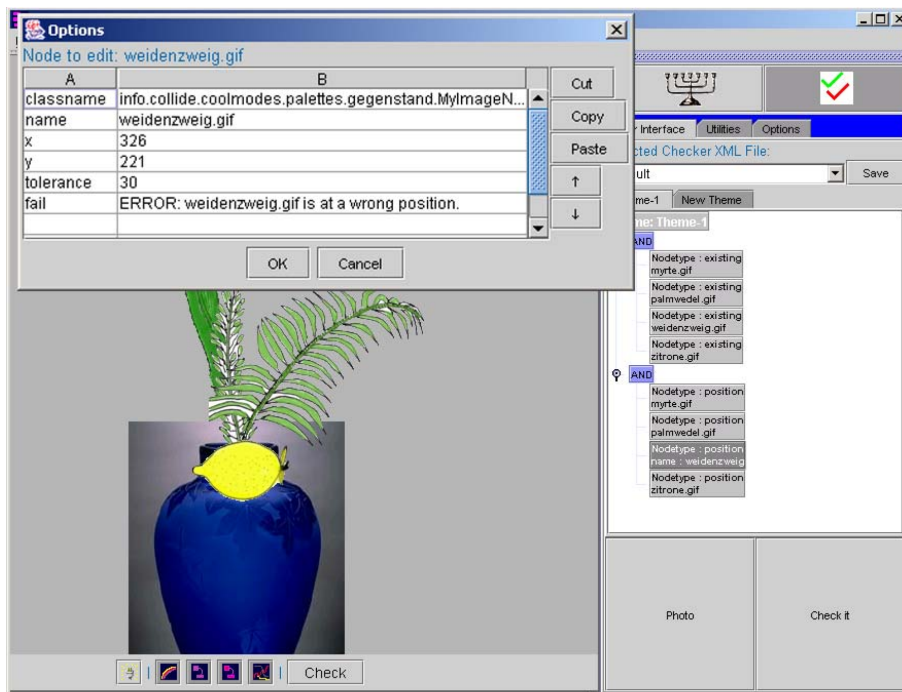

fig. 2: Cool Modes screen shot showing the checker UI on the right and an option dialog in the middle

3. An Example Application

The following example results from an interdisciplinary project at the University of Duisburg. In this project, material for the computer supported religious education at secondary schools was developed in cooperation with the department of Jewish studies. The specific domain was Jewish religious ceremonies. The task of "building a lulaw" described below is a simple use case for the checker. A lulaw is a bundle of four objects which have to be arranged in a special way. It plays a role in the celebration of Sukkoth.
A typical use case for the checker consists of three steps:

- *Constructing the task:* Before using the checker in an educational situation, the teacher must construct a solution for the task to be assigned to the students. He does

this in the same way students would do in the school lesson. In this simple example, constructing the solution only means choosing the four correct plants that belong to a lulaw and putting them into a vase on the workspace.

- *Configuring the checker:* In a second step, the teacher configures the checker by using a special tool. He "takes a photo" of the workspace, and the checker then automatically produces an XML representation of what the teacher has constructed before. Figure 2 shows the screen after pressing the "photo" button. On the right you can see the user interface of the checker tool, including the tree representing the solution and the "Photo-" and "Check It-" buttons. The tree in the middle of the user interface consists of two AND structures with four children each. The first AND structure describes the conditions for the existence test, while the other one contains the claimed positions for the position test. The small dialog window in the middle of the screen shows the configuration screen for one of the checking conditions, a position test in this case. Different settings can be edited here: the feedback message, the permitted tolerance from the exact position, and more.
  In this simple example, no further changes must be made, because we accept the standard messages the checker produces and the default tolerance for the position test. Also we do not forbid the presence of things not belonging to the lulaw on the screen, although this could be done for a real lesson preparation.
- *Using CCM in a classroom:* Now, during a lesson, the students can try to build the lulaw autonomously. CCM automatically produces hints in the case of errors and messages in the case of success. The students can work, either alone or in small groups, at their own speed. In some time intervals or after certain events (e.g. when a new picture is added to the workspace), the checker checks the workspace and returns feedback about the progress. Figure 3 shows three checker messages that might appear during this process: the first one says that two nodes are missing. The second one appears when the student has selected all plants correctly, but arranged two of them at the *wrong place*, and the third one shows the success message after correctly building the lulaw. The messages shown at the figure are the default messages of the checker. These standard messages are editable to get a more meaningful output. To do that, the teacher can use and change existing feedback templates and produce new ones. One of the template used in this example is `"ERROR: <objectname> does not exist"`, where `<objectname>` is

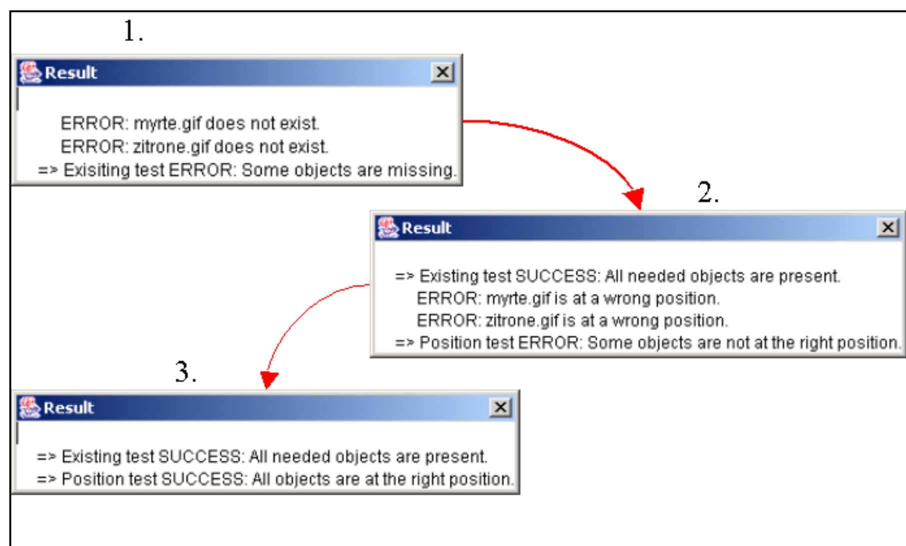

fig. 3: Cool Modes screen shot showing the results of some checks.

replaced by the correct identifier automatically. CCM also returns machine readable information about the checks it made and their results. This information can be used for further processing and analysis of the learning process.

With the intention of demonstrating the usage rather than the abilities of CCM, the given example shows only a small part of its possibilities. Only basic functions have been used, and even those have only been used partially.

4. Selected Implementation Details

Cool Modes and CCM are implemented in Java. The most important class of the checker package is the `CheckerModel` class. It manages the XML checking conditions and is needed both during the configuration of the checker, and during its use. On the one hand, in a preparation situation, CCM calls the CheckerModel's method `photo(JGraph g)` to produce a new structure of checking conditions. On the other hand, when the produced XML structure is used for checking, the CheckerModel calls its method `check(JGraph g, Theme t)`. This method returns a `CheckerResultSet`, an object that contains the result data of a check and, beyond that, the ability to visualize the result on the screen.

4.1. The Checkable Interface

To use the extended features of CCM, a node must implement the `Checkable` interface. This interface contains only one single method declaration:

```
public Hashtable getContent();
```

The method returns a Hashtable that contains all checkable attributes of a node, similar to the Property features supported by many Java classes. Each node that implements the `Checkable` interface should support at least the `name` attribute mentioned in section 2.2. The Hashtable entry for this attribute is the String *name* for the key and e.g. *myrthe.gif* for the value, taking a node from the lulaw scenario as an example, where the names of the image files are used as value. The list of checkable attributes is easily extensible. Including a new checking parameter only means adding a new parameter to the Hashtable. This does not affect already existing nodes and checker files, these simply ignore unknown attributes.

4.2. StructureProducer and NodeChecker

A `StructureProducer` is an instance of a corresponding Java interface and maps an object structure (currently always a JGraph) to an XML document that contains checking conditions. At present, four different `StructureProducer` classes are implemented:

- An existing-test builds a structure that contains one `<existing>` XML element for each node of a JGraph and combines the elements to an AND structure.
- A position-test works analogous to an existing-test but produces elements that deal with the *position* of nodes.
- A content-test checks user defined attributes accessible by the Checkable interface. This one is the most powerful and flexible StructureProducer.
- A Tangram-test ignores all nodes that are not part of the Tangram game. Analyzing the Tangram pieces, it builds a complex three-level structure described more precisely at the end of section 2.3.

The first three StructureProducers are domain independent, while the fourth one obviously only works with the Tangram game. It is optional because everything it does can be done

by the other three StructureProducers, too. But providing a specialized StructureProducer relieves the user from most of the configuration work. For the lulaw example in section 3, only the first two StructureProducers have been used.

For each XML element created by a StructureProducer, a corresponding NodeChecker is needed. While an XML structure is checked, the NodeChecker is called each time an appropriate tag is found. If a checking condition like `<existing name="myrthe.gif">` appears, the NodeChecker that handles these elements starts examining whether the JGraph contains a node with the name *myrthe.gif.* A NodeChecker returns an XML element that contains information about the result of the check.

StructureProducer and NodeChecker are independent of each other. The StructureProducer used for the Tangram game e.g. builds elements of the types `<existing>`, `<position>` and `<content>`. Therefore a special "Tangram NodeChecker" is not needed and does not exist.

4.3. The XML Format

The StructureProducers build XML documents which contain the checking conditions for a JGraph. The structure of these XML documents is described now.

Figure 4 shows a simplified fragment of the XML file created during the lulaw example. The top level element is the `<theme>` element. A `<theme>` represents the description of a whole task, e.g. the complete description of the lulaw (cf. 3). A theme essentially consists of one or more `<structure>` elements. These elements represent the conjunctions or disjunctions of checking conditions, which are children of a given structure. Additionally a structure can contain further `<structure>` elements as children.

The different Boolean operations are realized by the minimum and maximum attribute a structure must have. These two parameters determine how many of the child conditions of this structure must be fulfilled in order to return "success". Using these two parameters, AND (min=max=number of child nodes), OR (min=1; max=number of child nodes) and NOT (min=max=0) can be expressed as well as "fuzzy" checking conditions like "Of the following 5 conditions, 2 to 4 must be fulfilled".

On the lowest level, there are the tags that represent the atomic checking conditions as described in section 3. The fail and success attributes of this node describe the message the checker returns in case of a successful or unsuccessful check result.

```
...<theme>
    <structure minimum="all" maximum="all">
     <existing name="palmwedel.gif" classname="MyImageNode"/>
     <existing name="myrte.gif" classname="MyImageNode"/>
     <existing name="weidenzweig.gif" classname="MyImageNode"/>
     <existing name="lemon.gif" classname="MyImageNode" fail="lemon
             missing" success="Congratulations. Lemon is present."/>
    </structure>
   </theme> ...
```

fig.4: Fragment of the XML file constructed during the lulaw example

5. Discussion and Outlook

What remains to do?

- One of the major goals with the design of CCM was to make its use as simple as possible. It should be possible to use the checker without any knowledge of

programming languages or XML. This goal was not achieved completely. Configuring the checker still means a lot of work for the user who prepares a task. That should be changed in the future.

- Although CCM in its basic configuration uses geometric constraints, it can produce "semantic illusions", even without adding domain-specific check routines. Imagine, an additive number pyramid as it is used in primary school mathematics. If the task of the students is to fill holes in a given pyramid with number cards, then the geometric check will have the same effect as an arithmetic one. Thus, the checker can "behave like" if it could calculate. Developing further plug ins for Cool Modes that make use of this "pretend-as-if"-ability will be an interesting goal.
- At present, the result of a check condition is always a Boolean value. A more flexible rating system would be better in order to move from "right-or-wrong" messages to a more differentiated feedback.
- We are planning to integrate a user awareness component into CCM. In a collaborative environment, each action can usually be assigned to its producer. Using this information would allow a differentiated feedback, e.g. a special support for users who make more mistakes than others.
- Another topic on the agenda is the integration of a time component into CCM in the future. New checking possibilities like "A must happen before B" or "C must be done within 10 seconds" would be possible this way.

References

[1] Hot Potatoes 5.5. http://web.uvic.ca/hrd/halfbaked/index.htm

[2] Mohamad Jamil Sulaiman, Ng Kok Weng, Cher Dong Theng (2002). Intelligent CAD Checker For Building Plan Approval. In *International Council for Research and Innovation in Building and Construction, CIB w78 conference 2002.* Aarhus, Denmark.

[3] Pinkwart, N., Hoppe, U., Gaßner, K. (2001). Integration of Domain-specific elements into Visual Language Based Collaborative Environments. In Borges, M. R. S., Haake, J. M. & Hoppe, H.-U. (ed.), *Proceedings of 7th International Workshop on Groupware* (CRIWG 2001) (pp. 142-147). IEEE Computer Society: Los Alamitos, California.

[4] Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schütt, H. & Thüring, M. (1992). SEPIA: A coopoerative hypermedia authoring environment. In *Proceedings of the 4th ACM Conference on Hypertext*. (pp. 11-22). Milan (Italy).

[5] Suthers, D., Weiner, A., Connelly, J. & Paolucci, M. (1995). Belvedere: Engaging students in critical discussion of science and public policy issues. In Greer, J. (ed.), *Proceedings of the 9th World Conference on Artificial Intelligence in Education (pp. 266-273)*. Washington DC (USA).

[6] Joolingen, W. R. van (2000). Designing for collaborative discovery learning. In Gauthier, G., Frasson, C. & VanLehn, K. (eds.), *Proceedings of 5th International Conference on Intelligent Tutoring Systems* (pp. 202-211), Montréal (Canada). Berlin, Heidelberg: Springer.

[7] Mühlenbrock, M. & Hoppe, H.U. (2001). A collaboration monitor for shared workspaces. In J. D. Moore, C. L. Redfield, & W. L. Johnson (editors): *Proceedings of the International Conference on Artificial Intelligence in Education* AIED-2001, (pages 154-165). San Antonio, TX, May, Amsterdam: IOS Press.

[8] Constantino-González, M, Suthers, D.D. (2000). ACollaborative Learning Environment for Entitiy-Relationship Modeling. In G. Gauthier, C. Frasson, K. Van-Lehn (Eds.), *Intelligent Tutoring Systems, 5th International Conference*, Montéal, Canada. Berlin: Springer.