Programming with ASLT and Metainformation

Prof. Dr. Karl Hayo Siemsen

FachHochschule
Oldenburg/Ostfriesland/Wilhelmshaven
Fachbereich Technik, INK
Constantiaplatz 4
26723 Emden

De Montfort University,
Software Technology Research Laboratory

The Gateway
Leicester LE1 9BH, UK

ABSTRACT. The following paper is a short explanation and overview of the work of the team together with Mr. Wolke, Mr. Yermashov and Mr. Rasenack presented at the same conference. We do not prefer source code as the basic definition of an application. Instead we use the ASLT as the basic view. For a first understanding, this is source code parsed into a tree form and stored permanent. Changes on the application definition during development normally take place in the ASLT. Two converters, java2aslt and aslt2java produce the ASLT or the source code. A tree is a much simpler form to do automatic code insertion into an existing application. There are further views into the application. They can be used as input. They are synchronized with the ASLT basic view. The tree form can support a special form of "comments" called metainformation. This metainformation can be attached to each node forming a node of its own. So metainformation can be attached to a metainformation node producing nested metainformation. The concept is supported by tools to access the ASLT, for example to insert a node, to remove it, to change its contents, to validate the type or to evaluate the metainformation and process it, maybe during design time, may be during runtime. These tools are called metainformation processing tools (MIPTs).

1. Introduction

Sourcecode has the disadvantage of line numbering. If a line is inserted, all later line numbering marks are not valid. In the ASLT metainformation is used instead. Parts of the ASLT defined by metainformation can be brought

to source code form and vice versa.

Parts of source code at different places but logically coherent can be temporally sorted together by metainformation to change code (grouping and folding). The ASLT form in wide parts can be defined language independent. Metainformation fulfills functional and non functional requirements.

In figure 1 you see an ASLT in the middle, different views around the center. The synchronisation in Java is realized by events and listeners, no polling. Figure 2 shows an ASLT with metainformation attached to some nodes, and figure 3 shows a typical Java ASLT detail.

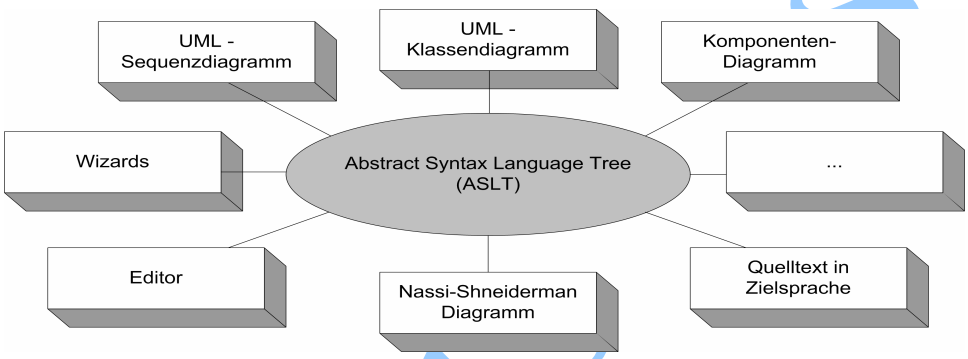

Fig. 1: *Synchonisation of the ASLT and other views*

To work with an ASLT, tools are needed to support the tree form. Different from Java, there is one ASLT file for an application, no split into source class files. If one variable is declared and that place is searched in the ASLT, all other uses of that variable are references (links). If the name of the variable has to be changed, search the place of the variable declaration once and change the content of that node, that's all. Other variables with the same name are out of scope and are not influenced. The UML class diagram is a special fold of the ASLT: only the classes, the "extends" and "implements" (arrows), methods, "dependencies" (arrows) and variables are unfolded. With a tool they can be shown in the usual visual presentation. Different parts of the ASLT can be unfolded by metainformation, all other parts remain folded. This form can be temporarily brought back to a view of the source code where pieces of the code are put together which are many lines away from each other in the normal view of the source code, may be in different files. The named unfolds are stored in a database. You can switch between different unfolds, the folding or grouping or slicing information is metainformation and not lost when switched to a different unfold. Figure 4 shows a grouping.

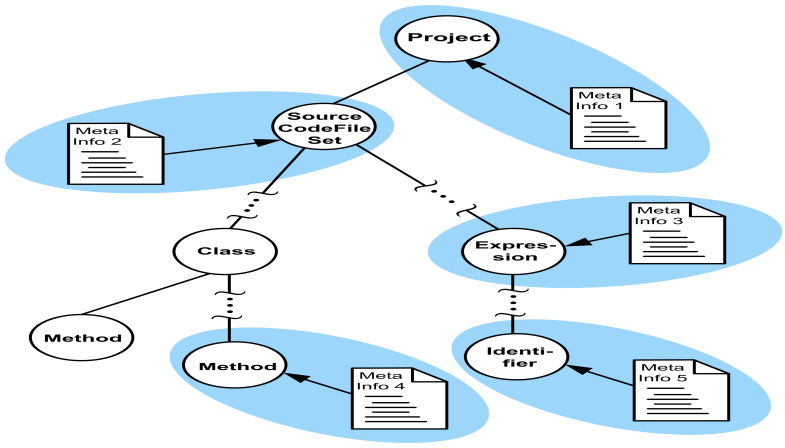

Fig. 2: *Schematic view of an ASLT*

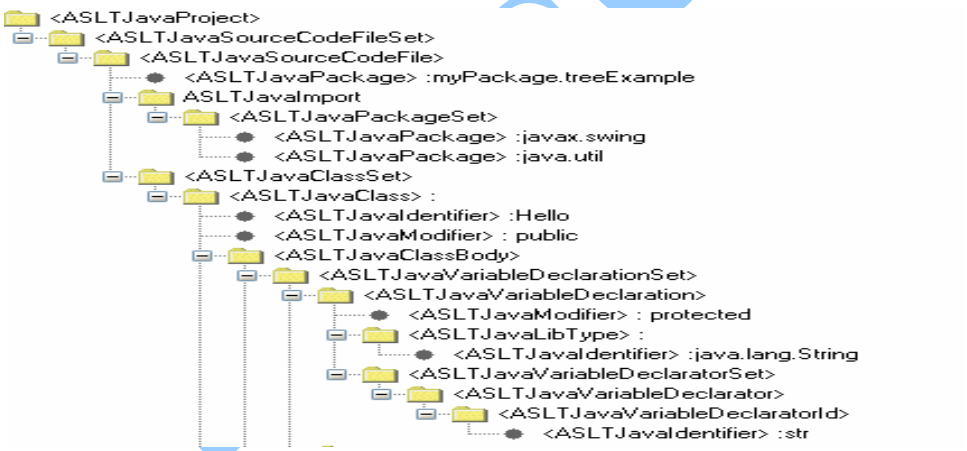

Fig. 3*: Detail of an ASLT (part of Hello World)*

The ASLT is handled by tools. For example a metainformation can specify a template, for example “search”. The template is taken from a library, inserted by a MIP tool and replaces the metainformation of the ASLT (but the metainformation is still held in the ASLT to be able to take a different search algorithm). This replacement allows source code substitution by metainformation. At any time the replacement can be folded to its metainformation placeholder.

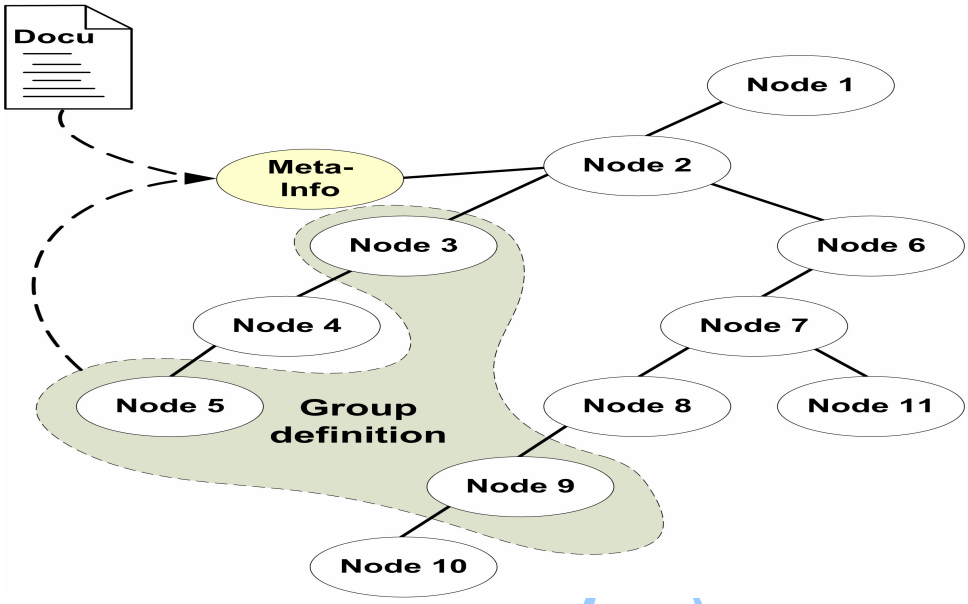

Fig. 4: *Grouping before folding and unfolding the group in an ASLT*

An ASLT can be made relatively independent from any special programming language. As shown with the template example, the language of the source code of a language can be extended by templates. For example, Java can be extended by occam PAR or a pipeline or a farm, templates not included in the Java language itself. This is the foundation of programming language conversion, for example from assembler to Java or from C to Java and vice versa. In general a third independent ASLT is needed as an internal step for language conversion.

Because the ASLT is language independent, most of the tools working with ASLTs have the same property of language independence. If they are built, they can be used in a different project in a different language.

The ASLT, metainformation and the tools described until here support the full language in use; there is no loss in generality.

2. Failover as a non functional requirement of an application

If there is a tested application you can add non functional requirements by metainformation. This is valid for other non functional requirements too, for example monitoring during runtime to an administration server. You should be able to add it as easy as possible. One of the examples is *failover*. First you insert metainformation half automatically by a tool, for example virtual

breakpoints. Call them failover points. You can do that for example after each user input of the application. Then every user input is stored in a database at its failover point. If the application crashes, the failover points with the user inputs can be called, especially the last executed failover point. At this point (somewhere in the middle of the code) the application can be restarted. This is nearly invisible to the user. All his information – for example in a basket - is valid and taken from a database, no new input from the user is needed even if the connection or the server or the application had crashed. This basic concept can be extended if the failover points are not only located after user inputs but after longer calculations as well or if the data structures to be stored are more then strings from the user but objects. At the failover points, which means after some metainformation, the non-functional code is inserted automatically by a MIP tool. Only the placement of the failover points is done by the developer. Because the metainformation is not necessarily transferred to the source code, the previous original source code can be reproduced at any time if needed to modify that code. The modification can be brought back to the ASLT without disturbing the failover metainformation inserted earlier. Until this explanation there is no restriction in generality as there is with the following example.

3. Visual domain specific programming, an example

There is the typical situation: you have non-programmers as users and they have to program a specific task they want to use.

The contradiction can be solved: let the users program visually with a *set of icons along a toolbar* and by using *drag and drop* into a pane. They drag the icon, drop it on the pane and – maybe - have to connect that icon with a former one, for example for communication. The icons are programmed by experts, the user only uses them in an intuitive way. The programming (generating the source code of the application and compiling it to a running application) is hidden to the user.

We call this view of programming the Neurath view. Otto Neurath was the inventor of icons called Isotypes. Again the way of programming is done via ASLT and metainformation. Domain specific means: we concentrate on the pre-produced icons of the toolbar and insofar we are restricted to those “programming” elements, to that domain.

Earlier there was a hint to Occam constructs within the Java language. If in the toolbar there are all 35 or so Java language elements extended by the Occam constructs, this is not really "domain specific". All what can be done by a general language like Java and some extras can be done. Basically this is a Nassi-Shneiderman form of Java with some Occam extensions. What we have in mind with priority are domain specific examples where no knowledge of programming is needed and drag and drop of the icons of the toolbar is what has to be done during visual programming.

The example we want to discuss is a specific domain with inputs (sensors), controls (macros) and outputs (actors), the connections between these elements in a relation layer as a blueprint of an application and finally the instantiation to a task. During discussion during the conference we found other interesting examples, a Petri Net composed by a non-programmer or a supply chain (A. Roth, A. Fortis).

The elements input (sensors), controller (macros) and output (actors) are within a toolbar ready for drag and drop. They are put on the pane. Grouping and folding of some basic macro elements or of some sensors or actors should be possible at the pane at any time. These folded elements (elements with combined characteristics) can be brought back to the toolbar and reused. With the last step the instantiation the reusable elements are fixed to one single application. Earlier, for the view we used as a graphic tool the package *Jung*, but we are just changing to the open source graphics tool *Jgraph* able to group, to fold and to support different layout managers. Please notice: the graphics *view* is separated from *model* and *controller* by the MVC design pattern. So the view is exchangeable.

The elements of the toolbar can be combined by the user to a running task. To produce the relation layer definitions there are docking places in the ASLT part trees. The relation layer combines the elements not only locally but - if needed - remote. A second layer can insert security keys to the transmission line if the information is transmitted over the internet. Connections not valid are checked during iconic design and rejected. Again the toolbar elements are pre-programmed and in ASLT form. There are docking places to combine them.

The application is done by drag and drop of ASLT part trees, the instantiation and compilation are not visible to the user.

The icons themselves can be extended to contain some extra information. For example a sensor can contain a thermometer scale and some values. The sequence of the following figures 5 to 8 will show you the editors with their panes for building sensors, macros and instantiated tasks.

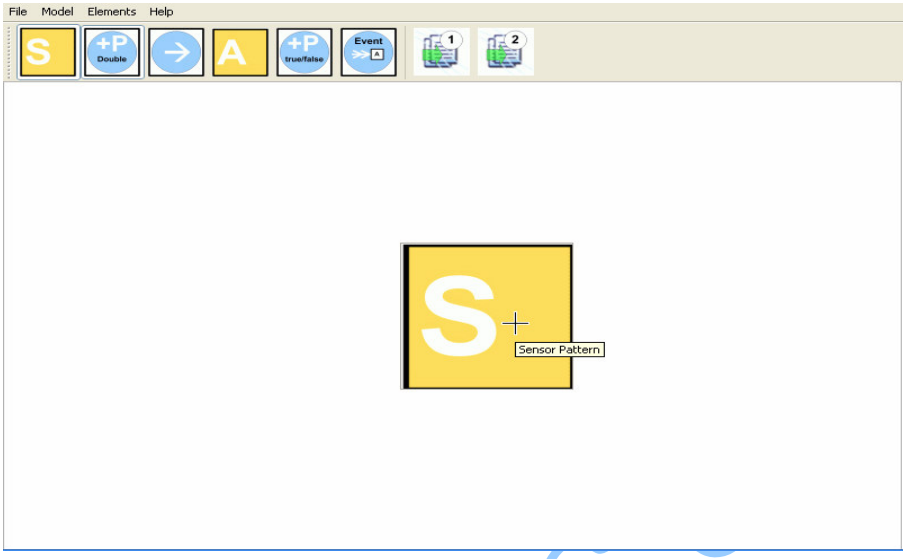

Fig. 5: *The sensor editor creating an extra sensor pattern (start)*

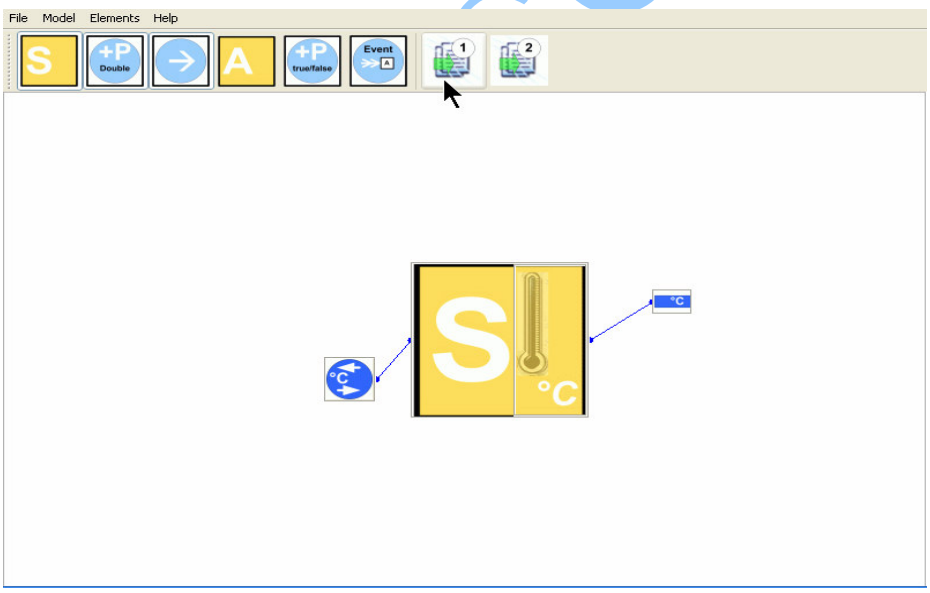

Fig. 6: *The sensor editor creating an extra sensor pattern (design)*

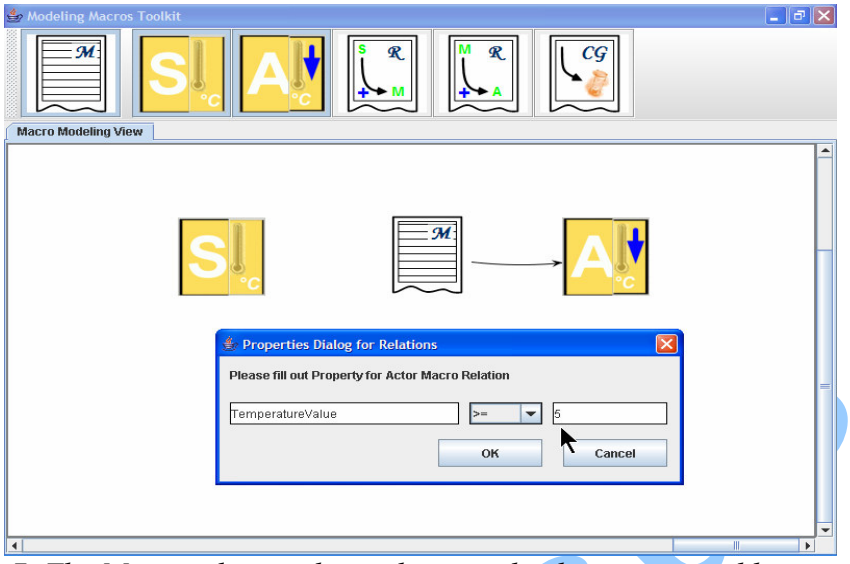

Fig. 7: *The Macro editor, relation layer and values supported by a wizard*

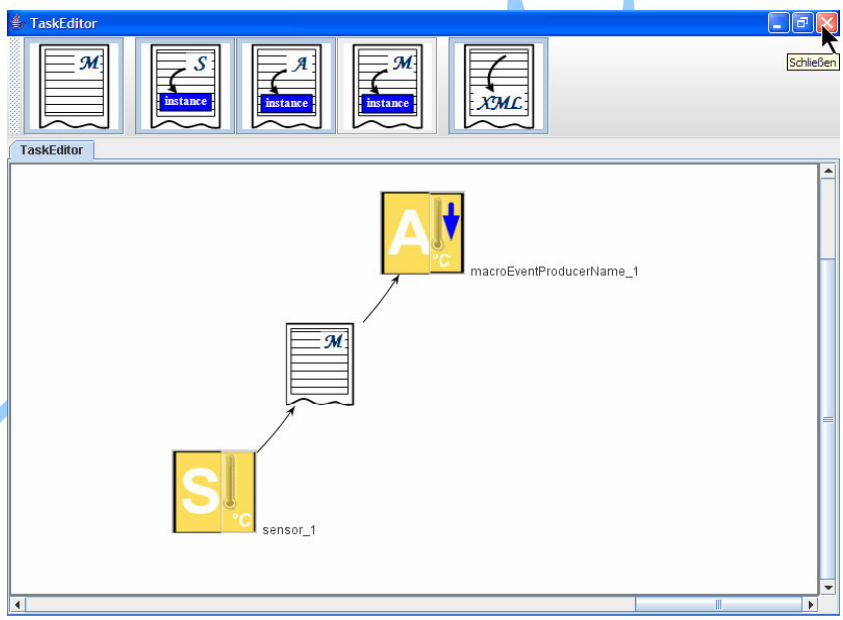

Fig. 8: *The task editor producing an application including instantiation*

The basic elements build are brought to the toolbar of the next editor for further use.