Implementing Hierarchical Graph-Structures*

Josef Tapken**

University of Oldenburg, Germany

Abstract. We present concepts for the implementation of hierarchical graphs, which can be used as basis for the implementation of tools for graphical formal description techniques (gFDT) like SDL or statecharts. Our approach provides a strong modularity of a specification by a loose coupling between different hierarchy levels and it serves for a rapid development of interactive editors for gFDTs by a special technique of describing hierarchy. Furthermore, this technique allows the reuse of graph editors in different applications. Our concepts are explained by means of the graphical design tool MOBY/PLC for a special class of real-time automata, called PLC-Automata.

1 Introduction

Application of formal methods is necessary in the development of correct distributed systems, e.g. in the area of telecommunication, and particularly for safety-critical systems like railway-crossings or traffic control.

Very important for the acceptance of formal methods is the availability of tool support. The description techniques used in the tool should be as simplified as possible in order to minimize the time for a system designer to get familiar with that tool.

Graphical representations of formal description techniques (FDT) are useful and increase their acceptance. Examples for practical used FDTs with graphical representations are statecharts (cf. [7]) or SDL (**S**pecification and **D**escription **L**anguage, see [9]). A commercial design tool for statecharts is e.g. STATEMATE (cf. [8]) and for SDL the SDT-tool (cf. [14]).

In this paper we present concepts for the development of tools for graphical FDTs (gFDTs). In doing so we refer to a design tool for PLC-Automata (cf. [4]), a special class of real-time automata which is tailored towards a certain kind of hardware called PLC (**P**rogrammable **L**ogic **C**ontrollers). This tool is called MOBY/PLC (cf. [12], [3]) and is part of the MOBY[1]-workbench which additionally provides Petri-net based validation methods like simulation and model-checking for SDL-specifications (cf. [5]). In order to show the generality of the concepts we also give a simple SDL-example.

* This research was partially supported by the Leibniz Programme of the Deutsche Forschungsgemeinschaft (DFG) under grant No. Ol 98/1-1.

** e-mail: tapken@informatik.uni-oldenburg.de

[1] **MO**delling of distri**B**uted s**Y**stems

All gFDTs have one problem in common: they loose their readability if the described system becomes very complex. Hence many gFDTs are enriched by several structuring concepts like abstract data types, orthogonality (parallelism) and/or hierarchy (abstraction). In this paper we will concentrate on the handling of hierarchy and exploit the fact that the concept of hierarchy is independent of the special gFDT under consideration. Following this line the object-oriented implementation of a hierarchical gFDT should be based on (derived from) a generic implementation of hierarchical graph structures, instead of extending each implementation of a gFDT by a special hierarchy concept.

In this sense we present the integration of hierarchical graphs into the Moby Class Library (MCL[2]) and show its applicability with respect to the design and implementation of the PLC-tool.

The paper is organized as follows: In section 2 we show an example of PLC-Automata in order to give an idea of the hierarchy concepts used in this class of automata. The core of the paper consists of a formalization and implementation of hierarchical graphs (section 3) and a special technique of describing hierarchy in a system comprising several types of graphs (e.g. like SDL-specifications, which comprise block, process and procedure specifications, section 4). Section 5 summarizes this work and gives some outlook to further work.

2 PLC-Automata

In [4] a new class of real-time automata was introduced which is tailored for modelling the behaviour of PLCs that are often used in practice to solve real-time problems. The automata are provided with a formal semantics in *Duration Calculus* (DC, [2]) to allow formal reasoning about their properties.

PLCs provide two convenient concepts for implementing real-time systems, namely timers and an automatic polling mechanism. The latter concept is realised by the cyclic behaviour of PLCs, whereat each cycle consists of the following phases:

- Poll all input channels and store the read values,
- compute new output values based on the stored input values, and
- update the output channels.

The only phase a programmer has to adapt is the computing phase, whereas the repeated execution of the cycle and the channel handling is managed by the operating system. Via input and output channels a PLC can be connected to sensors and actuators as well as to other PLCs.

Since PLC-Automata are closely related to the behaviour of PLCs, a specification using PLC-Automata consists of a set of parallel automata communicating through channels with each other. Communication is performed implicitly

[2] cf. [1]; MCL is a C++-class library which contains the application independent kernel of the Moby-workbench. Among others, it provides classes for building Motif based interactive graph editors.

by putting the calculated output values on the output channels, which can be read via an input channel by another automaton. Open systems can be specified by connecting some input and output channels to the interface of the system, which is graphically represented by a set of *ports*.

The parallel composition of the set of automata is called *system view* and is specified in one (or more) *system diagrams*. For the structured description of the system view some ideas of SDL are adopted, especially the notion of blocks and block substructures (cf. [9]). Therefore, a system diagram contains *system nodes* (blocks) which can be specified (refined) by another system diagram or by a description of its behaviour by means of a PLC-Automaton. A PLC-Automaton may also be structured hierarchically by using PLC-Subautomata.

As a running example we adopt the case study described in [4] which handles the gasburner control problem (a case study of the ProCoS-project, cf. [10]) using PLC-Automata. The case study specifies the control unit for a gasburner which gets two Boolean inputs, hr ("heat request") representing the state of a thermostat changed by the user and fl ("flame") indicating the status of a sensor monitoring the flame. The output of the specified system is a Boolean value controlling the gas valve. One real-time requirement for the system is that in every period of 30 seconds gas must not leak for more than 4 seconds.

Fig. 1 shows a screen shot of the system view of the specification. In order to give an idea of the parallel composition of PLC-Automata we divide the automaton described in [4] into two, one for handling the input values and the other one for calculating the output.

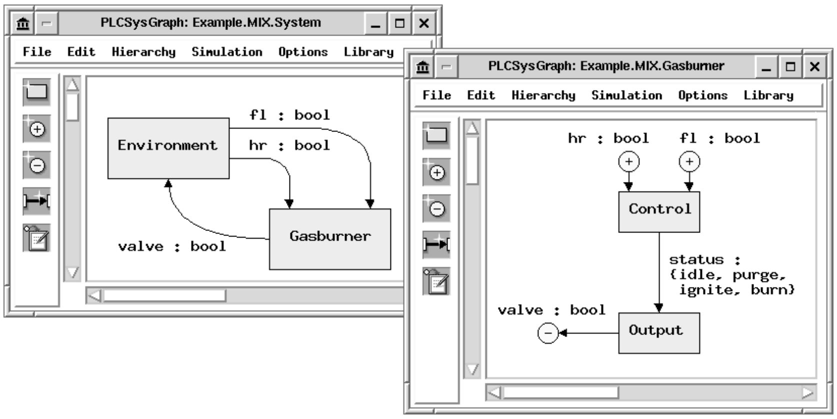

Fig. 1. System view of the gasburner specification

The system view is split into two diagrams. The left diagram shows the environment of the specified system consisting of one not further specified system node Environment, which delivers two input values for the gasburner subsystem (hr and fl) of type bool and receives one output value valve of type bool.

The refinement of the Gasburner node is shown in the right diagram. Thereby the interface of the system node consisting of incoming and outgoing channels is adopted from the left diagram and represented as ports ($\oplus$ and $\ominus$) in the refinement. Furthermore, the refinement contains two system nodes (Control and Output) which are connected by a channel called status. This channel is inscribed with an enumeration type ({idle, purge, ignite, burn}) which contains the output values of the Control system.

The behaviour of the two system components Control and Output is specified by two PLC-Automata (see Fig. 2). The description of the Control-automaton is split into two diagrams in order to show the hierarchy concept for PLC-Automata by using a subautomaton SubControl. (There is no conceptual reason for the seperation.) Each PLC-Automaton contains an interface description consisting of a set of *text symbols* which carry the declaration of input and output variables. (In this paper we do not handle local variables.) These variables have to correspond with channels specified in the upper diagram of the system view.

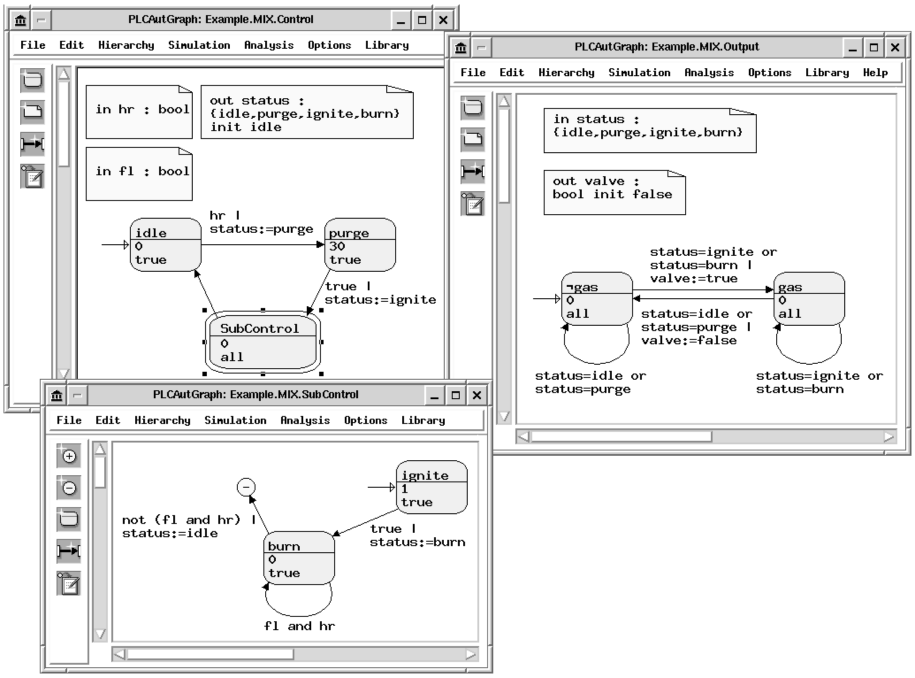

Fig. 2. PLC-Automata of the gasburner specification

A PLC-state is inscribed by its name and a pair (time-value, state-condition). state-condition is a predicate over the input variables and expresses that all inputs which fulfill state-condition should be ignored for time-value time units after entering the state. E.g. state purge of the automaton Control has the state condition true and the time value 30. The transition function is labelled by predicates which are built over all variables of the automaton and by lists of assignments to the output variables.

The operational behaviour of a PLC-Automaton is as follows: The PLC-Automaton checks in every cycle if the actual state is hold longer than time-value or if the state condition evaluates to false under the current binding of the variables, if it does the automaton reacts according to the transition function and all assignments of the transition are executed. (Thereby a transition can be taken if its predicate evaluates to true.) Else the state will be held for another cycle.

Fig. 2 shows that the Control-automaton starts in the state idle and remains there until hr is true. Then the automaton turns to purge and stays there several cycles until 30 seconds are elapsed. After that the initial state (ignite) of the subautomaton SubControl will be activated and held for at least one second. Then the automaton will change to burn and stay there as long as both the flame sensor (fl) and the thermostat (hr) provide the value true. If e.g. the ignition of the gasburner has failed (fl=false) then the subautomaton will be left via its output port and the state idle will be reached. The Output-automaton receives the status from Control and calculates the output value for the gas valve. The valve will be opened (valve:=true) if status is ignite or burn and closed otherwise.

3 Hierarchical Graphs

In this section we present and discuss the data structures (classes) for hierarchical graphs which we implement in MCL.

3.1 Formalization

Hierarchical graph structures like the previously presented diagrams are often described as *compound (directed) graphs* (see e.g. [11]). The definition of hierarchical PLC-Automata (cf. [3]) is also based upon compound graphs. A compound graph $C = (G, T)$ consists of a simple directed graph $G = (V \cup S, E_G)$ describing the transition relation and a tree $T = (V \cup S, E_T)$ representing a nesting relation between the different nodes. V contains the leafs of T which are called *base nodes* and S contains the inner nodes called *subgraphs.*

Fig. 3 shows the compound graph of the running example neglecting the different types of the graphs (system diagram, PLC-Automaton and PLC-Subautomaton).

Compound graphs are very compact descriptions of hierarchical structures and therefore useful for the definition of the semantics of hierarchical specifications. But they are not tailored to be used directly as base structure for (the

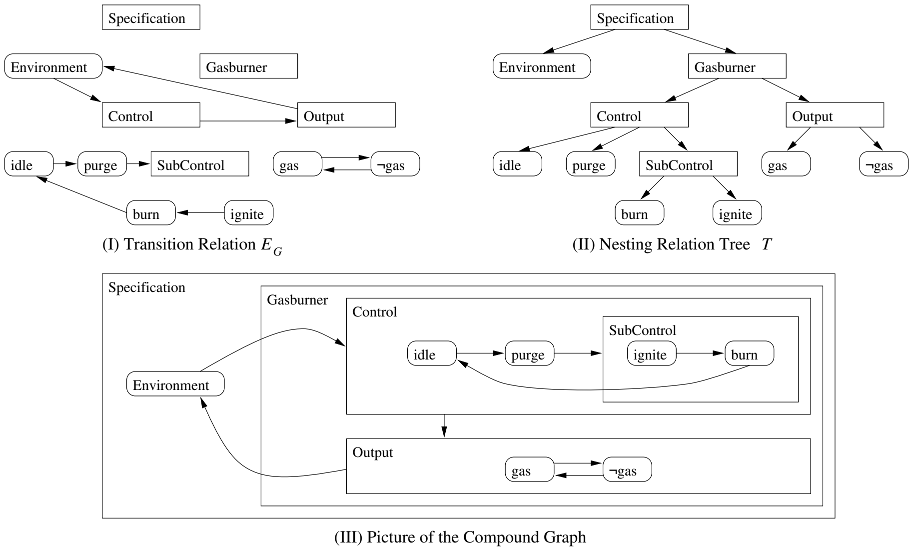

Fig. 3. Compound graph of the gasburner specification

implementation of) a gFDT, because a gFDT has to provide methods for dividing specifications into modules of manageable size in order to allow the user to focus upon certain modules while neglecting others and/or to reuse certain modules in different specifications. In SDL this aim is pursued by the notion of *remote* specifications, i.e. the user separates a part (module) of the specification from its defining context and refers to this part just by using a unique name (*reference*, e.g. see [9]). In compound graphs this compositionality requirement is violated by the fact that they allow inter-level edges, i.e. edges between nodes of different levels.[3]

For the implementation of hierarchy in MCL we use a different structure, called $\mathcal{H}$*-graphs*. $\mathcal{H}$-graphs arise from compound graphs by introducing a special class of nodes called *ports*[4] which describe explicitly the interface of a subgraph. Thus inter-level edges are substituted by a list of one-level edges between ordinary nodes and ports. For example the inter-level edge from burn to idle in Fig. 3 is divided into one edge from burn to a newly introduced output port and one from the subgraph SubControl to idle (cf. Fig. 4).

Newly introduced edges connected to a subgraph differ from ordinary edges in the sense that they have exactly one corresponding port inside the subgraph which represents the virtual source (resp. destination) of the edge. The distinction between different types of edges is known from statecharts (cf. [7]) where

[3] Two different nodes of a compound graph are in the same level if they have the same predecessor in the nesting relation tree.

[4] Since we consider directed graphs we distinguish input and output ports.

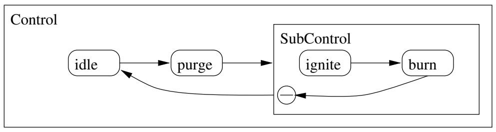

Fig. 4. Part of the previous compound graph

edges are allowed to lead *to* as well as *into* a statechart (resp. *from* and *outof*). But statecharts do not consider ports explicitly, they just allow dangling edges which have no source (or no destination).

The elimination of inter-level edges results in a modular data structure for hierarchical graphs called $\mathcal{H}$-graph. Fig. 5 sketches this structure by means of the gasburner specification. An $\mathcal{H}$-graph contains two types of graphs, namely *interface graphs*, which serve for the description of one compound graph level, and a *design graph*[5] which composes the different levels, i.e. the design graph covers the information of the nesting relation tree.

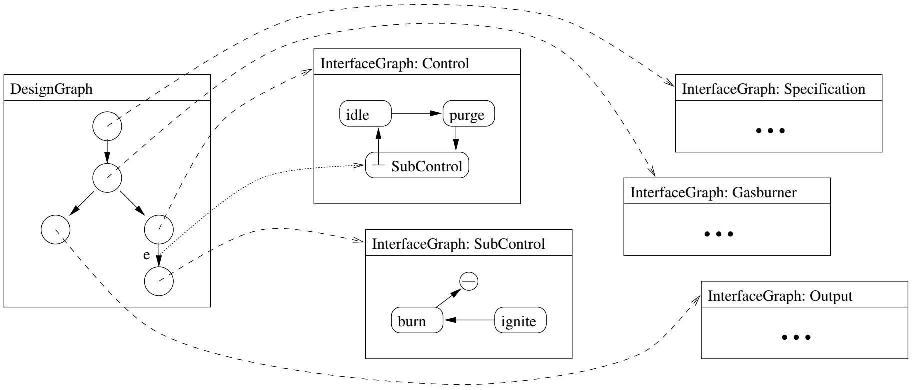

Fig. 5. Data structure of the gasburner specification

The idea of the design graph is that each node represents one interface graph and that each edge represents a refinement of a vertex in the 'source graph' by the 'destination graph'. Therefore, the edge is equipped with an extra link to that vertex being refined.

In the following we formalize the data structure of $\mathcal{H}$-graphs. We first define the class of interface graphs. An *interface graph* is a tuple $g = (V(g) \cup P_I(g) \cup P_O(g), E(g))$ with

– $V(g)$ is a set of nodes,

[5] Design graphs arising from compound graphs are actually trees.

– $P_I(g)$ and $P_O(g)$ are sets of input and output ports (building the interface of g),
– $E(g) \subseteq (V(g) \cup P_I(g)) \times (V(g) \cup P_O(g))$ is a set of edges labelled by a function $\lambda : E(g) \longrightarrow \mathcal{P}(\{i, o\})$.

The labelling function determines the status of an edge which can be normal ($\emptyset$), dangling-in (i), dangling-out (o) or dangling-in/out ($\{i, o\}$). Additionally, we postulate that edges do not dangle into (resp. out of) a port, i.e.

– $\forall (n, p) \in (V(g) \cup P_I(g)) \times P_O(g) \bullet i \notin \lambda(n, p)$, and
– $\forall (p, n) \in P_I(g) \times (V(g) \cup P_O(g)) \bullet o \notin \lambda(p, n)$.

The interface of a node v in g is described by its dangling edges, thus for $v \in V(g)$ we define

– $D_I(v) = \{(n, v) \in E(g) \mid i \in \lambda(n, v)\}$, and
– $D_O(v) = \{(v, n) \in E(g) \mid o \in \lambda(v, n)\}$.

Fig. 5 shows five interface graphs, whereat e.g. the interface graph SubControl contains two nodes (ignite and burn), one output port and two normal edges.

For the composition of interface graphs we define the structure of $\mathcal{H}$-graphs. An *$\mathcal{H}$-graph* is a tuple $H = (d, \mathcal{G}, \rho)$ with

– $d = (V(d), E(d))$ is a directed acyclic graph, called *design graph*,
– $\mathcal{G}$ is a set of interface graphs,
– $\rho = (\rho_V, \rho_E)$ is a pair of functions with
  • $\rho_V : V(d) \longrightarrow \mathcal{G}$ is a bijection between the vertices of d and the set of interface graphs, and
  • $\rho_E : E(d) \longrightarrow V$ is a refinement function, with $V = \bigcup_{g \in \mathcal{G}} V(g)$ and the restriction $\rho_E((v_1, v_2)) = v \Longrightarrow v \in V(\rho_V(v_1))$.

The bijection ρ_V associates to each vertex in the design graph an interface graph. The edges of the design graph together with the refinement function ρ_E realise the connection between the different hierarchy levels. An edge (v_1, v_2) in the design graph, e.g. the edge e = (Control,SubControl) in Fig. 5, represents the information that the vertex $v = \rho_E((v_1, v_2))$ (SubControl) in graph $\rho_V(v_1)$ (Control) is refined by the graph $\rho_V(v_2)$ (SubControl).

Since both objects, the vertex v and the refinement $\rho_V(v_2)$, provide their own interface description, we have to check their consistency. Thus, we call an $\mathcal{H}$-graph H *consistent* if for all $(v_1, v_2) \in E(d)$ holds

– $\exists$ bijection $\beta_I : D_I(v) \longrightarrow P_I(\rho_V(v_2))$, and
– $\exists$ bijection $\beta_O : D_O(v) \longrightarrow P_O(\rho_V(v_2))$

with $v = \rho_E((v_1, v_2))$.

For the edge e this requirement holds since the interface graph SubControl has only one output port and the corresponding node has one dangling-out edge.

3.2 Implementation

The implementation of $\mathcal{H}$-graphs is based upon an object oriented implementation of graphs. Both substructures of $\mathcal{H}$-graphs, the interface graph and the design graph, are derived from simple graphs. Real applications like MOBY/PLC are based upon $\mathcal{H}$-graphs simply by deriving concret graphs-like system diagrams from the abstract interface graph.

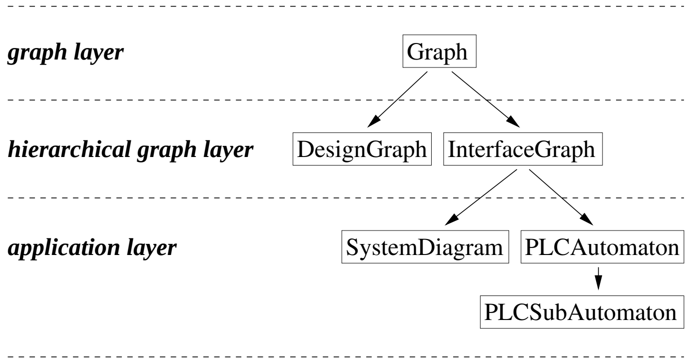

Fig. 6. Layered architecture of MOBY/PLC

This leads to a layered architecture for a concret application which is sketched in Fig. 6. Each box represents a package of classes which build up together the intended data structure.

Fig. 7 shows the main classes of the different packages. Each rounded box describes one class. The name and the superclass is stated in the top line. The members of a class are listed in the lower part of the box.

In MCL a graph is represented by a Graph-object which contains a collection of Vertex-objects. Each vertex stores a reference to its graph and two lists of its incomming and outgoing edges, which are represented by the class Edge. Each edge knows its source and destination.

The data structure of a design graph is build up by the classes DesignGraph, DesignVertex, and DesignEdge, which are derived from the appropiate classes of the graph package. A DesignVertex is enriched by a reference to an InterfaceGraph and a DesignEdge by a reference to a vertex of an interface graph.

Interface graphs distinguish two classes of vertices, namely Ports and refinable vertices (IF_Vertex). A Port carries a flag whether it is an input or an output port. The graph object itself holds a reference to its representing DesignVertex and the edges of an interface graph (IF_Edge) may have two references to ports. These references represent the dangling information of an edge. If there is no reference stored, the edge is a normal one, else a reference links the edge to a port in the refinement graph of the connected vertex.

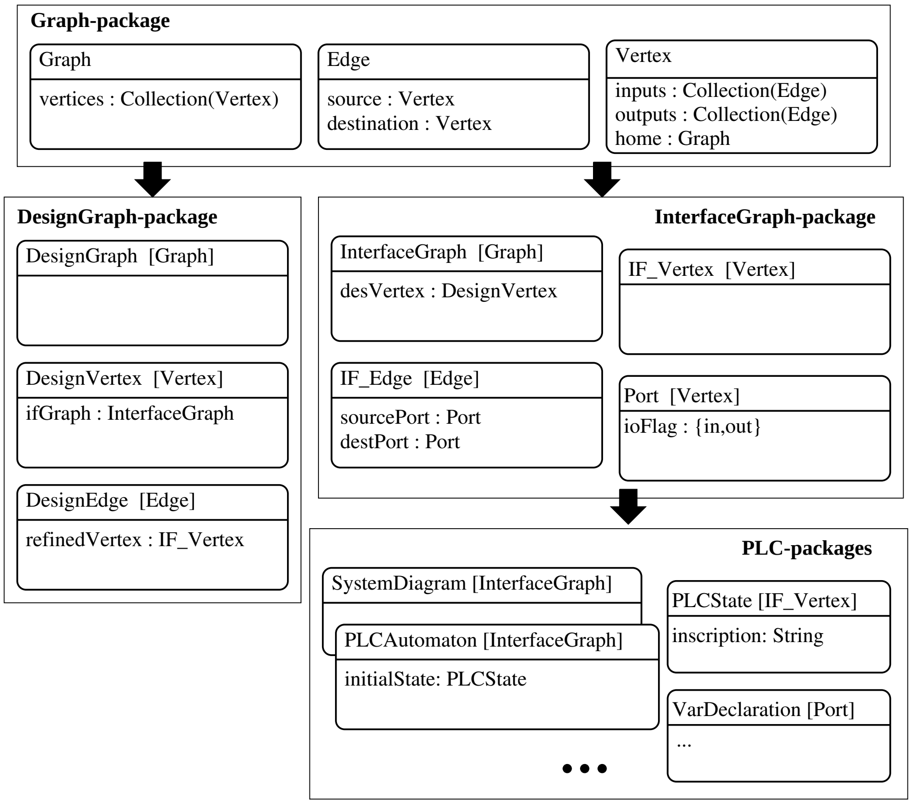

Fig. 7. Main classes of the $\mathcal{H}$-graph data structure in MOBY/PLC

The extensions which are needed for the MOBY/PLC-application are only sketched in Fig. 7. E.g. a PLC-Automaton stores its initial state and the states carry some additional inscriptions. Variables are special ports since they describe the interface of a PLC-Automaton.

3.3 Discussion

In this subsection we discuss the main properties of the presented hierarchy concept and show its application to an SDL-specification.

The data structure for hierarchical specifications which we propose provides the modularity of the specification by a very loose coupling between the different hierarchy levels. Thus each module consisting of one interface graph can be edited independently, in particular several users can simultaneously work on the same specification. Additionally, this concept facilitates the reuse of certain modules in different specifications.

The design graph provides directly an assistance for traversing the whole specification, i.e. in order to edit a leaf of the hierarchy tree the user can access the node in the design graph instead of browsing through the whole specification.

Furthermore, the concept of $\mathcal{H}$-graphs allows us to combine several types of interface graphs in one specification. This feature is necessary e.g. for the running example (see section 2) where we need system diagrams for the description of the system architecture and we need two kinds of automata diagrams for specifying the behaviour of the different components.

It is also useful for a tool which allows to describe distinct parts of one specification in different languages, provided that they have a consistent semantics. This can be relevant for big projects where one part can be adopted from a former project specified in language A (e.g. SDL) and the new part should be specified in language B (e.g. statecharts). One approach to combine semantically several specification languages is elaborated in [6].

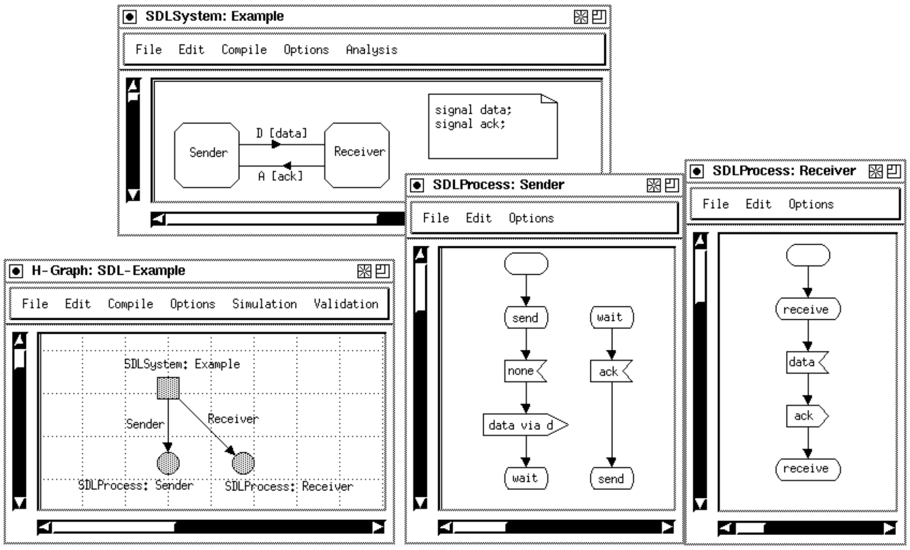

Fig. 8. SDL-specification in MOBY

At the end of this section we give a short SDL-example which shows the applicability of our hierarchy concepts for other gFDTs than PLC-Automata. Fig. 8 shows a screen shot of an SDL-specification in MOBY. It presents a (very) simple communication protocol where the Sender process sends data packages via the channel D to the Receiver and gets the acknowledgement via the channel A. Fig. 8 contains in the lower left corner the design graph with three nodes, which represent the SDL-diagrams on the top and on the right side, namely the block diagram Example and the two process diagrams Sender and Receiver. These types of SDL-diagrams are derived from interface graphs and extended by e.g. channels in block diagrams and several kinds of nodes, like input and output symbols in process diagrams.

4 Descriptions of Generic Hierarchical Graphs

In the previous section we have shown that the concept of $\mathcal{H}$-graphs is suitable to combine several types of interface graphs into one specification. New types can be defined by deriving them from the class of interface graphs. But for the combination of different interface graphs there are some additional informations necessary because not all refinement relations between interface graphs make sense. E.g., in the gasburner specification PLC-states should only be refined into a PLC-Subautomaton and not into a system diagram, whereas a system node could be refined into a system diagram as well as into a PLC-Automaton.

The additional refinement information could be handled by the interface graph itself, but then it is not possible to use one special class of interface graphs in several contexts, e.g. to use system diagrams for the description of system architectures in an SDL-tool as well as in a PLC-tool. On the other hand, if the information is directly implemented in the design graph each application has to define its own design graph.

Thus, it is advisable to use some features of generic programming to describe the refinement capabilities in a hierarchical application.[6] The description of the refinement capabilities is based upon the set of all types of interface graphs GT and the set of all types of nodes VT which are defined in an application. In the PLC-example the graph types
$GT = \{\mathsf{SystemDiagram}, \mathsf{PLCAutomaton}, \mathsf{PLCSubAutomaton}\}$ and the node types $VT = \{\mathsf{SystemNode}, \mathsf{PLCState}\}$ are used.

A *refinement description* is a tuple $(root, ins, ref)$ with

- $root \in GT$ is the type of the initial graph in a specification,
- $ins : GT \longrightarrow \mathcal{P}(VT)$ is an *insertion function* describing what kind of nodes can be inserted into a graph, and
- $ref : VT \longrightarrow \mathcal{P}(GT)$ is a *refinement function* which determines what refinements are allowed for a node.

For a concrete application the $\mathcal{H}$-graph is instantiated with a special refinement description. In our PLC-example this will be

- $root = \mathsf{SystemDiagram}$,

[6] C++ directly supports generic programming by templates, but we avoided the use of templates in MCL in order to enhance portability, because some compilers do not handle templates in combination with extensive inheritance correctly. Instead, we reach the facilities of generic programming by a special class Class which provides runtime class information and allows to create instances of a dynamically determined class.

– $ins(\mathsf{SystemDiagram}) = \{\mathsf{SystemNode}\}$,
$ins(\mathsf{PLCAutomaton}) = \{\mathsf{PLCState}\}$,
$ins(\mathsf{PLCSubAutomaton}) = \{\mathsf{PLCState}\}$, and
– $ref(\mathsf{SystemNode}) = \{\mathsf{SystemDiagram}, \mathsf{PLCAutomaton}\}$,
$ref(\mathsf{PLCState}) = \{\mathsf{PLCSubAutomaton}\}$.

This refinement description states that the specification will start up with a system diagram where only system nodes can be inserted in. A system node can be refined again into a system diagram or into a PLC-Automaton. A PLC-Automaton and a PLC-Subautomaton can only contain PLC-states which on their part can be refined into PLC-Subautomata.

In this refinement description we do not treat ports. There are two ways to handle ports in our concept, first they can be directly associated to an interface graph (no generic description for ports) or they can be treated as a special kind of nodes, then they have to be added to the insertion function.

In MCL the refinement description is used as a parameter for the graphical user interface of a hierarchical editor system, i.e. the buttons and menu entries for creating and refining vertices are automatically generated out of the refinement description. E.g. the contents of the vertical button bars shown in Fig. 1 and 2 are directly dependent on the refinement description of the application. Since the refinement description is evaluated at runtime it is possible to change it during a session by means of a graphical editor.

5 Conclusion

In this paper we have shown a flexible concept for implementing hierarchical graphs which is tailored to build a common basis for several implementations of gFDTs.

For the development of a new gFDT-tool the programmer only has to extend the data structure by deriving new classes of interface graphs, nodes and if necessary ports and edges, but (s)he has not to implement the hierarchical relations between these classes, (s)he just defines the refinement description in a graphical editor. In order to change the appearance of the tool the programmer only has to slightly modify the graphical user interface by redefining some methods of the new classes. E.g. (s)he has to add some menu entries to the graph editor or has to overwrite the display methods of a node.

The 'real' functionality of the new tool, like simulation algorithms or compile algorithms to other representations (e.g. from SDL-specifications into Petri nets), can be added to the chosen data structure independently. This can be implemented directly in the new classes or by defining 'service'-classes which operate on the graph structure.

In the Moby-workbench we successfully applied the concept of $\mathcal{H}$-graphs to implement several graphical design tools. We are currently working on a tool for the validation of SDL-specification, on a hierarchical editor for (Object-)Z-specifications (each basic node contains one Z-schema and the hierarchy is used for structuring the specification), and on the PLC-tool we mentioned in this paper. This tool allows to generate runnable PLC-code out of an automaton specification and is enriched by several validation methods like a simulator for PLC-Automata (cf. [13]), the implementation of specific analysis algorithms for PLC-Automata and a connection to a model checker for timed automata. Furthermore, we are planing to integrate the Z-editor for the description of abstract data types in PLC-Automata.

Acknowledgements

The author thanks H. Fleischhack, E.-R. Olderog, and the other members of the "semantics group" in Oldenburg for fruitful discussions on the subject of this paper.

References

1. P. Amthor, N. Arndt, H. Fleischhack, and J. Tapken. The Moby Class Library. Technical report, University of Oldenburg, 1998. in preparation.
2. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Inform. Proc. Letters*, 40/5:269–276, 1991.
3. H. Dierks and J. Tapken. Tool-Supported Hierarchical Design of Distributed Real-Time Systems. In *Euromicro Workshop on Real Time Systems*, pages 222–229. IEEE, 1998.
4. Henning Dierks. PLC-Automata: A New Class of Implementable Real-Time Automata. In *ARTS'97*, LNCS. Springer Verlag, May 1997.
5. Hans Fleischhack and Josef Tapken. An M-Net Semantics for a Real-Time Extension of μSDL. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *LNCS*, pages 162–181. Springer Verlag, 1997.
6. Bernd Grahlmann. Combining Finite Automata, Parallel Programs and SDL Using Petri Nets. In *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 102–117. Springer Verlag, 1998.
7. David Harel. On Visual Formalisms. In *Communications of the ACM*, number 5 in 31, pages 514–530, 1988.
8. i-Logix. Languages of STATEMATE, January 1991.
9. ITU-T Recommendation Z.100. *Specification and Description Language*. ITU General Secretariat - Sales Section, Geneva, 1992.
10. A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and Verifying Requirements of Real-Time Systems. *IEEE Transactions on Software Engineering*, 19:41–55, January 1993.
11. Georg Sander. Layout of Compound Directed Graphs. Technical Report A/03/96, FB Informatik, Universität des Saarlandes, 1996.
12. J. Tapken. MOBY/PLC – A Design Tool for Hierarchical Real-Time Automata. In *Proceedings of FASE'98*, volume 1382 of *LNCS*, pages 326–329. Springer Verlag, 1998.
13. Josef Tapken. Interactive and Compilative Simulation of PLC-Automata. In W. Hahn and A. Lehmann, editors, *Simulation in Industry, ESS'97*, pages 552 – 556. SCS, 1997.
14. Telelogic AB. SDT – the SDL Design Tool, http://www.telelogic.se/.