Component Tools: A vision for a tool

Alexander Gepting, Joel Greenyer, Ekkart Kindler, Andreas Maas, Sebastian Munkelt, Csaba Páles, Thorsten Pivl, Oliver Rohe, Vladimir Rubin, Markus Sanders, Andreas Scholand, Christian Wagner, and Robert Wagner

University of Paderborn, Department of Computer Science
`pg-components@upb.de`

Abstract. Petri nets are a powerful technique for modelling, analysing and verifying dynamic aspects of all kinds of systems. And so are many other techniques from the field of formal methods, e. g. State Charts or timed automata.
All of these techniques have their particular strengths and weaknesses, which often make it necessary to have models in different notations for the very same system. Or it might be necessary to have one abstract model and another more detailed model of the same system for performing different analysis tasks. The problem, however, is that the different techniques come with their own composition mechanisms, which make it very expensive to use and to maintain different models of the same system.
In this paper, we discuss some concepts that deal with the above problems. The basic idea is to define component types with their interfaces and their possible connections completely independently from any particular modelling technique. For each component type, a set of possible models in different notations and on different levels of abstractions can be defined. From these component definitions, which we call a *component library*, an engineer can build a system and can fully automatically generate an overall model. Then, this overall model can be analysed or verified with the techniques provided for this particular technique. The generation of the overall model will be defined by triple graph grammar (TGG) rules, which will be interpreted by a TGG-interpreter.

1 Introduction

Petri nets as well as many other techniques from formal methods are a powerful means for modelling, analysing, validating, and verifying all kinds of systems. Each of these techniques has its particular strength and its particular weakness. They are useful for some purposes, but they may fail with respect to another purpose. Therefore, it is necessary to apply different techniques to modelling the same system for different purposes and possibly even combine different techniques. It might even be useful to have different models in the same notation, but on different levels of abstraction.

Moreover, system engineers would like to construct systems from some components in a way that is independent from the specific composition mechanism provided by the underlying formal method. And they would like to switch between different aspects or views of their system. Still, they would like to use the power of different techniques, once they have constructed their systems or subsystems.

In this paper, we present the concepts of a tool that will help to solve these problems, where some parts of the tool have already been implemented as a prototype [1]. The rest of it will be implemented in a course called project group[1] at the University of Paderborn. The tool will be called ComponentTools since it will support many different concepts to build a system from components, to transform these models, to export these models to analysis tools, and to import analysis results.

The basic idea of ComponentTools is a mechanism for defining *component libraries* from which systems can be built, where each component defined in the library can be equipped with different models from different modelling techniques. These models of the components, however, are not directly visible to the engineer. Once the engineer has built a system from the available components, ComponentTools provides a mechanism to build system models in the different notations that will be used for analysis and verification purposes. To this end, ComponentTools provides a general mechanism for defining such transformations, which will be based on *triple graph grammars* (TGGs) [4].

Originally, the Component Tools project was inspired by the modelling and verification of flexible manufacturing systems. Therefore, we will use a simplified example from this area for explaining the basic concepts of ComponentTools. The concepts, however, are much more general and we are pretty sure that there are more application areas where ComponentTools can be applied.

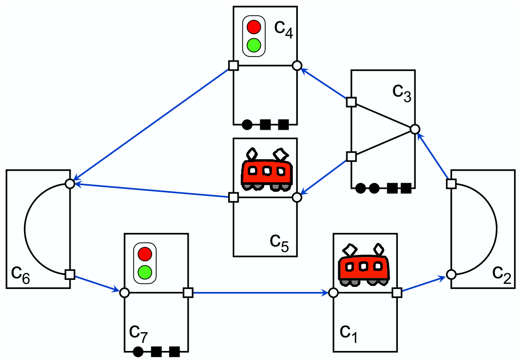

Fig. 1. A toy train built from components

[1] The authors of this papers are the supervisors and the participants of this course. See `http://wwwcs.upb.de/cs/kindler/Lehre/SS04/PG/` for more details.

2 Component Libraries

In this section, we discuss the idea of component libraries by means of a simple example. We have chosen a very simple toy train system, resembling a project in flexible manufacturing systems.

Components, Ports, and Connections Figure 1 shows a simple toy train system built from components. In this example, there are four different *component types*: straight tracks, curved tracks, tracks with a stop signal, and switches[2]. Each component type has some *ports*, which are graphically represented as small boxes and circles at the border of the components, and which can be used for connecting the different components. In our particular example, we have ports that represent physical connections of tracks: the connections from white squares to white circles. Moreover, there are ports for controlling the switches and the signals, which are indicated as black circles and squares; these would correspond to electrical wires attached to these components. But we did not connect the toy train to a controller in our example.

In order to build such a system, we must provide a *component library*: A component library defines all the available ports, which typically correspond to mechanical, electrical, or logical plugs for connecting different devices. The component library defines which types of ports are available as well as their graphical appearance. Moreover, the component library defines the component types. For each component type, it defines the ports and the position of these ports. Furthermore, the component library defines how ports may be connected, which is called a *connection paradigm*. There may be different types of connections, for example for mechanical connections, for pneumatic pipes, for electrical wires (of different voltages), or for signal lines. The connection paradigm defines all available connections and their graphical appearance; and the connection paradigm defines which connections may be attached to which ports as well as the fan-in and fan-out of the corresponding ports. In addition, a component type can be equipped with some parameters, which must be provided for each instance of this type. In our example, the speed of the train on a particular track could be such a parameter.

In addition to all this formal information, a component library provides informal or textual information on all defined elements. There are textual descriptions for the meaning and intended use of the defined ports, components and connections. These descriptions could refer to the terminology of this particular application area. For example, it could describe the plugs and the voltages for some electrical wires, or it could give the diameters and the maximal pressure of some pressure pipes.

Once a component library has been defined, ComponentTools can be used for building a system from these components; moreover, it guarantees that the user provides the required parameters for each component instance and that

[2] For simplicity, we have splitter switches only. The joining switches are built by connecting two tracks to a single one.

the connections meet the requirements of the connection paradigm. This part of ComponentTools is already implemented [1] and running, though some nice features are still missing[3].

Models So far, ComponentTools allows us to build a system from components, and the tool guarantees that components and connections are used in a syntactically correct way. But, how about analysis, verification, and validation of the system?

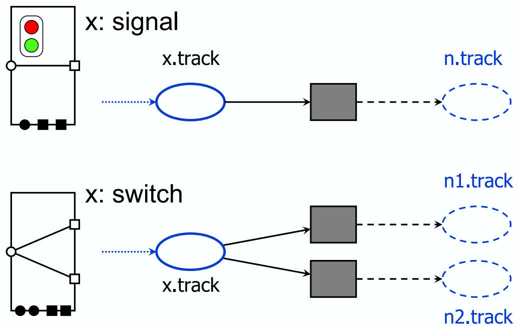

Fig. 2. Two models for components

In order to deal with these issues, each component type of the component library can be equipped with a model that defines its behaviour. Figure 2 shows the Petri net models for two of the components. In fact, we can provide even more models for each component. For example, there can be very abstract models like the ones shown in Fig. 2, and there can be more detailed ones. Or there can be additional models in different notation such as State Charts [2] or other notations. From these models and the system built by the user, ComponentTools can generate one or several overall models of the system, which can then be analysed by the appropriate tool for this formalism.

Model Generation and Transformation There are many different ways how to build an overall model from the different models of the component. ComponentTools itself does not know any of them. The idea is that these transformations are defined by the component library itself.

An easy way to define a translation would be to implement a program that performs this translations. The disadvantage of this approach, however, is that making available a component library would also require some programming. Moreover, in many cases we need not only a generation of an overall model, but

[3] Right now, components cannot be rotated and there are no user defined components. But, these features can be easily added.

we need to translate analysis results back to a component system in order to visualize these results.

Therefore, we decided that these translations and transformations should be defined on a higher level of abstraction. We chose to use *triple graph grammar* (TGGs) [4] for this purpose. The component library will include one TGG for each type of translation, and ComponentTools will interpret these TGGs for performing the translations among the different models (see [3] for more details).

Altogether, ComponentTools with a given component library allows an engineer to construct a system and to analyse it without even knowing the underlying techniques. The informal descriptions of all ports, connections, and components should give enough information to use the component library.

3 Additional Features

In the previous sections, we have discussed the basic concepts of ComponentTools. To complete the picture, we briefly discuss some other important features.

Constraints For some components, there are dynamic constraints for their use. The switch in our toy train, for example, has the constraint that its direction must not be changed when there is a train on it, because the train could derail in this situation. In order to keep track of such dynamic constraints, each definition of a component type in a component library can be equipped with such *constraints*, which could be a temporal logic formula or a property formalized in some other notation. Again, ComponentTools has no fixed notation for constraints. The interpretation of a constraint will be defined by translation rules similar to the translation of the models, in such a way that some analysis tool or model checker can check whether the constraints are met for all component instances, once the system is constructed.

Again, there will be an informal description of each constraint, so that an engineer knows their meaning without knowing the notation of the constraint.

Views When building flexible manufacturing systems, different users with different background and interest are involved. Some are more interested in the mechanical parts and connections, other are more interested in the electrical wiring, and others might be more interested in the control part. To this end, we introduce the concept of *views*, which allows us to edit different aspects of the system independently from each other. In our example, there could be a physical or geometrical view for editing the layout of the tracks; in another view a user could add the electrical wiring, etc. In the component library, we can define a set of independent views. Each view basically consists of a set of ports, connections, and components that are visible in this view, and it consists of a subset of these components and connections that may be edited in this view.

There could be even different kinds of editors for the views. For the geometrical view, there could be an editor where one-to-one-connections are represented by placing the corresponding ports at the same position. Figure 3 shows an example of such a geometrical view, where the ports of the right circle have not yet 'snapped'. In the original 'wiring' view from Fig. 1, we would see all these connections, but they would not be editable, in order not to introduce inconsistencies with the geometrical view.

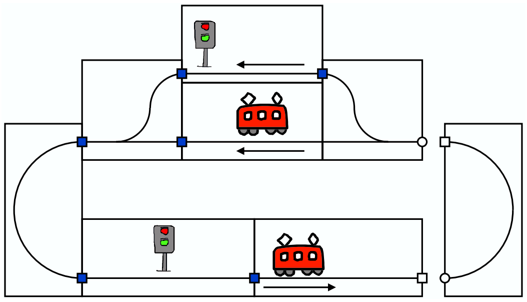

Fig. 3. A toy train built from components

4 Conclusion

The main idea of the ComponentTools concept is a general mechanism for the definition of component libraries in different application areas. ComponentTools will support the construction of systems from these libraries, and it will provide a mechanism for defining different views for the different involved users. In particular, it will not be necessary that a user knows the underlying formal methods, still these models will play an essential role in the analysis and verification or even controller synthesis for such systems.

References

1. Joel Greenyer. Maintaining and using component libraries for the design of material flow systems: Concept and prototypical implementation. Bachelor thesis, Department of Computer Science, University of Paderborn, October 2003.
2. David Harel. Statecharts: A visual formalism for computer systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
3. Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An adaptable TGG interpreter for in-memory model transformation. Unpublished manuscript, July 2004.
4. Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, Proceedings*, *Lecture Notes in Computer Science* 903, pages 151–163, June 1995.