Component Tools: Application and Integration of Formal Methods

Ekkart Kindler, Vladimir Rubin, and Robert Wagner

Department of Computer Science, University of Paderborn, Germany [kindler|vroubine|wagner]@upb.de

Abstract. The field of *formal methods* provides all kinds of powerful techniques for the specification, design, verification, validation, and start-up of systems. Unfortunately, the different techniques have different underlying formalisms and notations, they use different concepts and methods, and they are supported by different and, in many cases, incompatible tools. Therefore, most applications of formal methods are restricted to one technique or formalism – though using several techniques in combination would have many benefits.
The *Component Tools* project aims at easing the application and the integration of different formal methods with different underlying formalisms, notations and tools for some particular application area. To this end, Component Tools supports the definition of components with different underlying formal models for different purposes and a set of transformations for different tools. Then, an engineer can use these components for designing, verifying, and validating a system with support from formal methods and their tools under a uniform visual user interface – without even knowing the details of the underlying formal methods.
In this paper, we outline the basic idea, the concepts, and the main ingredients of Component Tools by using a simplified example from the area of flexible manufacturing systems.

1 Introduction

The field of *formal methods* provides quite powerful techniques for the specification, design, implementation, verification, validation, and start-up of all kinds of systems. In order to take full advantage of formal methods, a system engineer must apply different techniques from the field of formal methods, which, typically, use different formalisms, notations, and tools – let alone different principles and philosophies. A single formalism and tool will help in one stage of the development process or with respect to one problem or design task, but cannot deal with others. Since engineers are, typically, not experts in all necessary techniques of formal methods, they cannot switch back and forth between different formalisms. Therefore, formal methods are often abandoned at all – in spite of their potential.

In order to improve this situation, *Component Tools* supports the definition of components for a particular application area along with a set of models and

external tools supporting suitable formal methods. This, basically, defines a tool with a uniform visual user interface for constructing systems in the chosen application area that supports the chosen formal methods without knowing the details of the underlying methods. We call such a tool a *component tool*. The basic idea is to define *components* that are equipped with *ports*. Different *instances* of such components can be *connected* at these ports. This way, an engineer can construct a system such as a manufacturing system from these components. Along with each component definition, there will be one or more models of the behaviour of this component, which might use different notations and formalisms and can be on different levels of abstraction or can cover different aspects of the behaviour of the component. In addition, there will be *transformations* that define how the models of the different instances are combined into an overall model in order to apply a particular formal method. On this model, an *external tool* for the particular formal method can be started and the results of that tool will be transformed back to the visual representation of Component Tools. In order to support the bidirectional transformation, we use *triple graph grammars* (TGGs) for defining this transformations [1].

In this paper, we will discuss the basic idea and the concepts of Component Tools by the help of a simple toy-train example, which is introduced in Sect. 2. This is a simplified version of an application from the area of flexible manufacturing systems. The definition of components, the *component library*, will be discussed in Sect. 3, and the transformations as well as the integration of tools supporting a particular design or development *task* will be discussed in Sect. 4.

2 The Example

In this section, we briefly introduce our toy-train example, which resembles the application area of flexible manufacturing systems. It will serve as a running example, and we will use it for explaining the purpose and the use of Component Tools. But, Component Tools is much more general and can be used for many other application areas – actually its focus is the definition of component libraries and tools for new application areas.

2.1 The construction plan

Figure 1 shows a simple construction plan of our toy-train example. The plan consists of four different kinds of *component*: *straight tracks* (c_1, c_5), *curved tracks* (c_2, c_7), *switches* (c_3, c_6), and *signal units* (c_4, c_8). These components are combined by *connections*, which are indicated by arcs between the corresponding *ports*. There are one-to-one connections from white square ports to white circle ports, which represent the mechanical joints of the tracks of the toy train.

In our toy-train example, we assume that the trains are running all the time and can be stopped only at the signal units by switching the signals to stop. This setup is inspired by a mono-rail system used for flexible manufacturing systems. In order to control the route of the trains, the switch components and the signal

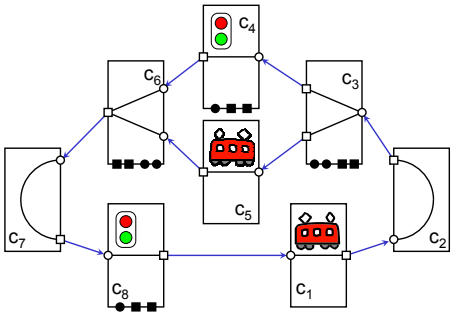

Fig. 1. Toy-train: Construction plan

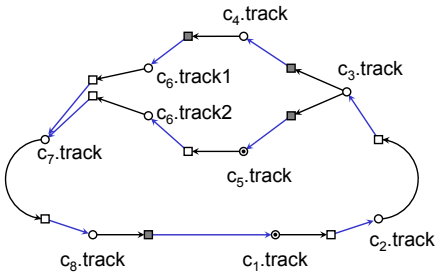

Fig. 2. A generated model

units are equipped with addition ports for changing the direction of the switch and for changing the signal states between stop and go. These ports are shown as black squares at the corresponding components. These ports correspond to electric plugs, which can be connected to some controller, which is not shown in Fig. 1. Moreover, there are *sensors* for sensing the current direction of the switch and for sensing the presence of a train at some signal unit. The ports corresponding to these sensors are represented as black circles.

2.2 The models

Now, let us assume that an engineer has finished a construction plan. From this plan, the engineer could easily build the real hardware of the system – called the *plant* in flexible manufacturing systems. And due to the simple components, he can also get a good understanding of the overall behaviour of the system. The purpose of Component Tools, however, is to support the engineer in some of his engineering tasks. In our example, these tasks are the design of a *controller* for the plant, the validation and verification of the controller, and to really build the system, implement the controller and to ramp-up the system, which, basically, means identifying and replacing faulty hardware (fault-diagnosis).

In order to support these tasks, we need some models defining the components and, in particular, the dynamic behaviour of the components. Component Tools is completely independent from any formalism, notation or syntax and any formal method. In our example, we use Petri nets for a very simple model of the dynamic behaviour of the components. Figure 3 shows the simple behaviour models for the signal unit and the switch units, where the shaded squares indicate Petri net transitions that can be controlled by an attached controller. With these models, we can simulate the system and by applying some formal methods, we could synthesise a simple controller for our toy train system.

Actually, there can be many more models for each component which serve different purposes and can be used in different tasks during the development, implementation, and ramp-up process of the plant and the controller. For example, there could be models that define the physical shape of the trains, the tracks

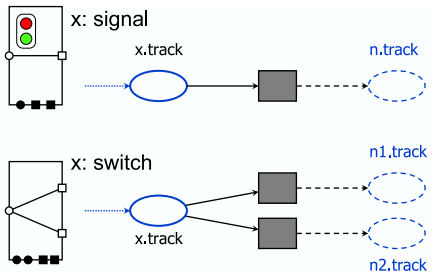

Fig. 3. Components: Simple models

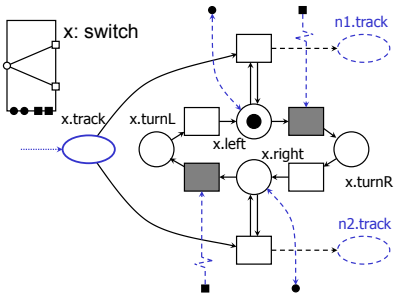

Fig. 4. Components: Detailed model

and slightly extended Petri net models in order to animate the behaviour in a 3D-visualisation. This simulation and visualisation could help in the validation of the controller. Another example are more detailed models of the behaviour that cover all kinds of intermediate states so that a more detailed controller can be designed avoiding all kinds of illegal intermediate states. An example of such a more detailed Petri net model of a switch is shown in Fig. 4.

In order to support fault diagnosis during the ramp-up process of a new plant, there can be even more detailed models that also cover possible faults in the hardware components. These fault-models can be used by some techniques from model checking for detecting hardware faults from observed failure behaviour.

In this paper, we will not go into the details of the underlying formal methods and the used formalisms and notations that are employed for the different development tasks. The crucial point, here, is that there are different models on different levels of abstraction in, possibly, different formalisms and notations, which support different design tasks.

2.3 The tasks and transformations

During the design and implementation of a new plant and controller, there are many different tasks that could be supported by some technique from the field of formal methods in different stages of the development process. Component Tools supports the definition of such *tasks* and the integration of tools supporting these tasks. To this end, the different models of the components will be used and transformed into formalisms that are understood by some tool supporting this formal method, and the results can be transformed back to the construction plan in order to visualise it to the engineer.

For example, from the simple Petri net models shown in Fig. 3, similar models for the other components, and the construction plan from Fig. 1, we could generate the Petri net model shown in Fig. 2. Note that these transformations are not programmed, but are defined in a declarative way by triple graph grammars, which will be discussed in more detail in Sect. 4. The main benefit is the back-and-forth translation, which easily allows results obtained by one method to be used by others. For example, a counter-example, i. e. an execution that

shows that a particular property is not valid, obtained by some model checker, could be passed to a tool that animates this execution in a 3D-environment. This way, an engineer virtually sees what goes wrong. In combination with a fault diagnosis tool, this could be also used to visualise the possible hardware faults in a 3D-model of the plant.

3 Component Libraries

Before Component Tools can be used in a specific application area, it needs some input. The most important input is the definition of a *component library*; the other is the definition of the *tasks*. In this section, we will discuss the component library and the different concepts that need to be defined. The definition of tasks will be discussed in Sect. 4.

Clearly, the main constituents of a component library are the definitions of different *components*. A construction plan is built from *instances* of these components which are joined via *connections* at the *ports*.

3.1 Ports

Since the same port type can occur in different components, ports need to be defined first. A component library defines any number of ports each of which corresponds to a physical, electrical or mechanical jack or joint. Note that, actually, the defined ports are *port types* – but we do not dwell on this issue here.

In addition to a unique *name*, each port type defines its particular *appearance*, i. e. its shape, line colour and fill colour. In our example, we used white circles and squares for the mechanical connections of tracks, and we used black squares and circles for electrical jacks. Moreover, each definition of a port type has a *description* of this particular jack or joint that informally defines its function and purpose. For example, a description could refer to a technical specification.

Clearly, the information on the appearance will be exploited when editing and displaying the components. The description of a port type can be used as a tool tip when the cursor is over such a port in order to help the engineer understand the purpose of this particular port type.

3.2 Connections

In a component library, there may be any number of *connections* resp. *connection types*. Again, each connection type has a unique *name* and an *appearance*, which defines the arrow heads and the colour of the arc, when connecting two ports. Again, there will be a *description* for this particular connection type.

More importantly, a component library comprises a *connection paradigm*, which defines how ports may be connected by particular connections and in which direction the connections may run. Moreover, the connection paradigm can define fan-in and fan-outs of connections in order to restrict the number of

incoming and outgoing connections at some ports. In particular, the connection paradigm can restrict connections at ports to one-to-one connections.

Component Tools will use the definition of the connection paradigm in order to guarantee that the construction plan never violates any structural restrictions implied by the components in the particular application domain.

3.3 Components

The definition of *components* resp. *component types* is the most important part of a component library. Each component type definition consists of a unique *name*, a definition of its *appearance*, i. e. its size, fill and line colour, and a *description* of its purpose and behaviour.

Moreover, each component has a list of *port definitions*. Each port definition, again, consists of a unique *name*, a reference to a port type, a *description*, and a *position* of this particular port. Note that it is not necessary to define the possible connections to a port of a component because this is already defined for the port types by the connection paradigm.

In each component type definition, there can be a list of *parameter definitions*, which, again, consists of a *name* and some data type defining the range of values of this parameter. In our example, the component type straight track has a parameter "number", which defines the number of trains that are on that particular track. Moreover, there are parameters defining the exact position of the start and end points of the track.

When a new instance of a component type is created in a construction plan, the values for all the parameters of this component type must be provided.

3.4 Models

In order to apply formal methods, each component is equipped with one or more models defining the behaviour or other aspects of the component (which, possibly, depends on the values of the parameters of the particular instance).

In order to deal with models, a component library defines a set of model types that are associated with each component type. Basically, this is a finite list of *names* equipped with a reference to some *format* in which these models are represented. And, again, there is a *description* of the purpose of this particular type of model. Then, each component type is equipped with all the models defined in the required representation.

These models will be used later for generating the models for the formal methods, which will be discussed in Sect. 4.

3.5 Views

Up to now, we have discussed the basic structure of a component library. Once all these concepts are defined in a component library, a *project editor* can be used for editing a construction plan from the defined components.

At last, we discuss a concept that supports different kinds of users involved in the same project. For example, there could be electrical engineers, mechanical engineers, and computer scientists working on the same project. All of them have different points of views and a different focus. In order to allow the different groups of users to focus on those aspects that are important for them, a component library may define different *views*.

Each view, basically, consists of a *name*, a *description* of its purpose or the class of intended users as well as a set of ports, connections, and components that are visible in that view. Moreover, each view can define a different appearance for the ports, connections, and components.

When an engineer selects a specific view on the project, he sees only the parts defined in that view. And the parts appear as defined for that particular view. The underlying construction plan, however, is the same for all engineers in order to avoid inconsistent views.

These different views can be all viewed and edited with a standard editor of Component Tools. But, it is also possible to add *dedicated editors* for a particular view[1]. For example, there could be an editor in which the physical extensions of the components are exactly visualised. In fact, the size and the position of the component instances defines the value of the corresponding parameters. Also the position of the ports could be changed and connectible ports could be connected by snapping to each other. Figure 5 shows how such a *geometry view* of the very same construction plan of our example could look like. Note, that this view does not show the ports for the electrical wiring; it shows the ports for the mechanical connections only.

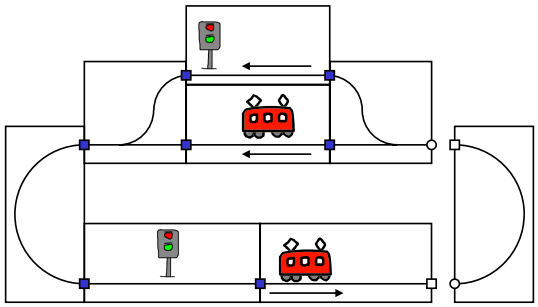

Fig. 5. The geometry view of the construction plan

[1] We use Eclipse plug-ins and extension points for implementing these concepts. But, these implementational details are not discussed in this paper.

3.6 Extensions

Note that there are all kinds of possible extensions when defining component libraries, which are not yet implemented, but could be useful in order to properly support the development tasks. For example, components are displayed as rectangles in the current version. Future versions could support other freely definable shapes such as polygons or dedicated images; in some views there could be even some vertexes that could be freely moved in order to define some parameter. Moreover, there could be all kinds of icons associated with the different components.

Up to now, all components are defined in XML documents. Future versions of Component Tools could also support the user to define his own components in a hierarchical way from components in the library. The concept itself is not difficult; still it is not yet implemented.

Up to now, there is only a very simple mechanism for defining the connection paradigm: a list of pairs that maybe connected by this connection along with a simple fan-in and fan-out. A future version could use a much more powerful mechanism; the exact mechanism that suits the needs of typical components and connections, however, has yet to be identified.

4 Tasks and Transformations

Up to this point, a component library can be used for building construction plans from its components and connections, and different engineers can work on them in their favourite view. The models along with their description might be useful in understanding the behaviour of each component; but, strictly speaking, the models do not have a function at all.

The purpose of the models will be explained in this section. Basically, the models will be used for constructing some overall model out of the construction plan in some formalism on which some formal method can be applied. In order to support transformations back and forth between the construction plan and the tools supporting a formal method, we use *triple graph grammars* (*TGGs*) for defining the transformations and combination of the different models [1]. By contrast to the classical approach, however, we use an interpreter for actually executing the transformation [2, 3].

4.1 Triple graph grammars

Here, we cannot discuss TGGs in full detail. Rather, we will explain the idea of TGGs by the help of our example.

Figure 6 shows a TGG rule that captures the translation of an instance of a signal unit to the corresponding Petri net. The graph resembles UML's object diagrams, where circle nodes are used only for emphasising the correspondence to the respective ports of the component and to the places of the Petri net. The Petri net model was shown already in Fig. 3. Basically, the part below the

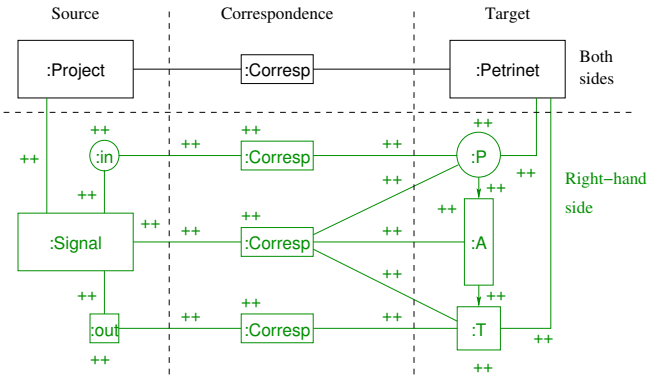

Fig. 6. The TGG rule for a signal unit instance

dashed line shows an instance of the signal unit with its two ports called in and out on the left-hand side, and it shows the corresponding Petri net model on the right-hand side. In the middle part, it shows the *correspondence* or mapping of the elements of the component to the elements of the Petri net. For example, all elements of the Petri net correspond to the signal unit, whereas the place of the Petri net corresponds to the in port of the component, and the transition of the Petri net corresponds to the out port of the component[2]. Minus syntactic sugar, the part below the dashed line of the TGG rule in Fig. 6 is a more formal presentation of the Petri net model corresponding to the component along with the mapping of the ports.

Next, we explain the meaning of Fig. 6 as a TGG rule. It represents a graph grammar rule. As all grammars, it consists of two parts a left-hand part and a right-hand part, where the parts that do not change occur on both sides. The top part above the dashed line of Fig. 6 belongs to both sides of the rule, and the part below the dashed line belongs to the right-hand side only. Such a rule means that once there is a graph as shown above the line, the part below the line can be added. This is the reason, why all edges and nodes in the lower part have a label ++. In fact, it is only these labels which makes the nodes right-hand side nodes – the horizontal dashed line is not part of the TGG rule. So, this rule, in combination with similar rules for the other components, can generate a set of component instances in the construction plan (called project) along with the corresponding Petri net models. Though the translation will not be executed this way, conceptually, we can assume that whenever a component is added to the project on the *source* side of the TGG, the corresponding Petri net elements will be generated on the *target* side along with correspondence objects that link the nodes on the source side with nodes on the target side. This way, the TGG

[2] Actually this was the reason for choosing the shape of the in port as a circle and the out port as a square.

rules exactly define a transformation from a construction plan, i. e. a project on the source side to a Petri net on the target side.

Up to this point, the connections have not yet been considered. In our simple example, we need only one TGG rule for dealing with the connections. This rule is shown in Fig. 7. Again, the new parts (the ones occurring on the right-hand side of a graph grammar only) in this rule are labelled with ++. This rule can

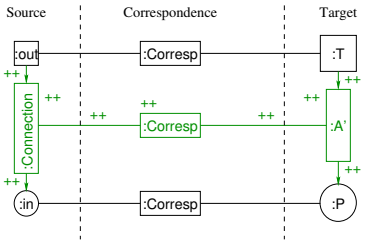

Fig. 7. The TGG rule for a connection

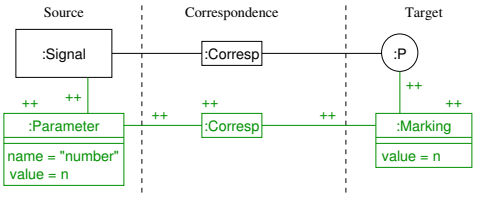

Fig. 8. The TGG rule for initial markings

be read in the following way: When a connection from some out port to some in port is inserted to the project, a Petri net arc is inserted to the Petri net between the transition and place corresponding to these ports.

Altogether, one TGG rule for each component and its model and the single rule for the connection precisely defines a translation of the construction plan to a Petri net model. For example, the construction plan from Fig. 1 along with the TGG rules for the simple Petri net models would translate to the Petri net shown in Fig. 2 – except for the initial marking, which will be discussed shortly.

When explaining the TGG rules, we assumed that both sides, the construction plan and the Petri net, are created at the same time. But, this is not necessary; we used this scenario only for explaining the idea and the general principle of TGGs. In practise, we assume that we have a construction plan, and we try to map a part of the project with the source part of the TGG rule and than generate the missing part in the target part and the correspondence part. A first idea of such an interpreter of TGGs was discussed in [2, 3] and a prototype implementation was just finished.

As mentioned already, the above TGG rules do not yet generate an initial marking for our Petri net. The reasons is that we did not consider the parameters of the component that define the number of the trains on that component initially. Figure 8 shows a rule that transforms the parameter number to a marking of the corresponding place. With this rule added, we would obtain the Petri net from Fig. 2 for the construction plan from Fig. 1. But, this rule can do more. Suppose, some analysis or simulation tool changes the marking of a place in the Petri net. Then, we can apply the rule backward, and translate the changed marking back to the component instance and change the parameter number accordingly.

In principle, we could even change the Petri net and change the construction plan accordingly. But, since not all Petri nets correspond to a construction plan, we might run in problems with more complicated changes here.

Altogether, TGGs are an appropriate means for defining transformations among different models, and for actually performing these transformations in both directions. In order to make this really work, we need to define the nodes and the associations between these nodes that may occur in the source model, in the target model, and in the correspondence part of the TGGs first. We call such definitions *graph models*. Figure 9 shows an example of a graph model for Petri nets, which resembles UML class diagrams. A TGG is based on such graph

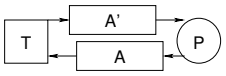

Fig. 9. Graph model for Petri nets

models and the nodes occurring in a TGG actually refer to these graph models[3].

4.2 Tasks

In order to complete the definition of a component tool based on a component library, we need to define how the tool supports the different tasks of the engineer. For simplicity, we call this the *task definitions*.

A task definition, basically, consists of two parts: a TGG transformation along with the corresponding graph models and an *external tool* that will be started on the transformed model. Technically, an external tool is a class implementing a tool interface with a method that passes the model to the external tool and a method that starts the tool on that model. Even more technically, this class must be installed as a plug-in in the Eclipse platform, but we do not go into these details here.

The external tool could do anything on the transformed model; it could modify it, analyse it save it or start third party tools on it. Actually, we distinguish two kinds of tasks. *Attached tasks* are those that stay connected to the original model (via the TGG correspondences) and all modifications made on the transformed model will be transformed back to the construction plan and can be seen in the editor. *Detached tasks* do not stay connected to the original models once they are started.

Clearly, the attached tools are the more attractive ones since they show their results or effects in the construction plan of the engineer. So, the engineer does

[3] Here we discuss the conceptual part only. For technical reasons, we will need some mapping from Java objects implementing a graph model to the objects of the graph model. These *model adapters* will be discussed in Sect. 5.

not need to adjust to a different formalism or notation for viewing the results. In some cases, the engineer might want to interact with the tool directly, in which case it might be used as a detached tool.

4.3 The example revisited

Altogether, a component library along with the transformations and task definitions defines a tool for a particular application area. With these definitions Component Tools implements this *component tool*: With this component tool, an engineers can use the components of the component library for editing construction plans and different engineers can have their particular view on these construction plans. The component tool will provide buttons for each defined task so that the engineers can start the corresponding transformations and external tools, and – for attached tasks – see the result in their construction plan. The different models and formalisms for each component can be inspected by the engineer, but it is not necessary for him to see them and to know the formalisms at all.

For our toy train example, we have equipped the different components with simple Petri net models and the transformations discussed above. The external tool started on the transformed model is the *Petri Net Kernel* (PNK) [4] along with the PNVis tool [5], which simulates the Petri net and animates the behaviour in a 3D-visualisation. All the user must do for starting this visualisation is providing a construction plan as shown in Fig. 1 and then press the button of the corresponding task. What is more, while the 3D-animation is running the number of the corresponding shuttles at the particular instances of the components would change in the construction plan accordingly.

Other task, working on the very same construction plan, can be easily added. Right now, we are working on external tools for the controller synthesis, and for generating code for PLC controllers for flexible manufacturing systems, and fault analysis during the ramp-up process of such plants.

5 Implementation and Future Extensions

In the previous sections, we have discussed Component Tools on a conceptual level and from the perspective of potential users of Component Tools. Actually, there are two kinds of users, the first type is an engineer defining a component tool for a specific application area using some specific formal methods; the other type would be a user of that component tool. In this section, we discuss some implementation issues of Component Tools and give an overview on the state of the implementation and some future extensions.

A first prototype of Component Tools was implemented based on the Eclipse platform [6]. This prototype covered the basic concepts of the component library, but did not cover transformations and did not support tasks. Basically, the implementation consisted of an editor for construction plans. The component library, its ports, connections, components was defined in some XML files.

In a one-year master's project [3], this first prototype was extended into an implementation covering all aspects discussed in this paper by a group of ten students. In particular, the concepts of views and dedicated editors for some of these views were added. Moreover, the concepts of transformations and tasks were added, which included an algorithm that interprets a TGG for transforming the corresponding models.

This TGG interpreter works on Java implementations of the underlying graph models. In order to access these Java implementations, the Java implementation must be mapped to the graph model of the TGGs. To this end, we developed the concept of an *model adapter* that must be implemented for each Java implementation of a particular model. With the help of these adapters, a TGG interpreter can transform the source model into the target model and vice versa [2].

Altogether, the current implementation of Component Tools covers the example discussed in this paper. A first version of it will be made available under the GPL very soon. Some interesting features, however, are still missing, which will be added in the near future by some Bachelor and Masters theses:
Library and TGG editors: Up to now, the component library is defined by several XML files. For practical use, it is important to have a graphical editor for defining a component library and, in particular, the triple graph grammars defining the transformations. With such an editor, the component library could even be extended dynamically by user defined components and components could be defined from other components.
Visualising analysis information: Up to now, the result of the tools can be displayed in the construction plan only by changing some of its parameters. In order to make the visualisation more flexible, we need to develop a more sophisticated concept. For example, some ports, connections, or components could be high-lighted.

In many cases, results of analysis tools can be represented in terms of scenarios as an abstract form of executions. Then, the scenarios in the target model must be transformed back to a scenario in terms of the ports and components of the construction plan. This scenario could be either displayed or animated in the construction plan. The details of this idea, however, have still to be worked out (see [7] for some more details).
Generic adapter: Up to now, a new adapter must be implemented whenever a new Java implementation of a source or target model is needed for some new tool or component model, which needs some programming effort. The general idea of Component Tools is that defining a new component library, new transformations, and tasks does not require any programming effort (or at least not much). Therefore, we are thinking of implementing a language for describing this mapping and implementing a *generic model adapter* that takes this description as a parameter and then serves as a model adapter for the described mapping. It could use XMI technology for mapping meta-models to its XML representation, for example.

In addition to that there are many long-term ideas on how Component Tools could be equipped with cool features. These, however, are beyond the scope of this paper.

6 Related Work

In this section, we give an overview of the related work and tools and systems that inspired our work.

6.1 Tools

First, we discuss typical representatives of tools for the design, simulation, and visualisation of plants from components.

Tool suites such as *Simulink* of The MathWorks, $SimOffice^{TM}$ of MSC-Software, $AutoMod^{TM}$ [8] and *eMPower* [9] are used for simulation and provide an environment for building the models using the libraries of components. They support modelling in different areas, such as airport industry, manufacturing constructions, and logistics. The *LONTROL* tool [10] is focused on control engineering for material handling, automation, and assembly logistics. It uses a library of components for building the models and functional components to control the system. The research tool $d^3FACT\ INSIGHT$ [11] is built to support the analysis of simulation models. The focus of this tool lies on the possibility to create and simulate complex models based on components in collaborate work. The SEA Environment [12] presents a methodology for the design, analysis, and simulation of embedded real-time systems using different modelling paradigms and tools.

All of the tools discussed above use components in a similar way to Component Tools. The difference is in the purpose. Basically, all of the above tools support simulation and visualisation. The focus of Component Tools is on analysis and verification and the possibility for integrating new formal methods.

6.2 Model Transformations

With the advent of the Model-Driven Architecture (MDA) [13], model transformation has been put into the focus of many research activities. This leads to many different approaches for model transformation – each for a special purpose and within a particular domain with its own requirements. Here, we cannot give a complete discussion (see [14] for a survey).

The best-known approach for model transformation is XSLT [15]. It is used for the transformation of models represented as XML documents via the XMI specification. However, the description of the transformation is done textually in a highly procedural form. Hence, the specification of a transformation is not very user friendly.

Another class of transformation approaches comprises graphical transformation languages which are based on graph grammars and graph transformations.

These approaches operate on graphs representing the data structures which have to be transformed. The transformation is executed by searching a pattern in the graph and applying an action which transforms the pattern to a new data structure. Examples for model transformation approaches based on graph grammars and graph transformation include VIATRA [16], GreAT [17], and UMLX [18]. Common to all mentioned approaches is that the transformation must be specified for each transformation direction separately. Hence, it is not well suited for the specification of bidirectional transformations as required by our approach. This is why we use TGGs [1].

7 Conclusion

In this paper, we discussed the basic ideas and concepts of Component Tools. The objective of Component Tools is to combine a bunch of different formal models, formal methods, and tools supporting them under a uniform visual front-end, which defines a component tool. With such a component tool, an engineer can use the component library and all the associated tasks without even knowing the underlying formalisms. What is more, there is only one composition mechanism for constructing systems from component libraries; it is up to the transformations defined along with the component library to map these compositions to the composition mechanism provided by the underlying formal method.

We have discussed these ideas by the help of a simple toy-train example which resembles applications in our current field of research, flexible manufacturing systems. But, we believe that the scope of Component Tools is much broader. By the help of our example, we have demonstrated that the concepts do work and we are currently working on component tools that support engineers in the area of flexible manufacturing systems in the task of designing PLC code and in identifying hardware faults during the ramp-up phase of a new flexible manufacturing system.

Up to now, the different models of a component library are completely independent of each other. It is an interesting (but formalism-dependent) task to investigate conditions that guarantee that one model refines another or that different models are consistent to each other so that the results of different tasks based on different models can be combined. This, however, is left to future research.

Acknowledgement We would like to thank all the participants of our one year master's project 'Component Tools': A. Gepting, J. Greenyer, A. Maas, S. Munkelt, C. Pales, T. Pivl, O. Rohe, M. Sanders, A. Scholand, and C. Wagner. The discussions in this project significantly shaped the concepts of Component Tools and made our blurred vision much clearer and come true in a prototype implementation of Component Tools.

References

1. A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science, 20th International Workshop,*

WG '94, ser. LNCS, E. W. Mayr, G. Schmidt, and G. Tinhofer, Eds., vol. 903, Herrsching, Germany, June 1995, pp. 151–163.
2. E. Kindler, V. Rubin, and R. Wagner, "An adaptable TGG interpreter for in-memory model transformation," in *Proc. of the Fujaba Days 2004*, Darmstadt, Germany, Sept. 2004, pp. 35–38.
3. A. Gepting, J. Greenyer, E. Kindler, A. Maas, S. Munkelt, C. Pales, T. Pivl, O. Rohe, V. Rubin, M. Sanders, A. Scholand, C. Wagner, and R. Wagner, "Component tools: A vision of a tool," in *Proc. of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN)*, ser. Tech. Rep. tr-ri-04-251, Paderborn, Germany, Sept. 2004, pp. 37–42.
4. M. Weber and E. Kindler, "The Petri Net Kernel," in *Petri Net Technologies for Modeling Communication Based Systems*, ser. LNCS, H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, Eds. Springer, 2003, vol. 2472, pp. 109–123.
5. E. Kindler and C. Páles, "3D-visualization of Petri net models: Concept and realization," in *Application and Theory of Petri Nets 2004, 25^{th} International Conference*, ser. LNCS, J. Cortadella and W. Reisig, Eds., vol. 3099. Springer, June 2004, pp. 464–473.
6. J. 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, Oct. 2003.
7. H. Giese, E. Kindler, F. Klein, and R. Wagner, "Reconciling scenario-centered controller design with state-based system models," in 4^{th} *International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'05), Satellite event of ICSE '05*, May 2005.
8. Brooks Automation Corporate, "AutoMod Suite," http://www.automod.com/products/products.asp.
9. Tecnomatix Technologies Ltd., "eMPower Solutions," http://www.tecnomatix.com/.
10. FASTEC GmbH, "Lontrol," http://www.fastec.de/.
11. W. Dangelmaier, B. Mueck, C. Laroque, and K. R Mahajan, "d3fact insight: A simulation-tool for multiresolution material flow models," in *Simulation in Industry - 16th European Simulation Symposium (ESS2004) SCS - Europe*, I. Lipovszki, György; Molnár, Ed., 2004, pp. 17 – 22.
12. C. Rust, J. Stroop, and J. Tacken, "The Design of Embedded Real-Time Systems using the SEA Environment," in *Proc. of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98), Adelaide, Australia*, Sep 1998.
13. OMG, *Model Driven Architecture*, http://www.omg.org/mda/.
14. K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
15. W3C, "XSL Transformations (XSLT) Version 1.0," November 1999. [Online]. Available: http://www.w3.org/TR/xslt
16. D. Varró, G. Varró, and A. Pataricza, "Designing the automatic transformation of visual languages," *Science of Computer Programming*, vol. 44, no. 2, pp. 205–227, August 2002.
17. A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, and A. Vizhanyo, "The design of a simple language for graph transformations," *Journal in Software and System Modeling*, 2005, in review.
18. E. D. Willink, "UMLX: A graphical transformation language for MDA," in *MDAFA'03*. Entschede, Netherlands: University of Twente, September 2003, pp. 13–24.