Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 263 (2010) 145–160
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Systematic Approach to Construct Compositional Behaviour Models for Network-structured Safety-critical Systems

Johannes Kloos and Robert Eschbach [1]

Testing and Inspections
Fraunhofer Institute for Experimental Software Engineering
Kaiserslautern, Germany

Abstract

This paper considers the problem of model-based testing of a class of safety-critical systems. These systems are built up from components that are connected a network-like structure. The number of possible structures is usually large. In particular, we consider the following issue: For many of these systems, each instance needs its own set of models for testing. On the other hand, the instances that should be tested will have to be chosen so that the reliability statements are generally applicable. Thus, they must be chosen by a domain expert. The approach in this paper addresses both of these points. The structure of the instance of system under test is described using a domain-specific language, so that a domain expert can easily describe a system instance for testing. At the same time, the components and composition operators are formalized. Using a structure description written in the DSL, corresponding test models can be automatically generated, allowing for automated testing by the domain expert. We show some evidence about the feasibility of our approach and about the effort required for modelling an example, supporting our belief that our approach improves both on the efficiency and the expressivity of current compositional test model construction techniques.

Keywords: Model-based testing, safety-critical system, DSL

1 Introduction

Safety-critical embedded systems such as control applications in industrial automation or train automation are very complex due to their broad range of possible applications. Among these systems, industrial plant controllers, railroad coordination systems, logistic planning systems etc. form a class that is especially adaptable to many different application scenarios. One thing that is common to many of these systems is that they control an underlying network of physical entities that are connected in some way, often with several hierarchy levels. The network formed by these entities induces data flows between neighbouring components and vertically through

[1] Email: ⟨firstname⟩.⟨lastname⟩@iese.fraunhofer.de

doi:10.1016/j.entcs.2010.05.009

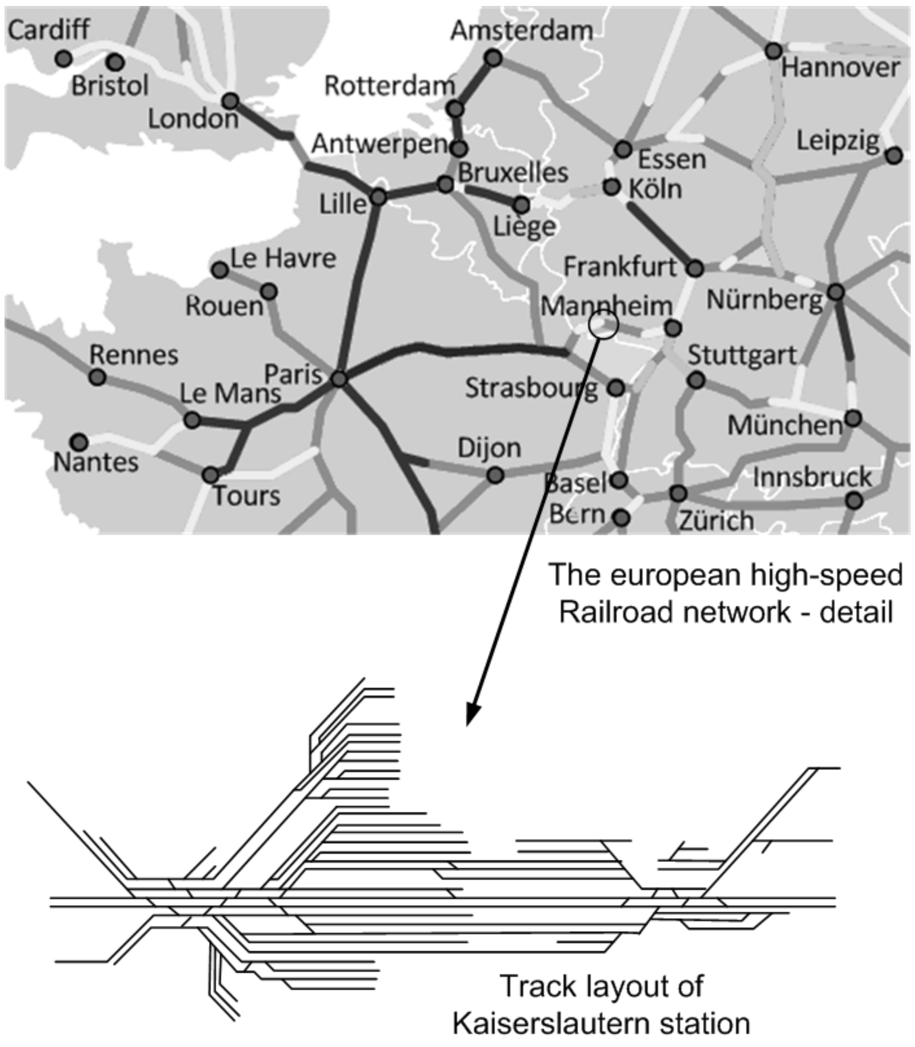

Fig. 1. An example of a hierarchical network structure: The European high-speed railroad network and a detail, Kaiserslautern station

the different levels of hierarchy. These data flows are observed and manipulated by the control system. As an example, a railway control system for a given segment of the railroad network will use the underlying structure of the tracks, switches and signals on the lowest level of its internal model, aggregating them into larger control units that finally form the network (cf. figure 1). We will call this kind of systems "network-structured systems".

To handle the complexity inherent in these structures, one way to construct such embedded control systems is a modular, component-based approach. For each installation of such a system, e.g., for a given industrial plant or a given railway network, an instance of the embedded system is created by instantiating some basic component types and plugging them together in an appropriate way. For many of these systems, one finds that the number of possible instances, is huge and effectively unbounded.

From the perspective of quality assurance it is not clear how to test such a huge space of instances w.r.t. different quality properties like correctness, safety or reliability. Reliability is one of the most important non-functional quality properties of embedded software systems required by contract or different standards like the IEC 61508 [10], IEC 61511 [9], CENELEC 50126 [3], ISO CD 26262 [11] or DO178B [17]. Measures like mean time between failures (MTBF) or failure rates are typical reliability indicators.

The problems that a tester faces for these systems are twofold: For one thing, which instances should be tested? If one is interested in generally applicable statements about system reliability, a set of instances that exercises the handling of critical situations is required. In most cases, only a domain expert will be able to decide exactly which instances are relevant here.

On the other hand, even if the instances to test are known, one still has to generate test cases for each of these instances. When one tries to demonstrate the reliability of a system, a large number of test cases per instance is necessary. Multiplied with the number of instances, the amount of required test cases becomes huge.

A common approach to solve the problem of test case construction is the automatic generation of test cases. Among the possible method for generating test cases, model-based test approaches form an important and well-researched subclass. These techniques generate test cases (i.e., test inputs and expected results) from one or more models that describe the behaviour of the system and/or the system's environment. These models are known as test models.

When testing for reliability, one usually chooses techniques that generate test cases based on the operational profile of the system, allowing unbiased estimations of reliability measures. One of these techniques is Model-Based Statistical Testing [20], also known as Statistical Testing. This approach uses a description of the possible system inputs from a user's point of view to generate test cases. This description is formalized as a Markov chain of possible input event sequences. The expected outputs can either be annotated to the Markov chain, or calculated using a second model, the system model, for more complicated systems.

When trying to apply model-based testing techniques to network-structured systems, one finds that each instance requires test models tailored to the underlying network structure. To avoid the overhead of generating test models for each instance, it appears worthwhile to construct test models for each instance from some general information and the description of the concrete instance. As the structure of the instances lends itself to a natural composition-based description, a component-based approach seems promising. Naturally, this approach must be designed in such a way that the instance description given by the domain expert can easily be translated into an instance of the test models.

Given some method that can be used to construct test models for a network-structured system from pre-defined components and a description of the system instance by a domain expert, it becomes possible to carry out testing of this system with little work for each instance, as the most labour-intensive step, namely the

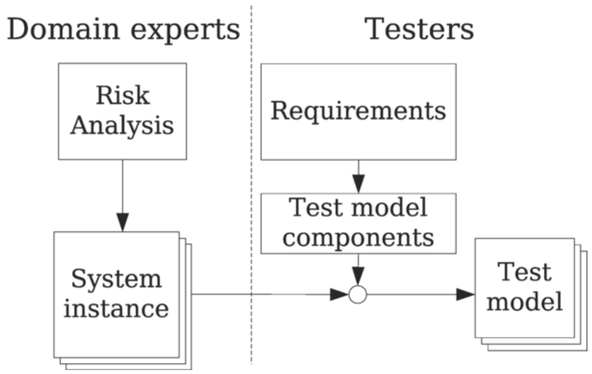

Fig. 2. Elements of the approach

construction of the test model components, has to be done only once, and the translation from the domain expert's representation of an instance to test cases can be fully automated (cf. figure 2). Moreover, this approach allows the domain experts to carry out (parts of) testing without the burden of having to learn the details of testing the system under consideration.

In this paper, we present an approach called COMPOSE that allows for the systematic compositional construction of test models using domain-specific information. This approach comprises an integrated technique to describe state-based component behaviour and compose these components according to a specification given in a domain-specific language.

We believe that our approach improves both the efficiency and the expressivity compared to state-of-the-art work in the compositional construction of test models. In particular, we state the following claims:

(i) Constructing one composite model using the modelling techniques of COMPOSE does not require more effort than using other techniques.
(ii) The construction of several composite models for different system instances is more efficient using COMPOSE when compared to other techniques.
(iii) The resulting models produce identical test results.

The rest of this paper is organized as follows: In section 2, prior work is described. The COMPOSE approach is described in section 3. It is illustrated by a running example. Section 4 describes how we plan to evaluate our approach with the claims stated above, and gives some preliminary results. Finally, we summarize our work and point out further improvements and directions.

2 Related Work

Many ideas in this paper are based on common ideas found in component-based software and model-based engineering. The focus of this paper is not on extending these results, but rather on the application of these techniques to software testing, in particular, to model-based testing. It continues the work of a prior case study from the railway domain, which has been described in [12]. As noted there, the use of domain-specific languages for describing compositional models is, in itself, not a new idea; see, e.g., [7].

Often, composition operators are given by describing the action of the operator in natural language, usually together with a mathematical specification (cf. [5,21,13] and various others). We are not aware of any formalizations of composition operators in some machine-readable language, except for some simple examples of operators for process calculi [14].

Most model-based testing methods are based on automata or Markov chains. Apart from the most commonly described composition operators, namely different forms of parallel composition [18], some more general operators have been defined in the literature.

For Markov chains, only few compositional approaches exist. Parallel compositions are obvious candidates, but are subsumed in other approaches. One general approach that we found is known as Stochastic Automata Networks [4]. It describes a Markov chain as a set of stochastic automata. These automata can interact using synchronized events (i.e., some transitions of the Markov chains are labelled using an event name and may only fire simultaneously) and functional rates (i.e., a transition rate may depend on the current state of some other Markov chain). Another approach is the Test Generation Language [15], which describes how stimulation sequences generated by several different Markov chains can be woven into one stimulation sequence for testing. This is done by giving several operators working on event sequences. In our experience, using a high-level approach like the Test Generation Language is preferable, as it allows clearer descriptions of intent.

For Mealy automata and more general state-event systems, a number of compositional techniques exist. First of all, Statecharts [6] can be seen as a method to describe the composition of state-event systems in a compact notation. Another approach is the use of communicating automata, for example in UPPAAL [2]. Further compositional approaches have cropped up in the area of Model-Driven Development. Some of these approaches (e.g. [19]) describes connectors using process calculi, which is quite close to the approach taken here. As our goal was to have a single technique for the construction of composite Markov chain models and system models, these approaches were not directly applicable to our problem, as these approaches do not provide appropriate handling for Markov chains.

Choosing a different point of view, model-based testing needs to generate sequences of stimulus events. When abstracting away the details of which underlying model is used, we are left with event-generating processes. These processes can be naturally described using process calculi, such as the π calculus. Indeed, the π calculus [14] already has a notion and graphical notation for the composition of different communicating processes. By modelling both component models and composition operators as communicating processes, one can use this kind of process composition to easily describe complex models.

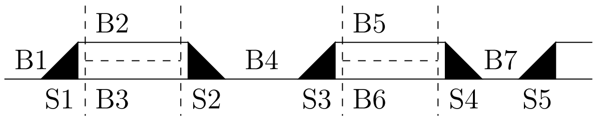

Fig. 3. An example track layout. Notation: The black triangles labeled S1 etc. are switches. B1 up to B7 are blocks, whose borders are the dashes lines.

3 The approach

In the following, the approach sketched above will be described in detail. As a running example, the construction of a behavioural model for a Train Control System [12,22] will be shown.

The approach has five steps:

(i) Identify the atomic components.

(ii) Determine appropriate composition operators.

(iii) Describe the behaviour of the atomic components using finite-state automata.

(iv) Define the behaviour of the composition operators, using the stimulus stream model described below.

(v) Define a DSL that uses the composition operators and atomic components to build a system model from the domain description.

Usually, one wishes to have a direct mapping from syntactic components of the DSL to composition operators and atomic components. On the other hand, a true one-to-one mapping will usually not be possible – some information may be implicit in the domain description, or there may be extraneous information.

For the example, an assistance system for a railway operating procedure known as “Zugleitbetrieb” is modelled. This system is called ZLB-PS. Briefly, the operation of this system can be summarized as follows:

(i) ZLB-PS is an assistance system that acts as substitute for the “train sheet”, a paper form that records the position and movement of trains as well as track reservations.

(ii) ZLB-PS uses track-side equipment (e.g. vacancy detectors) to counter-check and enforce the train director’s decisions. All track-side equipment is controlled by ZLB-PS.

(iii) The railroad network is partitioned into blocks, which contain switches and other track-side equipment. Moving a train means moving it from one block to another.

(iv) ZLB-PS can be configured for arbitrary track layouts.

(v) For a train to move from A to B, it must first reserve a track and ensure that this track is set up. During movement, it must notify the train director of arrival and may also indicate departure.

The track layouts used for ZLB-PS are network-like structures, which motivates our use of this example. A simple but typical layout is depicted in figure 3.

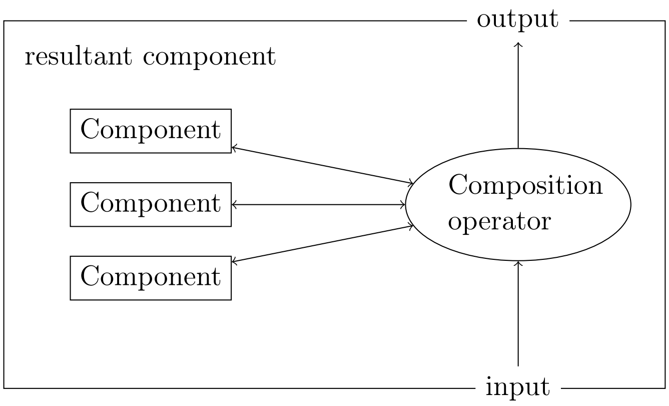

Fig. 4. The mechanics of composition operators; the arrows denote data streams

3.1 Identification of atomic components

In the first step, the basic components from which the test model is to be built must be identified. As a starting point, one starts from the domain expert's view of the system, choosing elements of this description as candidates for components. These components are then compared against the (informal) description of the system. One may find that some components are not explicitly described in the system representation chosen above, e.g., with coordination components that have only one instance. These components are then added to the candidate set.

The candidate set can then be considered element by element. One may find that an element does not actually contribute to the behaviour being modelled or does not have a clear-cut behaviour. In this case, the element should be removed or replaced be suitable other components.

This step may be iterated when new information from further steps is available, adding, removing or changing components as required. In any case, one should try to keep a simple mapping from the DSL to the components.

As a first approximation, one finds that the structure of a railroad network is made up from lengths of track, switches and block borders. Looking at the system's requirements, it can be seen that the block borders themselves as well as the lengths of track are not directly considered, but make up parts of a block. Hence, one exchanges the track and block border components for a block component.

Also, signals are mentioned. As they can occur only on block borders, it is clear how they can be modelled in the domain expert's description: they correspond to block borders.

Finally, it becomes clear that reservations et cetera are described on a level above blocks, therefore a small mediating component will be needed as well, the central interlocking unit (short CIU). The component types are summarized in Figure 4 as rectangles with thick borders.

3.2 Determination of composition operators

Based on the identified components and the set of possible configurations, one next identifies how these components should be composed. The composition of a set of components yields a new, larger component, allowing a hierarchical description of the resulting system.

For this approach, one takes a black-box view of the components, describing their behaviour solely using streams of events entering and leaving the component. The job of the composition operator is to tie together the components by connecting to their input and output streams and coordinating which data is sent to and received from each component (cf. Figure 4). The possible composition operators can range from very simple operations, such as a simple scatter/gather operator that simply broadcasts inputs to all attached components and passes through their outputs, to complex and application-specific operators.

The railroad example will use two composition operators: At one level, a block contains its signals and switches, who can only operate correctly in cooperation with their environment inside the block. Thus, a "low-level" composition operator is defined, which describes how a block instance is built up from switch models, signal models and the (abstract) model of a general block. This composition operator needs additional information, namely the concrete layout of switches, signals and connections inside the block.

The concrete block instances, together with the central interlocking unit, can then form the description of the complete system. Here, the "high-level" composition operator is used, which describes how the signals from the CIU are distributed to the concrete block instances. Again, this operator needs extra information, namely which blocks are neighbours of a given block.

Again, a summary of the composition operators and their application is shown in Figure 4: The composition operators are shown as ellipses, their configuration information is given by the italic text and the components that are composed using a composition operator are connected to it via arrows. The result of a composition is given by a rectangle (with thin borders) surrounding all necessary information; it forms a new component, which can again be composed with other components.

3.3 Describing the atomic components

The next step is the description of component behaviour. As single components are modelled as Mealy machines, one can use various methods in this step, e.g. by direct construction of the state/transition graph [8], by transformation from Statecharts (cf. [6]) or a wealth of other techniques (e.g., [16]).

For the running example, sequence-based specification was chosen because backwards traceability to the requirements was considered as an important requirement [12]. The application of this method yielded four Mealy machines, which are described in the paper just referenced.

3.4 Describing the composition operators

Now that the interfaces of the components are known, the composition operators can be constructed. To do this, one assumes that each Mealy machine is a (communicating) process with an event input stream and an event output stream. The role of a composition operator is thus to describe how several such Mealy machine processes can be connected to form a larger component with one or more input and output event streams (cf. the section on the determination of composition operators, especially Figure 4.

A good way to describe the behaviour of a composition operator is to use a process calculus, such as CSP or the π-calculus. The approach described in this paper is based on a (slightly extended) version of π-calculus [14]. A companion version of the stochastic π-calculus exists as well, which is used to describe composition of Markov chain-based usage models for software testing. For reference, the following syntax is used for the π calculus:

```
Def ::= name ’(’ Patterns ’)’ ’:=’ Proc
                                        Process definition
Proc ::= ’0’                            Null process
     | ’(’ Proc ’)’
     | GuardedProcs
     | Proc ’|’ Proc                    Parallel composition
     | ’replicate’ Proc                 Proc is replicated, i.e.,
                                        infinitely many parallel copies are created
     | ’new’ Bindings ’.’ Proc          Introduction of new name bindings
     | name ’(’ Terms ’)’               Process instantiation
GuardedProcs ::= GuardedProc
     | GuardedProc ’+’ GuardedProcs
                                        Nondeterministic guarded choice
GuardedProc ::= Guard                   When the guard holds, do nothing
     | Guard Proc                       When the guard holds, perform
                                        new bindings and execute Proc
Guard ::= var ’(’ Patterns ’)’ ’?’      Receive on channel, with data
     | var ’?’                          Receive on channel, no data
     | var ’(’ Terms ’)’ ’!’            Send on channel, with data
     | var ’!’                          Send on channel, no data
Patterns ::= Pattern | Pattern ’,’ Patterns
Pattern ::= var                         Accept any name/term, bind to var
     | ’_’                              Dummy pattern, accept anything
     | name ’(’ Patterns ’)’            Constructor expression
     | name ’(’ ’)’                     Empty constructor expression
Terms ::= Term | Term ’,’ Terms
Term ::= var                            The binding of var
     | name ’(’ Terms ’)’               Constructor expression
     | name ’(’ ’)’                     Empty constructor expression
Bindings ::= Binding | Binding ’,’ Bindings
```

Binding ::= var	Normal binding
| var ':' 'imm'	"immediate" binding

Semantics for the π calculus are given in Milner's book [14], while our extensions are described in a technical report [1].

As there is a natural transformation from Mealy machines to π processes, the composition of several Mealy machines using a given composition operator can be described by assuming that the Mealy machines are given by processes M_1 to M_k, and the composition operator by C. Then the resulting model is the π process defined similar to the following:

```
Composite(in, out) ::=
new in1, out1, ..., ink, outk.
    M1(in1, out1) | ... | Mk(ink, outk)
  | O(in, out, in1, out1, ..., ink, outk)
```

In the following example, it will be shown that the π calculus can be used as a kind of "programming language" to describe the operational semantics of the composition operators. This has several advantages:

(i) It is easy to define application-specific operators.

(ii) Tool support for model compositions is possible.

(iii) Operators can be built from other operators using simple language constructs, viz., process instantiation.

For ZLB-PS, two composition operators have to be modelled. These composition operators built as far as possible using standard operators. At first, consider the general structure of the operators.

The high-level composition operator works as follows: An instance of the Central Interlocking Unit is composed with a set of instances of concrete blocks. Whenever a stimulus describing some operation on a path in the network enters the Central Interlocking Unit, all blocks on this path are sent the corresponding stimulus in the reservation protocol. If the stimulus has a synchronous response, a transactional pattern is employed to guarantee that all blocks are in a consistent state.

Therefore, the composition operator works as follows: If the central interlocking unit outputs `request(f,t)`, the path from f to t is computed. Next, each path element is sent `request`in turn. If the request succeeds, the next element in the path is considered (the end of the path is handled in an appropriate way), and if this element is successfully reserved, OK is returned. If the next element fails (be it because the element itself failed or some problem happened later in the path), or if the request does not succeed, NOK is returned. The handling of `tearDown(f,t)` is and `setup`is analogous. On the other hand, the `cancel`stimulus is simply sent to all relevant blocks.

As a π-calculus process, this operator can be expressed as follows:

```
// Entrance to the high-level composition
```

```
HighLevel(ciu, listBlocks, adjacencies) :=
  // Initial state: normal operation state
  HighLevelNormal(ciu, adjacencies)
  // Run each block component instance,
  // all of them in parallel.
| Parallel(listBlocks);

// The behaviour of the central interlocking unit
// when not processing a request.
HighLevelNormal(ciu, adjacencies) :=
  // A request is received
  ciu(REQUEST(f, t))? new path, result.
    // Start three processes:
    // A path calculation...
    ( CalculatePath(adjacencies, f, t, path)
    // ... the transaction process that tries
    // to reserve all the blocks on the path (except
    // the first, which is a special case) ...
    | path(first)? Transaction(RESERVE(), path, result)
    // ... and the process that acts on the result,
    // providing an answer and entering the appropriate mode.
    | result(OK())? ciu(OK())!
        HighLevelRequested(ciu, adjacencies, f, t))
     +result(NOK())? ciu(NOK())! HighLevelNormal(ciu, adjacencies))
  // A teardown is received; basically, the same handling as for
  // a request.
 +ciu(TEARDOWN(f, t))? new path, result.
    ( CalculatePath(adjacencies, f, t, path)
    | path(first)? Transaction(TEARDOWN(), path, result)
    | result(what)? ciu(what)! HighLevelNormal(ciu, adjacencies));

// The behaviour of the central interlocking unit
// when processing a request.
HighLevelRequested(ciu, adjacencies, f, t) :=
  new path, result. (
    CalculatePath(adjacencies, f, t, path)
  | ciu(SETUP())? (Tranasction(SETUP(), path, result)
                  |result(what)? ciu(what)!
     HighLevelNormal(ciu, adjacencies))
   +ciu(CANCEL())? (Distribute(CANCEL(), path)
                   |HighLevelNormal(ciu, adjacencies));
```

The HighLevel operator uses three helper processes, namely CalculatePath, Transaction (which implements the transactional pattern) and Distribute (which distributes a stimulus along a path). These processes can be implemented as follows:

```
Transaction(reqType, path, result) :=
  // end of path: no obstacle to the reservation
  path(END())? result(OK())!
  // still on path: try the request on the current
  // path element\dots
+ path(STEP(s))? s(reqType)! (
    // if it fails, stop transaction
    (s(NOK())? result(NOK())!)
// if it suceeds, try next element\dots
  + s(OK())? new resNext. (Transaction(reqType, path, resNext)
            // hand down result, and call rollback if necessary
          | resNext(OK())? result(OK())!
   +resNext(NOK())? s(ROLLBACK())! result(NOK())!)

// Simple recursive implementation of a
// "for all elements" operation
Distribute(reqType, path) :=
  path(END())?
+ path(STEP(s))? s(reqType)! Distribute(reqType, path);

CalculatePath(adjacencies, from, to, path) :=
  [Code omitted for brevity: A function
   that calculates an arbitrary path through
   the graph given in adjacencies starting at
   from and ending at to. The path is output by sending
   a sequence of STEP(s) terms, one for each vertex on the
   path, and finally a END() term.]
```

The implementation of the low-level composition operator can be given in a similar way. It re-uses the definitions of `Transaction`, `Distribute` and `CalculatePath`. Again, we omit the implementation itself for brevity.

3.5 Definition of a DSL

The final step of this method is to define a DSL describing instances of the system and mapping it to applications of the composition operators to appropriate component instances.

To describe an instance of a complicated system, domain experts have usually developed their very own notation, e.g., block diagrams of the system or railroad network diagrams. The language is derived from such a notation, and should be designed in a way familiar to domain experts. Additionally, the translation from syntactic elements of the language to instances of the components and composition operators should be easy to implement. Optimally, each syntactic element maps to a fixed set of component instances and composition operator applications.

For the running example, a DSL based on the track layout was chosen (cf. figure

Figure 3). It contains, among other information, data on which blocks, switch and signal components exist, and how blocs and switches are interconnected. For convenience, an already-existing XML encoding of the track layout was leveraged, which contained large amounts of extraneous information.

The elements that were relevant for the model construction were the following:

- The definition of a single component. This was used to instance the appropriate component models.
- The introduction of a neighbourhood relationship between two blocks. This data was filled into the block connectivity graph, used by the high-level composition operator.
- The introduction of the follower sets of a switch, where a follower set describes which blocks can be reached when a switch is set in a given position. This information was preprocessed and then used to fill the low-level connective graph for the low-level composition operator.
- The definition of a single block additionally implies the application of the low-level composition operator to this block and all track devices (i.e., switches and signals) belonging to this block.

Finally, the (implied) CIU instance is created once and the high-level composition operator applied to the results of the low-level compositions.

For the track layout given in 3, we get the following component instances and composition operator applications (see also 5:

(i) Five instances of the switch model, one each for S1 to S5.

(ii) Seven instances of the block model.

(iii) Eight instances of the signal model, two each for the blocks B2, B3, B5 and B6.

(iv) One instance of the CIU.

(v) Seven applications of the low-level composition operator, namely on for B1 and S1, one for B4 with S2 and S3, one for B7, S4 and S5 and one each for B2, B3, B5 and B6 together with their respective signals. Each of these operators yields a concrete block, called C1 to C7.

(vi) One application of the high-level composition operator, composing the CIU with C1 to C7 and yielding the complete system model.

4 Experiences from the example

As this paper was based on prior work by the authors [12], some limited experience about efficiency is available. In particular, the approach described in [12] works along the same lines as COMPOSE, with all steps but step 4 of the technique virtually identical. Not surprisingly, we find that the implementation of the composition operators as an executable Java program required significantly more effort (a few man-weeks) compared to their specification as π-calculus processes (two or three days).

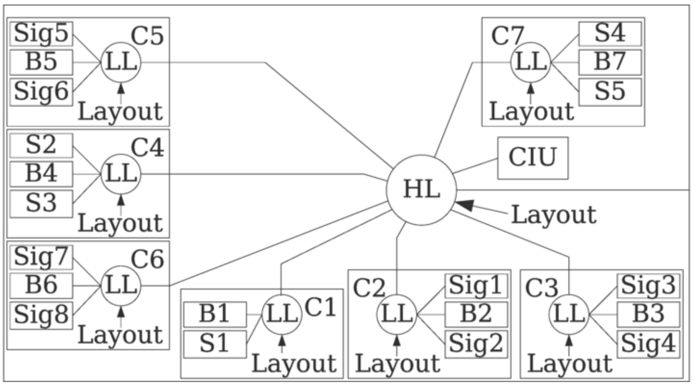

Fig. 5. The result of the composition. Each model (rectangle) is named by some letters symbolizing its type (B means block component, S switch, Sig signal and C concrete block) and, optionally, a number. Composition operators are signified by circles, where HL means high-level composition, and LL low-level composition.

All in all, we find that the COMPOSE approach seems to perform well: It permits the production of test models for network-structured systems with acceptable effort. This leads us to formulate several hypotheses:

(i) COMPOSE satisfies the requirements described in the introduction: It allows the construction of test models for network-structured systems in a way that is comprehensible for domain experts.

(ii) COMPOSE does not require significantly more effort to construct the artifacts for one composite model than other techniques.

(iii) COMPOSE requires significantly less effort to construct several different composite models, compared to other techniques.

(iv) The models generated using this technique generate the same test cases and/or test verdicts compared to models built by other techniques.

These hypotheses will form the basis of a serious evaluation of the COMPOSE approach. We plan to derive the necessary results by using a controlled experiment and/or an industrial case study. For the comparison to other, existing techniques, Stochastic Automata Networks [4] and Prowell's TGL [15] seem to be the most appropriate candidates.

5 Conclusion and Outlook

In this paper, we have demonstrated an approach called COMPOSE that allows for the systematic compositional construction of behavioral models. This approach comprises an integrated technique to describe state-based component behaviour and compose these components according to a specification given in a domain-specific language. In our approach, component behaviour is described using Mealy machines, while composition operators can be specified using process calculus expressions. This allows a tester or system expert to build models suitable for analysis and model-based testing. On the other hand, the instantiation of a model is described using a domain-specific language, allowing a domain expert to construct instances of the system and environment models for testing without any deeper knowledge about these models. The advantage of this approach is that the domain expert can easily test the system in as many configurations as he deems relevant and important, without having to worry about constructing test cases or computing test results manually.

Our approach is an extension of older work, described in a previous paper [12], where a special case of this approach was demonstrated. The improvement in this paper is that we can now specify the behaviour of the composition operators more easily, so that the approach can be applied with less effort.

Starting from here, we plan to do three things. The first task, which is already close to completion, is the implementation of a tool to support this approach. Next, we plan to apply this approach on different examples, e.g., an example from industrial control. Finally, we will investigate how the concepts behind this approach can be used for quantitative reliability estimation of complex systems.

6 Acknowledgements

The preparation of this work was funded by the Fraunhofer Innovation Cluster "Digital Commercial Vehicle Technology". The Fraunhofer Innovation Cluster "Digital Commercial Vehicle Technology" is funded by the European Union and the state of Rhineland-Palatinate in the context of the Ziel 2-Programm Rheinland-Pfalz. The example used in this paper was created by Siemens Mobility for the ranTest research project. The picture of the German railway network is based on a public domain picture taken from Wikipedia, cf. http://en.wikipedia.org/wiki/File:Bahn-Streckenkarte_Deutschland.png.

References

[1] Bauer, T., R. Eschbach, T. Hussain, J. Kloos and F. Zimmermann, *Komposition von benutzungsmodellen: Operatoren und anwendungsbeispiel*, Technical report, Fraunhofer IESE (2009).

[2] Behrmann, G., A. David and K. G. Larsen, *A Tutorial on Uppaal*, in: M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS (2004), pp. 200–236.

[3] EN 50126, "Railway Applications – The Specification and Demonstration of Reliability, Availability, Maintainability, and Safety (RAMS)." (1999), CENELEC, european Standard.

[4] Farina, A. G., P. Fernandes and F. M. Oliveira, *Representing software usage models with stochastic automata networks*, in: *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering* (2002), pp. 401–407.

[5] Garavel, H. and M. Sighireanu, *A graphical parallel composition operator for process algebras*, in: *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV 99)* (1998), pp. 185–202.

[6] Harel, D., *Statecharts: A visual formalism for complex systems*, Science of Computer Programming **8** (1987), pp. 231–274.

[7] Haxthausen, A. E. and J. Peleska, *Automatic verification, validation and test for railway control systems based on domain-specific descriptions*, in: S. Tsugawa and M. Aoki, editors, *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems* (2003), p. (pages unknown).

[8] Hopcroft, J. E., R. Motwani and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley Longman Publishing Co., Inc., 1990, 1 edition.

[9] IEC 61511, "Functional safety: Safety Instrumented Systems for the process industry sector," (2003), IEC.

[10] IEC 61508, "Functional safety of electrical/electronic/programmable electronic safety related systems," (2005), IEC, (draft).
URL http://www.iec.ch/functionalsafety

[11] ISO/CD 26262, "Road vehicles – Functional safety," ISO.

[12] Kloos, J. and R. Eschbach, *Generating system models for a highly configurable train control system using a domain-specific language: A case study*, in: *5th Workshop on Advances in Model Based Testing (A-MOST'2009)*, 2009, p. n/a.

[13] Milne, G., *The formal description and verification of hardware timing*, IEEE Transactions on Computers **40** (1991), pp. 811–826.

[14] Milner, R., "Communicating and Mobile Systems: The pi-Calculus," Cambridge University Press, 1999.

[15] Prowell and S.J., *Using Markov Chain Usage Models to Test Complex Systems*, System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on (2005), pp. 318c–318c.

[16] Prowell, S. J. and J. H. Poore, *Foundations of Sequence-based Software Specification*, IEEE Trans. Softw. Eng. **29** (2003), pp. 417–429.

[17] DO178B, "Software Considerations in Airborne Systems and Equipment Certification," RTCA.

[18] Schneider, K., "Verification of Reactive Systems - Formal Methods and Algorithms," Texts in Theoretical Computer Science (EATCS Series), Springer, 2003.

[19] Spitznagel, B. and D. Garlan, *A compositional formalization of connector wrappers*, in: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (2003), pp. 374–384.

[20] Whittaker, J. A. and J. H. Poore, *Statistical Testing for Cleanroom Software Engineering*, in: *Proc. Twenty-Fifth Hawaii International Conference on System Sciences*, 1992, pp. 428–436.

[21] Yevtushenko, N., T. Villa, R. Brayton and A. Petrenko, *Solution of Parallel Language Equations for Logic Synthesis*, in: *Proceedings of the International Conference on Computer Aided Design*, 2001, pp. 103–110.

[22] Zechner, A., "Entwurf einer generischen Systemarchitektur für den Zugleitbetrieb," Diplomarbeit, TU Braunschweig (2006).