DAWN for component based systems
– just a different perspective –

Ekkart Kindler
Universität Paderborn, Institut für Informatik, D-33095 Paderborn, Germany
kindler@upb.de

Abstract *DAWN* is technique for modelling and verifying network algorithms, which is based on Petri nets and temporal logic. In this paper, we present a different perspective of DAWN that allows us to use it for verifying *component bases systems* by modelling components and their interaction independently of each other.

1 Introduction

In a nutshell, a *component based system* consist of components which interact with each other. Each component encapsulates some kind of service. A component based system is built by combining and interconnecting the components. Therefore, the component based approach supports reusability and flexibility – the implementations of the individual components are independent from the overall structure (architecture) of the system.

For modelling a component based system, we need to model the behaviour of the individual components as well as their interaction. For modelling the components, there is a huge variety of formalisms: starting from programming languages such as *Java*, via modelling formalism such as *State Charts* [Har87], to more theoretical models such as *process algebras* [Mil89] or *Petri nets*. The interaction among the components is defined by *architecture description languages* (ADL), which, basically, identify the used components and how they are interconnected. Most of these languages were defined for automatically compiling a component based system from the components and the architecture definition, and for deploying it to a distributed hardware. Some languages, however, support the analysis of component based systems. For example, *DARWIN* [MDEK95] has an operational semantics, which is defined in terms of the π-calculus [MPW92].

In this paper, we show how to use the *Distributed Algorithms Working Notation* (*DAWN*) [WWV$^+$97, Rei98] for modelling and for verifying component based systems. Originally, DAWN was designed for modelling and verifying network algorithms. But, the underlying techniques are ready made for component bases systems, which will be demonstrated in this paper.

The components will be modelled by algebraic Petri nets and the architecture will be defined by giving a topology for the interconnected components. Given these two parts, we get an overall model of the component based system. Then, we can use the techniques of DAWN for verifying the resulting component based system. For example, we can use model checking for proving some properties. In some cases, we can show even for a class of architectures that all the resulting systems are correct.

In this paper, we cannot present this approach in full detail. We will present only the basic idea of DAWN and the different perspective that makes it work for modelling and verifying component based systems: a concrete system can be modelled in two separate parts: the *component models* and the *system architecture*. A component model defines the behaviour of a particular type of component. The system architecture defines how the different components interact with each other in a concrete system. Using phrases from software development, modelling components corresponds to *programming in the small*, whereas defining the system architecture corresponds to *programming in the large*.

Of course, there are many other aspects in *component bases software* such as reusability and separation of services from their implementation. These aspects are not covered in this paper. Here, we focus on the underlying mathematics of component models, architectures and the resulting systems, which will be presented by the help of a simple example.

2 The Example

As an example, we consider a simple algorithm for the calculation of a communication tree for a network of components such that each component has a minimal distance to some distinguished components, which are called *root components*. The other components are called *inner components*. So, there are two different kinds of components, which will be discussed below. We assume that the components are connected by a communication network, which will be discussed later.

The component models The behaviour of a root component x is quite simple. A root component sends a message to all components that are immediately connected to it. These components are called its *neighbours*. In this message, component x tells each neighbour component n_i, that n_i has distance 1 to a root, if n_i chooses x as its connection to a root. Figure 1 shows a model of the root component. It is a Petri net[1], where the place $init$ of x is initially marked. As usual in object oriented programming languages, the place $init$ of component x is denoted by $x.init$. There is only one transition, which adds one token to the interface place $message$ of each neighbour n_i of x. Note that the token representing the message is not the usual black token, but a structured token $(x, 1)$, which represents the contents of the message; the first element x of $(x, 1)$ represents the root's identity x, the

[1]Strictly speaking, this is not a Petri net because of the 'dots' in it and because of the dashed arcs, which do not have a meaning in Petri nets. We will use this notation for denoting the interface of a component, which defines the possible interconnections to other components. For simplicity, we present the interface of a component and its implementation in a single model; but, this could be easily separated from each other.

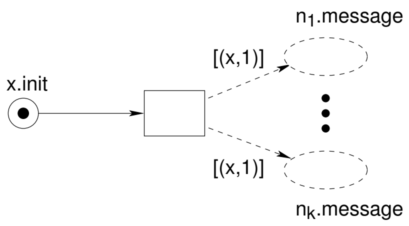


Abbildung 1: The root component x

second element represents the distance information for this neighbour. Up to now, we are considering each component separately. Hence, we do not know its concrete neighbours $n_1, \ldots, n_k$ when modelling the component. Therefore, we represent the corresponding arcs by dashed arrows, and some dots indicate that there can be several neighbours. The meaning of this is that, once we have define the architecture, each concrete instance of this component will be connected to the corresponding $message$ places[2] of its neighbours.

Next, we model the behaviour of an inner component[3] x. The inner component x waits for arriving messages. When it identifies a message from a better candidate for a parent (i. e. a candidate with a shorter distance to a root component), it chooses this candidate and stores this candidate along with its distance in its place $parent$. Initially, there is a token $(\perp, \omega)$, where $\perp$ denotes a not yet defined parent component and ω denotes infinity. Figure 2 shows the model of the inner component. The transition waits for a message

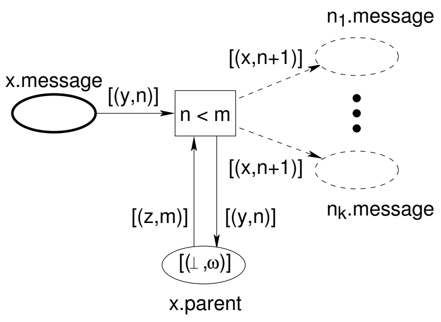


Abbildung 2: The inner component x

[2]Maybe, there are more appropriate notations for this. But, notation is an issue that strongly depends on the application area. Therefore, we concentrate on the underlying concepts in this paper.

[3]Note, that we use the same identifier x for the inner component as for the root component. The reason is a convention that always uses x for the component currently under consideration. We will see later on, that this convention gives us a simple transformation for obtaining the overall model of a component based system. Object oriented programmers may consider x as a key word like `this` or `self` in object oriented programming languages.

from some other agent y with a distance n that is less than its current distance m. This condition is represented by the transition guard $n < m$. In that case, the transition updates y as its new parent node with distance n in place $parent$. Moreover, it sends a message $(x, n+1)$ to all its neighbours $n_1, \ldots, n_k$. This message tells each neighbour that it could have distance $n+1$ from a root, if it chooses x as its neighbour. Sending these messages is modelled in the very same way as in the root component. Note that the place $message$ of an inner component is drawn with a bold face line. This indicates that this is an interface place on which other components may produce tokens.

Note that an inner component will never read or remove a message with a distance n that is worse than its current distance. This does not do any harm to our model, but it is a little sloppy. It would be easy to add another transition that removes these messages. For simplicity, however, we omit this transition from our considerations.

The architecture In our example, the architecture of a system is defined by a number of instances for each component and by their interconnection. This can be represented by simple a graph as shown in Fig. 3: There is one root component a and there are three inner components b, c, and d, which are connected in a ring. From this graph, we can easily construct the corresponding system. This is shown in Fig. 4, where we omit the names of places and the annotations of arcs for simplicity.

Analysis and verification Now, we could analyse or verify this concrete Petri net model. But actually, we can do much better: We can analyse and verify properties for all possible systems that can be built from the components. This is where DAWN comes in. First of all, we fold the concrete net from Fig. 4 to a single algebraic Petri net. This algebraic Petri net is shown in Fig. 5. Note that there are only three places $init$, $messages$, and $parent$ for all components now. In order to distinguish the identity of the components, we add a first element to each token, which identifies the component to which it belongs. Likewise, the labels of the arcs are equipped with a first element, which identifies the involved component when the transition fires. This first element is the variable x for all ingoing arcs of a transition. For the outgoing arcs to place $message$, it is the component to which a message is sent. The symbol R represents the root agent, the symbol I represents the initially undefined parents of the agents b, c, and d, and the function N represents the way, messages are passed on to other components. Note that the interpretation of these symbols comes from the architecture definition only.

It turns out, that the net itself is the same for all systems built from the components root and inner. The only part that changes is the definition of the meaning of the symbols R, I, and N. For each architecture definition, these symbols receive a new interpretation. Thus, the algebraic net form Fig. 5 along with the different interpretations of the symbols according to the architecture definition language, captures all systems that can be built from the components. What is more, DAWN provides techniques for verifying all these systems once and for all (architectures). The net from Fig. 5, for example, was verified in [KR97] – though not obtained from the component based approach.

Altogether, there are two ways of verification. On the one hand, we can verify a single

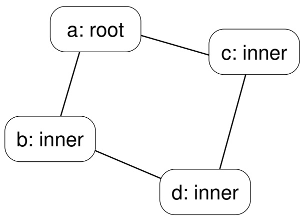


Abbildung 3: The architecture

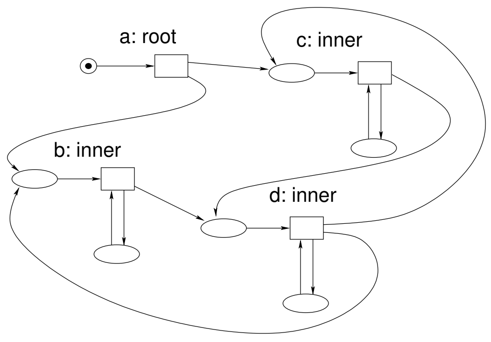
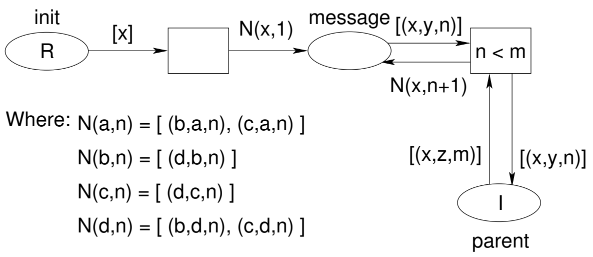


Abbildung 4: The system



Where: N(a,n) = [(b,a,n), (c,a,n)]

N(b,n) = [(d,b,n)]

N(c,n) = [(d,c,n)]

N(d,n) = [(b,d,n), (c,d,n)]

R = [a]

I = [(b,⊥,ω), (c,⊥,ω), (d,⊥,ω)]

Abbildung 5: The corresponding algebraic Petri net

concrete system, which is defined in terms of component models and a single architecture. Since models uniquely define a system, we can use model checking for doing automatic verification. On the other hand, we can do verification for a class of architectures (e. g. for all rings or for all fully connected graphs, etc.). To this end, we can use the support of automated theorem provers.

3 Conclusion

In this paper, we have illustrated how to model components and how to build systems from components by defining an architecture. The underlying techniques themselves are not new. Therefore, we did not formalize them, here. DAWN and its verification techniques have been introduced in [KR96, WWV^{+}97, Rei98]. The particular interpretation of x in the first element of the arc labels has been proposed by Desel [Des97] and has been further developed for modelling different communication paradigms [DK98, DK01].

The main contribution of this brief note is a new perspective that allows us to use these techniques for modelling and verifying component bases systems. Since the end model is the very same as in DAWN, we can use the very same techniques for doing all kinds of analysis and verification. We can use model checking for individual systems, which can be done fully automatically. Or we can use theorem provers for verifying properties for a complete class of architectures.

Obviously, the presented example is quite simple. In practice, architecture description languages are much richer. A careful investigation of a complete ADL with all its features is subject to future research and part of ongoing research. Moreover, a more detailed investigation of reusability of components in this setting needs further attention; we believe that classical Petri net results on refinement and abstraction can be reused to deal with this issue.

Literatur

[Des97] Jörg Desel. How distributed algorithms play the token game. In *Foundations of Computer Science: Potential – Theory – Cognition*, *LNCS* 1337. Springer, 1997.

[DK98] Jörg Desel and Ekkart Kindler. Proving Correctness of Distributed Algorithms Using High-Level Petri Nets – A Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 177–186, Fukushima, Japan, March 1998. IEEE Computer Society Press.

[DK01] Jörg Desel and Ekkart Kindler. Petri nets and components – Extending the DAWN approach. In D. Moldt, editor, *Workshop on Modelling of Objects, Components, and Agents*, pages 21–35, August 2001.

[Har87] David Harel. Statecharts: A Visual Formalism for Computer Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[KR96] Ekkart Kindler and Wolfgang Reisig. Algebraic System Nets for Modelling Distributed Algorithms. *Petri Net Newsletter*, 51:16–31, December 1996.

[KR97] Ekkart Kindler and Wolfgang Reisig. Verification of Distributed Algorithms with Algebraic Petri Nets. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential – Theory – Cognition*, *LNCS* 1337, pages 261–270. Springer, 1997.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifiying Distributed Architectures. In *Fifth European Software Engineering Conference, ESCE '95*, *LNCS* 989, pages 137–153. Springer, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes (Parts I & II). *Information and Computation*, 100(1):1–40 & 41–77, 1992.

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms — Modeling and Analysis with Petri Nets*. Springer, 1998.

[WWV$^+$97] M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, and J. Desel. DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, December 1997.