HAL
open science

A Process for Continuous Validation of Self-Adapting Component Based Systems

Viet Hoa Nguyen, François Fouquet, Noël Plouzeau, Olivier Barais

To cite this version:

Viet Hoa Nguyen, François Fouquet, Noël Plouzeau, Olivier Barais. A Process for Continuous Validation of Self-Adapting Component Based Systems. 7th International Workshop on Models@run.time of the MODELS 2012 Conference., Oct 2012, Innsbruck, Austria. hal-00764706

HAL Id: hal-00764706

https://inria.hal.science/hal-00764706v1

Submitted on 13 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Process for Continuous Validation of Self-Adapting Component Based Systems (Position Paper) *

Viet Hoa Nguyen
INRIA, Centre Rennes - Bretagne Atlantique
Rennes, France
viet-hoa.nguyen@inria.fr

Noël Plouzeau
IRISA, Université de Rennes1
Rennes, France
noel.plouzeau@irisa.fr

Francois Fouquet
IRISA, Université de Rennes1
Rennes, France
francois.fouquet@irisa.fr

Olivier Barais
INRIA, Centre Rennes - Bretagne Atlantique and
IRISA, Université de Rennes1
Rennes, France
olivier.barais@irisa.fr

ABSTRACT

In this paper we propose an approach to integrate the use of time-related stochastic properties in a continuous design process based on models at runtime. Time-related specification of services are an important aspect of component-based architectures, for instance in distributed, volatile networks of computation nodes. The models at runtime approach eases the management of such architectures by maintaining abstract models of architectures synchronized with the physical, distributed execution platform. For self-adapting systems, prediction of delays and throughput of a component assembly is of utmost importance to take adaptation decision and accept evolutions that conform to time specifications. To this aim we define a metamodel extension based on stochastic Petri nets as an internal time model for prediction. We design a library of patterns to ease the specification and prediction of common time properties of models at runtime and make the synchronization of behaviors and structural changes easier. Our prediction engine is fast enough to perform prediction at runtime in a realistic setting and validate models at runtime.

Keywords

model driven engineering, performance prediction, validation at runtime

*The research leading to the results stated in this paper has received funding from the European Community's Seventh Framework Program (FP7) under the grant of CHESS Artemis project.)

*Copyright 2012 ACM 978-1-4503-1802-0/12/10 ...$15.00.

1. INTRODUCTION

Component-based software design is now a well established approach for building reusable and trustable systems. In these systems, trust relies on precise specifications of component interfaces and on application of validation techniques on component implementations. Specifications include typing properties, behaviors and quantitative properties [4]. Soft real-time systems emphasize time related performance as a vital quality. Specifications for this kind of system rank response time and throughput as first class attributes of the expected system behavior. Designers therefore rely on quantitative analysis techniques to validate implementations against specifications. This validation task is mainly a design time activity, which provides prediction of quantitative properties for the system's needs and capabilities before these systems are deployed. Designers rely on these predictions to engineer an appropriate architecture that will meet the specifications, as long as a set of corresponding requirements remains valid at run time.

Soft real time systems such as Internet of Things systems (e.g. networks of smart sensors and personal digital assistants) are a particular class of component based systems, being highly flexible in their design and configuration. In this paper we focus on this category of systems that must support major architectural changes at runtime, without stopping but instead by hotdeploying. Such a kind of systems are sometimes named *eternal systems*. Architectural changes are a consequence of two main evolution causes: (1) changes of system's service definition; (2) changes of system's implementation. Changes of service definition include changes of systems specification stemming from changes of users' needs, for instance addition or removal of functionalities, changes in timing and throughput requirements, changes of preferences for power management on a mobile device, etc. Changes of service implementation include changes in resource availability from the supporting environment of the system, for instance fluctuations of network bandwidth, or addition of new computation nodes e.g. sensor equipped mobiles.

Internet of Things systems are large sets of computation nodes. These systems are opportunistic by design: for a given application, its execution platform is made of a continuously evolving set of computation nodes, with very diverse computing power and communication capabilities. For example, a real time cooperative social system can connect users that share geographical properties (e.g. cycling in the same city). As users move and switch activities, their personal digital assistants frequently connect to and disconnect from the social network, while their communication capabilities fluctuate rapidly[16].

Such systems must be able to reconfigure on the fly, and even self reconfigurable in real time, without requiring a restart after reconfiguration. These systems indeed have specific architectural features, and their design techniques are an active research topic [5]. However, flexibility should not be implemented at the expense of loss of trust in the correctness of these adaptive systems. As these systems' designs evolve continuously without human supervision and intervention, they must also implement self validation without human intervention. Therefore an autonomous self-validation subsystem must be present in the self-adapting system.

In this paper we introduce a design and validation process for on the fly prediction of time related extra-functional properties. Our approach is three-fold:

1. We integrate stochastic colored Petri nets as a metamodel extension for specifying time-related component properties (Section 2.4).

2. We set up a library of frequently used patterns to help designers to superimpose timed behavior descriptions on functional models (Section 2.5). These patterns also map timing evaluation results back into higher level system models.

3. We provide an integration of timing evaluation tools in the models at runtime paradigm (Section 3), with an evaluation time compatible with rapid changes in the architecture.

2. OUR APPROACH

2.1 Background

Our work targets highly dynamic software architectures that rely on the models at runtime technique (M@R for short)[14]. At the heart of M@R lies the capability to evaluate architecture variants continuously while the system is running, together with the ability to analyze these architectures in real time before deploying them, while the current architecture is running. Most applications need to master performance, in order to offer adequate quality of service. In the case of applications of the Internet of Things category (IoT), computing and communication resources are scarce, and therefore it is important to tune architectures wisely and to look for optimality of configurations, even while the underlying platform itself is changing its topology and capacities constantly.

As a practical example of highly flexible system we are developing a real time multi-user tactical decision platform for firefighters. Our experimentation configuration includes data processing nodes in vehicles, hand held tablets for commanding officers on the field, and sensors in personal protective equipment of firefighters. This platform helps us to evaluate our approach in a real life situation. We are designing most layers from scratch, from the distributed dynamic component model up to the computer supported cooperation application. Emergency situations evolve at a fast pace. Therefore our system must scale up as more and more teams are dispatched and arrive on site (resource driven changes). Moreover, user needs evolve on the field when commanding officers reorganize task forces (user needs driven changes). Similarly, communication needs and means evolve continuously, as our system deploys various wireless networks to maintain connectivity between its computation nodes. In this paper our sample example is taken from this experimentation architecture for firefighters.

2.2 General approach

Our approach builds on the models at runtime one, as we provide extension means to manage time related stochastic properties (e.g. average delay and throughput, worst case execution time). More precisely, we rely on structural models at runtime [14] superimposed with high level design patterns of timed behavioral descriptions. The *monitor, analyze, plan and execute* adaptation techniques (MAPE) operate at the platform level, while models at runtime support higher abstraction levels. MAPE and timed models at runtime are complementary: using our time extension, MAPE can use models at runtime time related properties to reason on models before deployment. Reciprocally, estimates computed at abstract level by prediction algorithms for models at runtime can be checked against real life values gathered by monitoring the platform after execution of the hot deployment plan.

However, the true power of models at runtime comes from the use of prediction: when the current architecture does not fulfill its goals, alternative architectures have to be generated and evaluated. Prediction algorithms can help to evaluate the quantitative properties of these architectures. These algorithms are often specialized and take specific partial models as inputs and outputs, leading again to the problem of mapping between the architecture model and the specialized prediction models used by the tools.

Our design process aims at combining specific quantitative prediction techniques with models at runtime. To this aim we rely on metamodel extensions to the Kevoree component metamodel [6, 1]. These extensions support description of timed behaviors using design patterns of colored stochastic Petri nets. The extension to augment component models with colored stochastic colored Petri nets in shown on Figure 1. These behaviors can be bound to component ports (required or provided), they can be bound to operations from the same component specification, or on operations in an assembly of component instances.

Our evaluation tool chain is managed as a virtual platform: using model at runtime we generate a configuration for this virtual platform, which operates with a simulated time scale. In a few seconds we can get performance results that would require long executions on a real platform. The use of a virtual platform improves the evaluation process by easing its

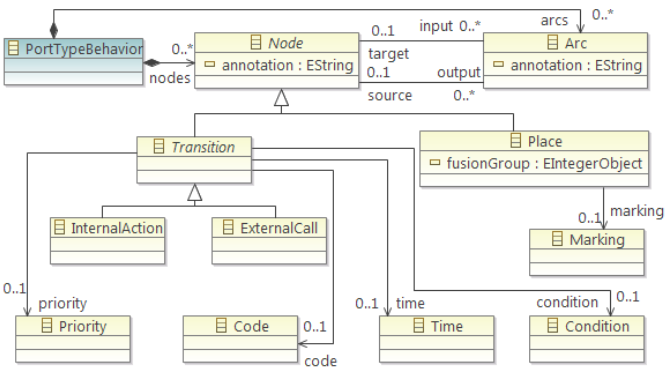

Figure 1: CPN extension to the Kevoree metamodel

integration into the global *models at runtime* architecture.

The models for the virtual platform use a specific platform metamodel that eases the transformation from timed component models to simulation models for the CPN Tools software [9]. This virtual platform provides estimates, for predicting time properties and checking them against specifications, and it is also able to generate abstract monitors that can be injected into the running system at the platform level. In this paper we will address the prediction features only.

2.3 A sample example

In the rest of this paper we present a simplified excerpt from our experimental platform as a sample example to illustrate the performance prediction principles. This example describes a network of sensors that monitors the temperature outside firefighters' personal protective equipment. Periodically, these sensors send data to a remote server, with a typical period of 1 second. Sensors are connected to the application platform through a combination of XBee mesh networks, Ethernet and 3G networks. We abstract communication between a sensor and the platform with a simple channel of the broadcast type, as shown in Figure 2. We focus here on the following quantitative properties: transmission delay of data between sensors and server, packet loss rate, throughput of incoming data in the notifier component. Our sample configuration is made of 3 sensors that periodically send information about the most recent updated temperature to a first server, named server #1 (or TemperatureNotifier component) on the centre_node. This server will then check that the temperature is in the acceptable interval; if it is greater than 80 or less than 5 than server #1 will transfer this information to server #2 (or TemperatureProcessor, on the centre_node), which monitors abnormal conditions for the whole firefighter's team.

The Kevoree component model in Figure 2 defines a system that includes three sensor components, a group communication channel, a data log component (TemperatureNotifier) and a temperature monitoring component. Typically the sensor Kevoree components are hosted on low power nodes, while data log and monitoring are deployed in a field computation cloud, usually on a compact ARM based node.

To describe the timed stochastic behavior of this system, this Kevoree architectural model is transformed automatically into a colored Petri net model shown in Figure 3. This model defines the semantics of timed stochastic behaviors for the sensor components, including the stochastic generation of values and the exponential distribution of transmission time. This stochastic generation added into the model can be seen as the usage model or the open workload of the system model.

Channel type definition. In Kevoree channel types are fully user definable, and they carry the binding semantics. The Kevoree model transformation process must map each channel instance to a colored Petri net. Channel instances are developed by fragments, one fragment on each node that connects to the channel using the FIFO semantic. This means that some *ChannelFragment* communicate with local ports, and other *ChannelFragment* of remote ports. As shown in Figure 3 there are four instances of the *Channel 1* implemented on each sensor node and on server node, because all three sensors and the server node contain components that connect to the channel. *Channel 1* is a broadcast channel, therefore, as we can see in Figure 4, the CPN model of the *Channel 1* instance on the *node 1* derived from the component model using a *Broadcast Channel Pattern*, which will be explained below. We can see that this instance of *Channel 1* has only two interfaces, which are *Arrive* and *Channel1* places because the channel instance on *node 1* has only one required port on server connect to. The transmission delay between *Node 1* and *centre_node* is expressed by the distribution function: *expTime(10)* marked on the *toServer* transition.

Figure 5 defines the behavioral view of the TemperatureNotifier component. This component processes input packets with a thread pool and a FIFO queue. Upon processing a packet the TemperatureNotifier sends it to the TemperatureProcessor if the data it contains indicates that the sensor's temperature is out of bounds. Figure 4 shows the internal model of the channel, which transfers packets to the component. The TemperatureProcessor logs abnormal temperatures received from the TemperatureNotifier.

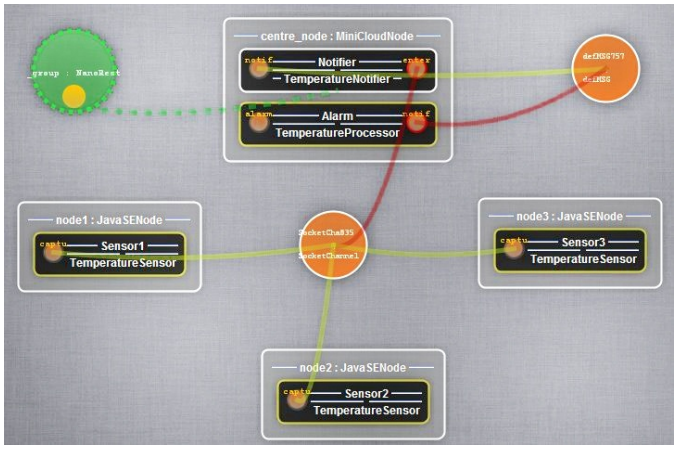

Figure 2: Architecture of the system using the Kevoree graphical notation

2.4 Injection of performance indicators

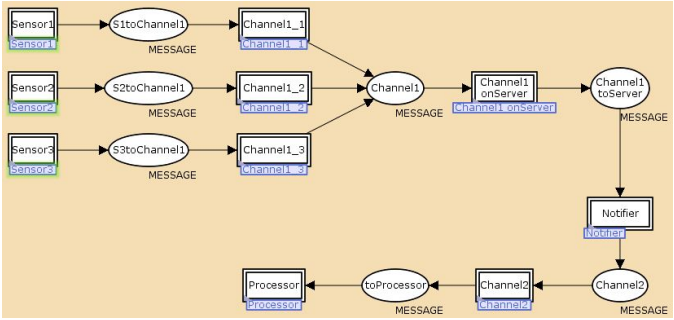

Figure 3: CPN toplevel model generated by Kermeta from Kevoree model

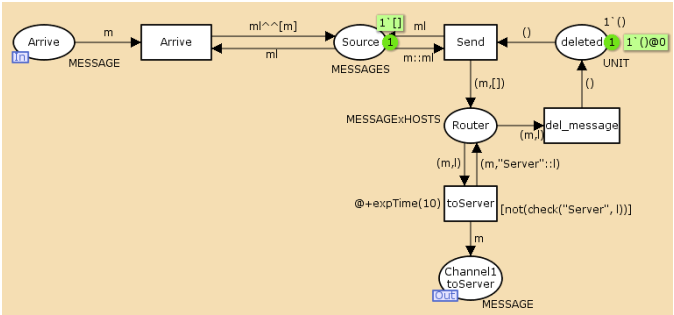

Figure 4: The CPN model after the transformation of the *Channel 1* instance implemented in node 1

Following separation of concerns principles, we separate performance indicators from service definition and implementation. Service definition consists in interfaces bound to ports of component types. Service implementation is either source code or assembly of component instances connected by instances of connector types (our Kevoree component metamodel supports user defined connector types with arbitrary communication and coordination properties). Component types designers use instances of our DPPMmetamodel to specify behavior and performance properties.

Definition of performance indicators. In our approach component developers describe quantitative properties of their component implementation by providing an abstract model of its behavior, including timing properties. We use colored stochastic Petri nets (CPN for short) to define abstraction. CPNs include time variables and distribution functions to define stochastic behaviors.

CPNs are used as a description of a component timed behavior, both for behaviors known in advance and for the definition of performance indicators where time variables are unknown in advance and must be solved by simulation. These two uses of CPNs correspond to two different activities of designers in component based systems:

- when defining a component type implementation, a designer provides sufficient definition of the implementation properties;
- when assembling components a designer must know the behavior of these components and the behavior of the assembly.

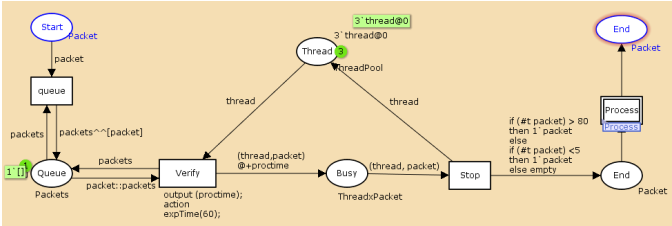

Figure 5: Behavioral view of TemperatureNotifier component

Monitor concept. A monitor serves this dual purpose of specification and simulation, they are constructed from stochastic colored Petri nets extended with measurement and simulation controls. Our tool chain uses the CPN Tools software [9] to implement this monitor notion. Monitors can observe, inspect, alter a running simulation of a CPN model, and periodically extract information from the markings and binding elements during the simulation, to use the information for different purposes (performance measuring for example). We have adapted the existing Access/CPN Eclipse plug-in to transmit simulation results on the fly while the simulation is running. Thanks to the definition of specific monitors, we can forward simulation results to specific Kevoree components and therefore provide the timing properties of a given architectural model to the models at runtime engine.

For example, Figure 6 represents a simple assembly of two components A and B, with a black transition to denote a message send. Once a component assembly has been made, the reasoning engine must select which global non-functional properties need to be measured, before proceeding to generation of the CPN model of the system derived from the component model. In Figure 6 example, the reasoner needs an estimate for the time elapsed between removal of a token from place P0 and arrival of a token into place P2, in order to compute the end-to-end response time. From a simulation point of view, we control the update of variables on monitor instances by defining a color as a record that contains a field named AT, which represents the arrival time of the user request. Our monitor composition engine generates a monitor instance with this color, in order to measure the time elapsed between places P1 and P2. This monitor associated with the T2 transition, when the T2 transition occurs, the response time can be calculated by subtracting the value of the AT field for the request from current model time.

2.5 Separation of concerns through patterns

While stochastic colored Petri nets are a powerful means of timed behavior specification, they are too fined grained to be used directly by designers of component based systems. Timing concerns are design concerns that can be managed more easily using patterns, to promote separation of timing concerns and ease reuse of timing specifications. In [15] the authors have proposed a set of empirical design patterns for modeling process-aware information systems, communication protocols, embedded systems, distributed systems,

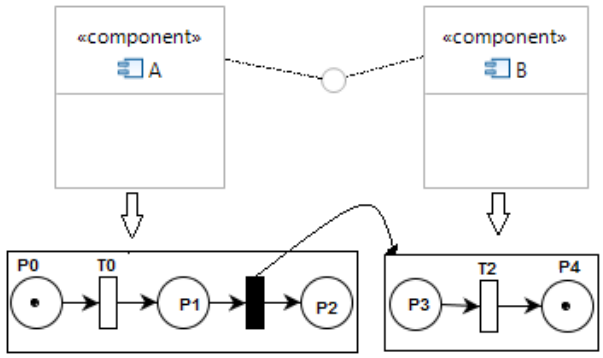

Figure 6: Example of a simple system made of a two components assembly

etc. We have applied this notion to prepare a set of frequent timed behavior specifications in the form of templates for Kevoree component types and channel types.

A template is a CPN abstracted as a single transition with parameters. Just like the well-known design pattern concept, each of CPN pattern addresses a specific timed behavior need. For instance, the *broadcast channel pattern* defines a parameterized time behavior useful for channel types that have a one to many message transmission semantics. Figure 7 describes the structure of the CPN template when the receiver set size N is 3.

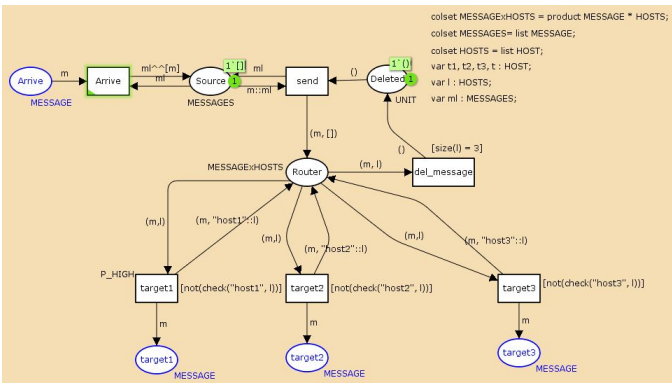

Figure 7: Broadcast Channel Pattern.

In this pattern model, we have $N + 1$ interface places, represented in blue, and a generic token type named Message. The internal token types (color sets) are declared in the top left part of Figure 7.

Using the Kermeta model transformation language we instantiate the *broadcast channel pattern* from the Kevoree model. To allow such transformation we have extended the Kevoree metamodel with a specific metamodel extension that provide support for pattern definition and reference in component models.

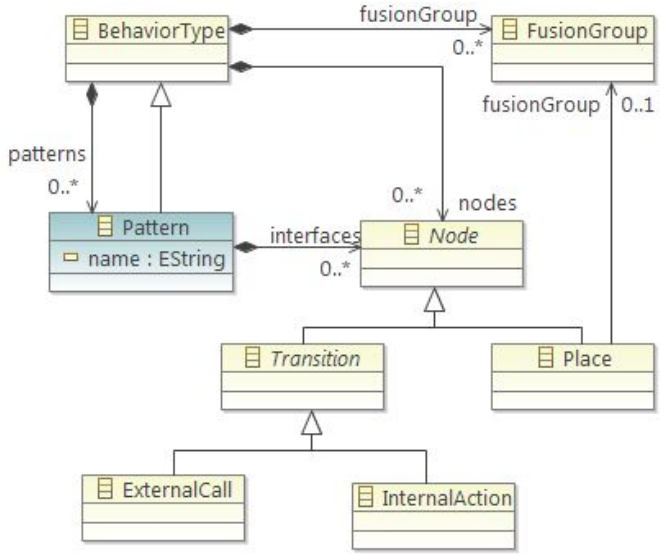

Figure 8: Metamodel extension for CPN patterns

3. SUITABILITY FOR MODELS AT RUNTIME

Our process has been designed to offer a balance of power of prediction and prediction computation time. The tool chain consists in a set of Kermeta compiled transformations, the CPNtools external analysis software (packaged as an Eclipse compatible plugin), a set of generated monitors created from the patterns instantiated in the Kevoree model and Kevoree wrapper components that interface with the models at runtime engine. Running the tool chain on a Kevoree model of dozen of components produces useful timing predictions in less than 10 seconds, the longest computation step being the parsing of the CPNTools model by the external CPN/Access plugin [9]. Running the toolchain on an ARM based node with one GB of memory is possible. In our current pervasive system configuration, we use this kind of node as support nodes that perform this kind of *models at runtime* computation.

4. RELATED WORK

In this paper we described how CPN models can be used for component based systems, using the Petri nets as an abstraction of control flow through the component. In this regard Petri nets play a role similar to Service Effect Specifications (SEFFs) in [3]: they describe how a provided service of a component calls its required services at some level of abstraction. In [2], the resource demanding SEFFs (RDSEFFs) have been introduced for performance prediction. The RDSEFFs describe dependencies between required and provided services of a component. As Finite State Machines (FSM) are insufficient for quality of service analysis, they can be extended with stochastic information and QoS characteristics (such as execution time) to make them analyzable for QoS properties. Paper [8] uses stochastic Petri nets to model SEFFs for QoS analyses. Paper [13] uses stochastic regular expressions to model SEFFs. Paper [12] uses annotated UML 2.0 activities as SEFF models. In [8] the authors give the necessary extension of RDSEFF with Stochastic Petri nets to model multithreaded systems. This helps to

detect resource conflicts in considering the influence of concurrency on Quality of Service attributes. Using stochastic Petri nets we can model competition for resources, e.g. thread pool resources, which has to be considered in a performance prediction model for distributed systems. As in our approach, [7] relies on CPNs to describe behavioral aspects of software architectures. The authors apply quality models to evaluate security, efficiency and reliability with CPN descriptions. Interfaces of components are modeled as colored places, components and connectors are represented with CPNs, but [7] does not support hierarchical components. While [7] supports non-functional aspects by attaching them to tokens, it does not address generation of monitors to measure these non-functional aspects. Composition of token colors is not managed for component assembly, which is an important limitation for the dynamic component architectures that we address. In [10], the authors show how Queuing Petri Net models (QPN models) can be used to predict performance of distributed component-based systems. QPN models describe hardware and software aspects of system behavior, such as hardware contention, scheduling strategies, software contention, resource possession, and synchronization, blocking, synchronization processing. The active resources are usually modeled using queuing places and passive resources such as threads, processes are modeled using tokens inside ordinary places. The interaction between components are described by connecting component transitions that are modeled as QPN places. The composite transactions are modeled using composite tokens, so that a separate token color is used for every sub-transaction. In this paper, the author assumed that the total service demand of a transaction at a given system resource is spread evenly over its sub-transactions. The modeling of the system may become much more complicated if one would not like to apply this assumption (this would sound more realistic). Paper [11] presents a research roadmap aiming to implement intelligent techniques for self-aware performance and resource management. The authors aim at making their PCM architecture performance models usable at runtime. They propose a process based on an online performance query mechanism for retrieving and combining the models of all involved services into a single architecture-level performance model aiming to answer performance-related queries arising during operation. This process combine two main steps : the first uses model composition techniques to make an architecture-level performance model and the second one is an automatic model-to-model transformation to generate the target predictive model (layered queuing network, queuing Petri nets) from the architecture level performance model.

5. CONCLUSION AND FUTURE WORK

We have presented the principles we use to perform prediction of time related properties for the models at runtime approach. We have experimented with a separation of concern strategy based on specific design patterns for the time dimension, together with automated model transformation. Our first results show that this approach is implementable and useful in the context of adaptive, dynamic pervasive systems that include some reasonably powerful nodes as servers for models at runtime management. We are currently building a larger experimental setup to evaluate the scalability of the approach and check our pattern library against adaptation algorithm for large distributed systems.

6. REFERENCES

[1] Kevoree web site. http://www.kevoree.org.

[2] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, pages 54–65. ACM, 2007.

[3] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82:3–22, January 2009.

[4] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[5] B. e. Cheng. Software engineering for self-adaptive systems: A research roadmap. In B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2009.

[6] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE '12, pages 135–144, New York, NY, USA, 2012. ACM.

[7] K. Fukuzawa and M. Saeki. Evaluating software architectures by coloured petri nets. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 263–270, New York, NY, USA, 2002. ACM.

[8] J. Happe and V. Firus. Using stochastic petri nets to predict quality of service attributes of component-based software architectures. In *Proceedings of the Tenth Workshop on Component Oriented Programming (WCOP2005)*, volume 94. Citeseer, 2005.

[9] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. In *International Journal on Software Tools for Technology Transfer*, page 2007, 2007.

[10] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Trans. Softw. Eng.*, 32:486–502, July 2006.

[11] S. Kounev, F. Brosig, N. Huber, and R. Reussner. Towards self-aware performance and resource management in modern service-oriented systems. In *Proceedings of the 2010 IEEE International Conference on Services Computing*, SCC '10, pages 621–624, Washington, DC, USA, 2010. IEEE Computer Society.

[12] H. Koziolek, J. Happe, and S. Becker. Parameter Dependent Performance Specifications of Software Components. In *Proceedings of the 2nd International Conference on Quality of Software Architectures (QoSA)*, 2006.

[13] H. Koziolek and R. Reussner. A model transformation from the palladio component model to layered queueing networks. In *Proceedings of the SPEC international workshop on Performance Evaluation: Metrics, Models and Benchmarks*, pages 58–78, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.

[15] N. Mulyar and W. M. van der Aalst. Towards a pattern language for colored petri nets, 2005.

[16] F. Rahimian, T. Nguyen Huu, and S. Girdzijauskas. Locality-awareness in a peer-to-peer publish/subscribe network. In K. GÃűschka and S. Haridi, editors, *Distributed Applications and Interoperable Systems*, volume 7272 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 2012.