A model-based method to manage time properties in component based software systems

Viet Hoa Nguyen

ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Viet-Hoa Nguyen

Préparée à l'unité de recherche
INRIA – Centre Rennes Bretagne Atlantique
Institut National de Recherche en Informatique et Automatique

---

Une méthode fondée sur les modèles pour gérer les propriétés temporelles des systèmes à composants logiciels

Thèse soutenue à Rennes le 5 Décembre 2013

devant le jury composé de :

Franck BARBIER
Professeur à l'Université de Pau / *rapporteur*

Philipe COLLET
Professeur à l'Université de Nice / *rapporteur*

Pascal POIZAT
Professeur à l'Université de Paris Ouest Nanterre / *examinateur*

Antoine BEUGNARD
Professeur à l'ENST de Bretagne / *examinateur*

Jean-Marc JEZEQUEL
Professeur à l'Université de Rennes 1 / *directeur de thèse*

Noël PLOUZEAU
Maître de conférences à l'Université de Rennes 1 / *co-directeur de thèse*

Abstract

This thesis proposes 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 computer 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 the 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. Furthermore, we apply the approach of Aspect-Oriented Modeling to weave the internal time models into timed behavior models of the component and the system. 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

Résumé

Cette thèse propose une approche pour intégrer l'utilisation des propriétés temporisées stochastiques dans un processus continu de design fondé sur des modèles à l'exécution. La spécification temporelle de services est un aspect important des architectures à base de composants, par exemple dans des réseaux distribués volatiles de nœuds informatiques. L'approche *models@runtime* facilite la gestion de ces architectures en maintenant des modèles abstraits des architectures synchronisés avec la structure physique de la plate-forme d'exécution distribuée. Pour les systèmes auto-adaptatifs, la prédiction de délais et de débit d'un assemblage de composants est primordial pour prendre la décision d'adaptation et accepter les évolutions qui sont conformes aux spécifications temporelles. Dans ce but, nous définissons une extension du métamodèle fondée sur les réseaux de Petri stochastiques comme un modèle temporisé interne pour la prédiction. Nous concevons une bibliothèque de patrons pour faciliter la spécification et la prédiction des propriétés temporisées classiques de modèles à l'exécution et rendre la synchronisation des comportements et des changements structurels plus facile. D'autre part, nous appliquons l'approche de la modélisation par aspects pour tisser les modèles temporisés internes dans les modèles temporisés de comportement du composant et du système. Notre moteur de prédiction est suffisament rapide pour effectuer la prédiction à l'exécution dans un cadre réaliste et valider des modèles à l'exécution.

Mots clés: Ingénierie Dirigée par les Modèles, Prédiction de Performance, Validation à l'exécution.

Acknowledgements

First of all, I would like to sincerely thank my advisors, Jean-Marc Jézéquel and Noel Plouzeau, for accepting me as your student and giving me the opportunity to pursue my Doctoral studies and working with you and within the Triskell group, one of the best team in the world in the MDE field, and also for guiding and supporting me over the years.

I would also like to thank Prof. Franck BARRIER and Prof. Philippe COLLET for spending time to review my thesis. Their comments and suggestions helped me to significantly improve my thesis. I would like also thank to my other thesis committee members, Prof. Pascal POIZAT and Prof. Antoine BEUGNARD for examining my thesis.

Special thanks go to the other members of the Triskell group for providing support and friendship that i needed, with whom i have shared many good moments and interesting discussions. I would also like to thanks all my Vietnamese friends for helping me and staying beside me in difficult times. I am indebted to them for their help.

Finally, i must express my gratitude to my family for their continued support and encouragement, for staying beside my all of the ups and downs of my research.

Résumé en français

0.1 Introduction

La conception des logiciels à composants est aujourd'hui une approche bien établie pour la construction de systèmes réutilisables et fiables. Dans ces systèmes, la confiance repose sur les spécifications précises des interfaces des composants et sur l'application des techniques de validation sur des implémentations de composants. Les spécifications comprennent des propriétés de type, des comportements et des propriétés quantitatives [19]. Un système temps réel souple souligne la performance liée au temps comme une qualité essentielle. Les tâches de spécification de ce genre de système classent le temps de réponse et le débit comme des attributs les plus importants du comportement attendu du système. Les architectes en logiciel s'appuient donc sur des techniques d'analyse quantitative pour valider les implémentations par rapport aux spécifications. Cette tâche de validation est principalement une activité faite au moment de la conception, et qui fournit une prédiction de propriétés quantitatives pour les besoins et les capacités du système avant que ces systèmes soient déployés. Les concepteurs comptent sur ces prédictions pour concevoir une architecture appropriée qui répond aux spécifications, tant qu'un ensemble d'exigences correspondantes reste valable au moment de l'exécution. Les systèmes dits à temps réel souple tels que les systèmes de l'internet des objets (par exemple les réseaux de capteurs intelligents et assistants numériques personnels) sont une classe particulière de systèmes à base de composants, car étant très flexible dans leur conception et configuration. Dans cette thèse, nous nous concentrons sur cette catégorie de systèmes qui supporte les changements architecturaux au moment de l'exécution, sans s'arrêter pour se redéployer, mais en effectuant un redéploiement à chaud. Ce genre de système est parfois nommé *système éternel*. Le changement architectural est une conséquence de deux causes principales d'évolution : (1) les changements de la définition de service du système ; (2) les changements de l'implémentation du système. Les changements de la définition de service comprennent les changements de spécification des systèmes du fait de changements des besoins des utilisateurs, par exemple, la suppression ou l'addition de fonctionnalités, des changements

de l'exigence de débit et de temps, les changements de préférences pour la gestion d'énergie sur un appareil mobile, etc. Les changements de l'implémentation de service comprennent des changements dans la disponibilité des ressources de l'environnement de soutien du système, par exemple la fluctuation de la bande passante du réseau ou l'addition de nouveaux nœuds de calcul, par exemple les dispositifs mobiles équipés avec les capteurs. Les systèmes de l'internet des objets sont de grands ensembles de nœuds de calcul. Ces systèmes sont opportunistes par conception: pour une application donnée, sa plate-forme d'exécution se compose d'un ensemble de nœuds de calcul en continuelle évolution, avec une puissance de calcul et des capacités de communication très diverses. Par exemple, un système social coopératif en temps réel peut connecter les utilisateurs qui partagent des propriétés géographiques (par exemple le cyclisme dans la même ville). Lorsque les utilisateurs se déplacent et changent d'activité leurs assistants personnels numériques se connectent et se déconnectent fréquemment du réseau social, tandis que leurs capacités de communication fluctuent rapidement [93]. Ce genre de système doit être en mesure de se reconfigurer à la volée, souvent en temps réel et de manière autonome, sans nécessiter un redémarrage après la reconfiguration. Ces systèmes ont des caractéristiques architecturales spécifiques, et leurs techniques de conception sont un sujet de recherche actif [27]. Toutefois, la flexibilité ne devrait pas être implémentée au détriment de la perte de fiabilité dans la justesse de ces systèmes adaptatifs. Comme la conceptions de ces systèmes évolue continuellement sans supervision ni intervention humaine, ils doivent aussi mettre en œuvre une auto-validation sans intervention humaine. Par conséquent, un sous-système autonome d'auto-validation doit être présent dans le système auto-adaptatif. Dans cette thèse, nous introduisons un processus de conception et de validation pour la prédiction à la volée des propriétés extra-fonctionnelles liées au temps. Notre approche est triple :

1. Nous intégrons les réseaux de Pétri colorés stochastiques comme une extension du métamodèle pour spécifier les propriétés liées au temps sur les composants.
2. Nous mettons en place une bibliothèque de patrons fréquemment utilisés pour aider les concepteurs à superposer des descriptions de comportement temporisé sur des modèles de services fonctionnels.
3. Nous fournissons une intégration des outils d'évaluation de temps dans les modèles à l'exécution, avec un temps d'évaluation compatible avec des changements rapides dans l'architecture.

0.2 Contributions

Notre approche s'appuie sur le paradigme des *models@runtime* qui emploient à l'exécution des modèles architecturaux. Nous fournissons des extensions pour gérer les propriétés stochastiques liées au temps (par exemple le délai moyen et

le débit, et le pire cas de temps d'exécution). Plus précisément, nous nous appuyons sur des modèles structurels à l'exécution [82] superposés avec des patrons de conception de haut niveau fournissant des descriptions comportementales temporisées. Les techniques d'adaptation reposant sur les principes *surveiller, analyser, planifier et exécuter* (MAPE) fonctionnent au niveau de la plate-forme, tandis que les modèles à l'exécution supporte des niveaux d'abstraction plus élevés. Le MAPE et les modèles à l'exécution sont complémentaires : en utilisant notre extension d'expression temporelle de propriétés, le MAPE peut utiliser des propriétés liées au temps des modèles à l'exécution pour raisonner sur les modèles avant leur déploiement. Réciproquement, les estimations calculées au niveau abstrait par des algorithmes de prédiction pour les modèles à l'exécution peuvent être comparées aux valeurs réelles obtenues en surveillant la plate-forme après l'exécution du plan de déploiement à chaud.

Cependant, la véritable puissance des modèles à l'exécution provient de l'utilisation de la prédiction : quand l'architecture actuelle n'atteint pas l'objectif, les alternatives architecturales doivent être produites et évaluées. Les algorithmes de prédiction peuvent aider à évaluer les propriétés quantitatives de ces architectures. Ces algorithmes sont souvent spécialisés et prennent des modèles spécifiques partiels comme les entrées et les sorties, ce qui conduit à nouveau au problème de la correspondance entre le modèle architectural et les modèles de prédiction spécialisés qui sont utilisés par les outils. Notre processus de conception vise à combiner des techniques spécifiques de prédiction quantitative avec des modèles à l'exécution. Pour cela nous nous appuyons sur des extensions du métamodèle de composants Kevoree [1, 39]. Ces extensions supportent la description de comportements temporels à l'aide de patrons de conception de réseaux de Petri stochastiques colorés. L'extension pour augmenter modèles de composants avec les réseaux de Petri colorés est représentée sur la Figure 3.6. Ces comportements peuvent être associés aux ports de composants (pour des services requis ou fournis). Ils peuvent être liés aux opérations à partir de la même spécification de composant, ou sur les opérations dans un assemblage d'instances de composants.

0.2.1 Les patrons et le modèle de performance des composants Kevoree

Dans les sections précédentes, nous avons indiqué que la modélisation du comportement des composants logiciels et des systèmes logiciels en terme de réseaux de Petri colorés s'est avéré être une bonne plate-forme pour capturer des informations critiques dans les systèmes temps réel, réactifs, concurrents et distribués. Toutefois, dans le développement des logiciels modernes, il n'est pas évident que les réseaux de Petri colorés soient familiers aux concepteurs. En conséquence, nous avons défini un ensemble de patrons pour modéliser les réseaux de Petri colorés, dans le but de faciliter le travail des développeurs. Une bibliothèque de patrons peut être établie à partir des expériences acquises en matière de modélisation. Les développeurs utilisent souvent des patrons dans la bibliothèque

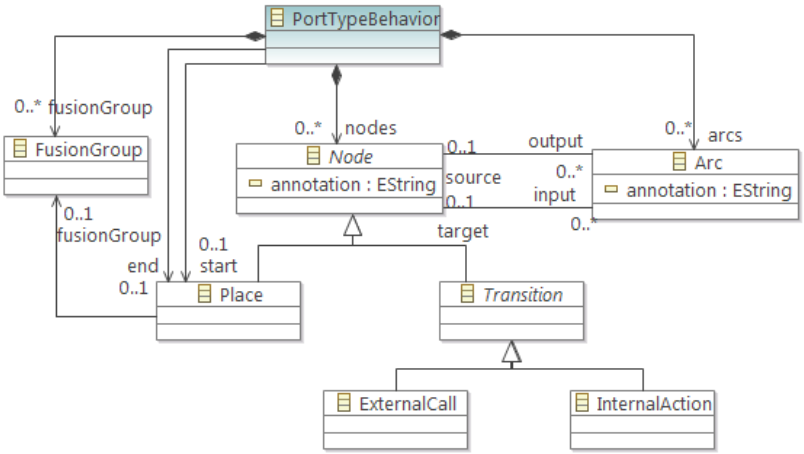

Figure 1 – Behavior model for Kevoree

pour construire leurs modèles de manière efficace, tout en évitant de réinventer des solutions déjà existantes pour résoudre leurs problèmes. Dans [86], les auteurs ont proposé un ensemble de 34 modèles de conception empiriques pour la modélisation de systèmes d'information fondés sur les processus et les protocoles de communication pour des systèmes embarqués distribués. Ces modèles servent à résoudre les problèmes qui apparaissent lors de la modélisation au moyen de réseaux de Petri colorés, et ils ont été documentés dans un format qui permet aux concepteurs de comprendre facilement et appliquent ces modèles dans leurs propres problèmes. Les concepteurs doivent déterminer les propriétés recherchées puis construire leurs modèles CPN en utilisant ces patrons.

Pour simplifier le travail des développeurs, nous proposons d'employer des concepts d'interfaces et de paramètres de patrons. Les interfaces d'un patron sont des transitions et des places qui peuvent être connectés avec des transitions et des places externes. Les paramètres du patron sont des informations qui caractérisent une instance du patron. Ces paramètres peuvent être des valeurs de jetons ou le nom de la transition externe (le service requis). En conséquence, l'application d'un patron donné n'est pas faite par copier-coller d'un modèle existant mais par la mise en correspondance des interfaces des patrons avec les éléments existants et la définition des valeurs des paramètres. Ce procédé est analogue aux techniques de conception par aspects. En utilisant le langage de transformation de modèle Kermeta, nous instancions les patrons utilisés à partir du modèle de comportement du composant Kevoree. Dans cette thèse, nous montrons des exemples de patrons pour modéliser les services synchronisés, les chaînes de diffusion de Kevoree.

0.2.2 Composition des modèles

Cette section explique comment composer les modèles de réseaux de Petri colorés des services de composants, en utilisant l'outil de transformation de modèles

Kermeta. Le résultat de la transformation produit un modèle du système entier qui peut être ensuite être simulé par l'outil CPNtools. Grâce à la similarité de la sémantique de notre métamodèle et de l'outil CPNTools, nous n'avons besoin de considérer que deux problèmes de composition des modèles : comment correspondre nos patrons à la sémantique de l'outil CPNTools, et comment gérer les éventuels conflits de la déclaration de variables dans les différent modèles réseaux de Pétri. Une fois les modèles composés, une transformation modèle vers texte est ensuite exécutée pour construire un fichier .cpn qui est conforme au fichier DTD de l'outil CPNTools. La figure 3.27 illustre le processus de transformation du modèle de composant Kevoree vers un modèle CPN du système entier prêt à être simulé.

Comme indiqué dans les sections précédentes, lors d'une adaptation la reconfiguration ou la génération de nouveaux modèles des composants Kevoree du système sont réalisées par le système d'analyse d'adaptation. Lors d'une adaptation un modèle est produit à l'exécution par les algorithme d'adaptation puis évalué par notre système pour déterminer les propriétés quantitatives de ce modèle. Le modèle produit doit être conforme métamodèle Kevoree étendu avec notre extension fondée sur les réseaux de Petri. La deuxième étape (# 2) du procédé de transformation est illustrée à la figure 3.27, qui est la transformation du modèle CPN généré dans l'étape (#1) du système entier à un fichier XML qui est compréhensible par l'outil CPNTools. Ce fichier XML est conforme à la *Document Type Definition* (DTD) de l'outil CPNTools. Dans cette étape de transformation de modèle vers texte, nous pouvons appliquer d'autres tâches de post-traitement, par exemple l'intégration d'un algorithme de mise en page pour redessiner les éléments de réseaux de Petri.

Le processus de transformation contient deux types différents de transformations :

1. modèle-à-modèle : la transformation d'un modèle conforme à notre métamodèle d'extension vers un modèle global conforme au métamodèle de réseaux de Pétri colorés compatible avec CPNtools ;
2. modèle-texte : la transformation du précédent modèle vers un fichier XML valide que l'outil CPNtools comprend.

0.3 Conclusion

Cette thèse présente une solution pour rendre prévisible les caractéristiques de performance d'architectures distribués adaptatives . Le travail de cette thèse est fondé sur les principes suivants. Tout d'abord, l'approche proposée repose sur une synthèse automatique des propriétés de performance du système à partir des propriétés des composants correspondants. Deuxièmement, l'approche permet la séparation des préoccupations de propriétés fonctionnelles et extra-fonctionnelles des composants et du système. Troisièmement, elle permet une analyse des per-

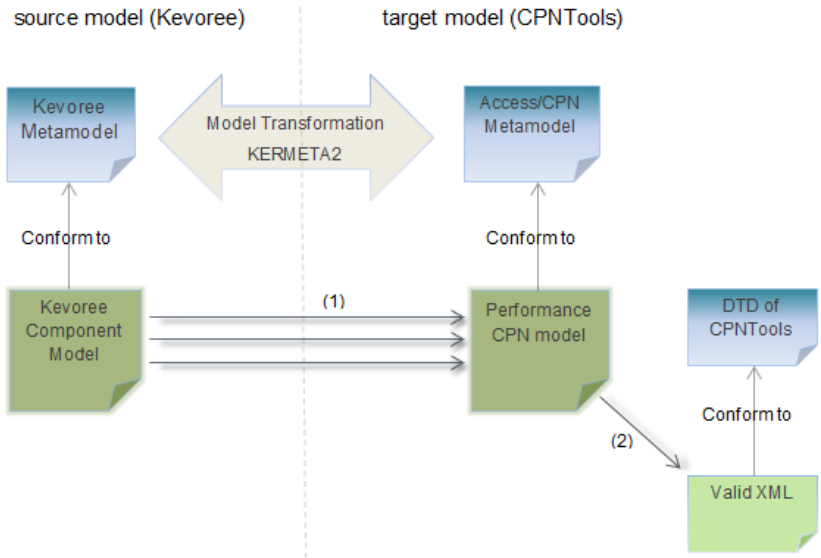

Figure 2 – Transformation process from the Kevoree component model to the system-level CPN model

formances efficace en termes de temps d'exécution. Notre cadriciel est complété par un processus de développement et de validation qui guide le développeur de composants et l'analyste de propriétés de qualité de service par un cycle itératif de conception. Ce processus de développement prend en compte des propriétés de performance.

Le cycle itératif comprend les phases suivantes. Une première phase construit un certain nombre de modèles de composants en utilisant des bibliothèques disponibles de composants Kevoree. Pour chaque alternative, une deuxième phase compose les modèles de comportement de ces modèles de composants individuels pour produire un modèle de performance du système global. Une troisième phase réalise l'analyse de ces modèles de performance par rapport aux exigences. Une quatrième et dernière phase permet au moteur de raisonnement de choisir la meilleure alternative à partir des résultats d'analyse et de réaliser la reconfiguration à partir de ce choix.

L'approche proposée dans cette thèse offre les avantages suivants.

- Conformément aux principes de l'ingénierie des modèles, au moment de la conception un architecte peut travailler sur les modèles de comportement correspondantes des composants, définir le modèle de performance attendue du système et procéder à l'analyse de la performance, avec un niveau d'abstraction égal à celui des propriétés fonctionnelles du système en cours de conception.
- L'analyse des propriétés de performance d'un modèle s'opère dans un délai compatible avec une adaptation continue du système ayant lieu plusieurs fois par minute.

- La bibliothèque de patron de conception contenant des descriptions types de modèles de performance et de comportement accélère la phase de conception et évite le recours à un expert en réseaux de Petri stochastiques.

  .

- L'emploi de modèles paramétrés à aspect pour la description des patrons de conception est compatible avec les pratiques de la conception par aspects.
- L'emploi de réseaux de Pétri colorés offre une grande puissance d'expression pour la construction des modèles de performance des systèmes complexes.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Component-based software design is now a well established approach for building reusable and trust-able 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 [19]. 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 thesis we focus on this category of systems that must support major architectural changes at run-time, without stopping but instead by hot-deploying. 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 user's needs, for instance addition or removal of functionality, 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 [93]. 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 have specific architectural features, and their design techniques are an active research topic [27]. However, flexibility should not be implemented at the expense of loss of trust in the correctness of these adaptive systems. As these system's 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.

To address the challenging issues discussed above, we propose to apply a formal language to the design and analysis of real-time adaptive systems. We also propose to use Model-Driven Engineering (MDE)[55] for an automatic evaluation process. MDE enables the formal specification of performance-relevant models of components; these models serve as an input into an automatic composition engine that assembles sub-models into a performance-relevant system-level model. Analyzing this system-level model will provide the reasoning engine with predictions on system quality attributes, including performance.

The problem of performance prediction of component-based assembly is a challenging task. Predictable Assembly (PA) [101] allows assembling at early design time a system out of individual independent-developed arbitrary components with predictable functional and extra-functional properties. Once the assembly is realized, and the attributes of individual components are available, the quality of the attributes of the assembly could be reasoned. In general, the process of performance prediction of component-based systems at design time requires the following tasks:

1. Modeling the functional behavior tailored with performance properties of individual components
2. Identifying the assembly structure and mapping scheme on hardware platforms
3. Analyzing and reasoning about the performance properties of the assembly.

A number of Component-based Software Engineering (CBSE) performance prediction methods have emerged during the last decades, such as PALLADIO [12], KLAPER [44]. A detailed analysis and comparison of these approaches is presented in section 2.3. However, most of them do not tackle the problem of performance evaluation at run-time, which is needed to support the continuous reconfiguration of modern adaptive distributed systems. Based on that analysis, we identify the following aspects that are important for supporting runtime performance evaluation:

- Modeling semantics for specifications of various performance properties of individual components. A third-party software component could be deployed in different environ-

ments. The need to consider its potential execution environments and incorporate them into the system performance modeling makes the problem more difficult. Additionally, modeling of complex systems needs a powerful formalism , for instance, the ability of concurrency, synchronization, and mutual exclusion of shared resource modeling, etc.

- Well-defined semantics and rules for assembling the performance-related models of individual components in an automatic way. The performance specification of software components and assemblies is a basic problem that must be solved to enable system assembly out of individual components. The description of performance aspects of the system-level mode should be derived from that of individual components, so that the system-level model could be available for analysis and the performance aspects of the whole system will be measured.

- Reasoning framework allowing to extract and analyze the performance analysis results. The performance results should be reasoned in a perfect time to adapt to the continuous changes of highly adaptive systems.

- Time-effect performance analysis for performance evaluation of alternative configurations. Using models-at-runtime more than one configurations must be evaluated, leading to more computation time for evaluation.

1.2 Research questions

Based on the above discussion, we consider a number of research questions that need to be addressed in this thesis:

Research Question 1 How should the functional and performance properties of individual independent-developed components be specified in order to enable automated composition of these properties and to capture all environment aspects which may influence the performance of the components?

Research Question 2 How to evaluate performance properties of combined system architectures at run time in an automatic way?

Research Question 3 How can the reasoning engine compare architectural alternatives and select an optimized one with respect to multiple quality attributes?

Research Question 4 Can this approach proposed in this thesis be applied to others systems, such as embedded real-time systems ones?

Research Question 5 How can other extra-functional properties like security, availability, etc be expressed and evaluated?

1.3 Thesis outline

Our approach is three-fold:

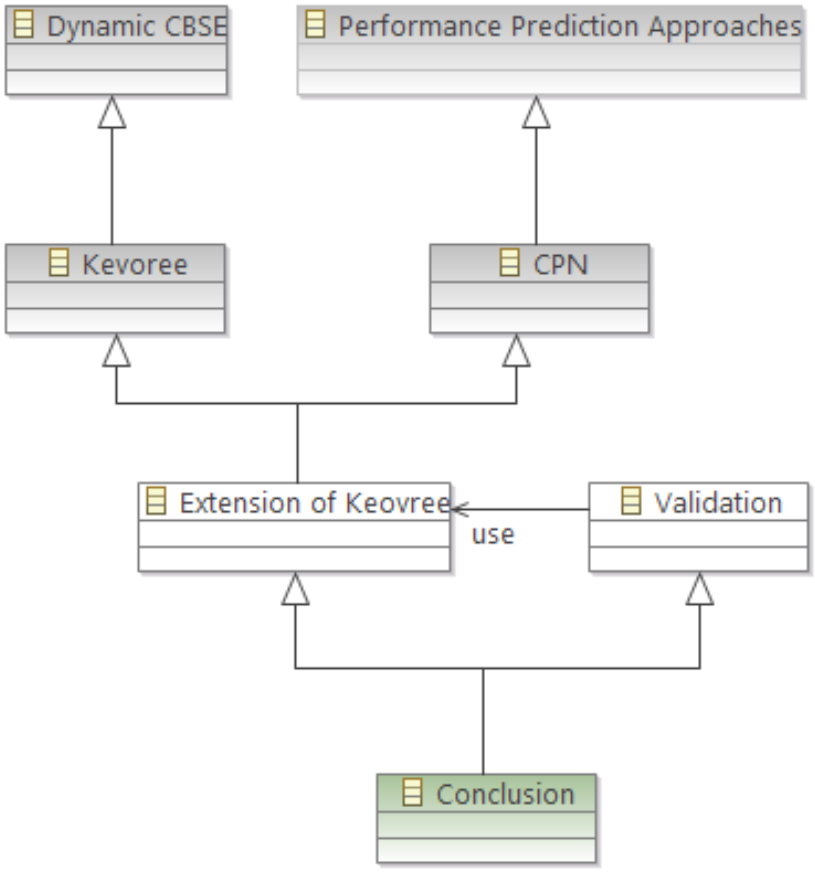

Figure 1.1 – Overview of the thesis structure

- We integrate stochastic colored Petri nets as a metamodel extension for specifying time-related component properties (Section 3.2).
- We set up a library of frequently used patterns to help designers to superimpose timed behavior descriptions on functional models (Section 3.3). These patterns also map timing evaluation results back into higher level system models.
- We provide an integration of timing evaluation tools in the models at runtime paradigm (Chapter 3), with an evaluation time compatible with rapid changes in the architecture.

Figure 1.1 gives an overview of the thesis structure. Figure 1.1 gives the contents of my thesis and the relations between different parts in my thesis. The light grey elements present the background and State-of-the-Art parts of the thesis. The white elements represent the description of methodology, framework, and ours contributions.

The remainder of the thesis is organized as follows:

- Chapter 2 presents the background in dynamic component-based software development and performance prediction approaches. Chapter 2 consists of three subsections. The

first subsection 2.1 (the Dynamic CBSE element in the top left-hand side of Figure 1.1) introduces fundamentals of component-based software engineering and provides background information on Model-driven software development and Models@Runtime. The second subsection 2.2 introduces different methods of performance analysis. Finally, the third subsection 2.3 (the Performance Prediction Approaches element in the top right-hand side) gives a survey on approaches for performance predictions of component-based systems.

- Chapter 3 first formulates the performance-oriented development process in the Kevoree framework. We then discuss in detail the extension of the Kevoree framework for performance prediction.
- Chapter 4 describes the validation part of the thesis. In this chapter, the approach is validated through a case study that is implemented based on our proposed framework.
- Chapter 5 concludes the thesis and presents perspectives. In this chapter, we discuss the advantages and drawbacks of the approach and then we list some open issues and we propose future work to response the research question #3.

Chapter 2

State of the art

This chapter presents the background and concepts that are needed to understand the rest of the thesis, and then discusses some related approaches. This chapter consists of three subsections.

The first section 2.1 introduces basics in the areas of component-based software engineering and background information on Model-driven software development and Models@Runtime. This section also defines the different types of extra-functional properties of component-based technologies. At the end of this section, an overview of the Kevoree framework is presented.

In the second section 2.2), we introduce different methods of performance analysis. This section also clarifies the difference between quantitative and qualitative analysis. At the end of the section, we introduce the concepts pertaining to coloured Petri nets, the CPNtools and Access/CPN software. We define a metamodel extension based on stochastic Petri nets as an internal time model for prediction.

The third section 2.3 gives a survey on approaches for performance predictions of component-based systems. Through this section, the drawbacks and advantages of each approach are also analyzed.

2.1 Dynamic component-based software engineering

2.1.1 Component-based approach

There are several proven advantages of the component-based approach. A major advantage of the component-based approach is that it allows for the rapid construction of qualified applications with reusable components, the facility of maintainability and the dynamic of applications [112] [43] [30] [51]. A deeper explanation of concepts such as component, component assembly, and component framework will be presented in the next sections. Benefits of the component-based approach can be summarized as follows:

Extendability Legacy software systems are difficult to extend. Due to the development of legacy software systems as monolithic systems, new functionalities must be inserted directly into the source code, and the whole program must be then rebuilt, whereas component-based systems can be updated and extended easily by adding new components or updating existing ones.

Reduced time-to-market The component developers can focus only on the construction of the functionality of the component, without taking into account system resource allocation, because a component framework provides other runtime services commonly needed by components.

Predictability All design rules and patterns of the whole system are defined by the component model, which is then enforced for each component. For that reason, it is unlikely to make a mistake in the design of a whole system.

2.1.2 Reusable components

The definitions of components vary and there has not been a unique general definition of what a component is so far. First, we refer to one of the precise definitions of components [102]: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Another definition of the components in [8] states that the component is an opaque implementation of functionality, subject to third-party composition and conforming to a component model. There exist different types of the components depending on the design purposes. We distinguish three types of component: Black-box, White-box and Grey-box.

Black-box is generally a part of a program which provides a functionality, in which users know only inputs and outputs. The users call the functions with inputs and expect outputs. The inner implementation of the functionality remains hidden. In contrast, White-box allows the users to have access to the inner architecture or source code, it makes called white-box. White-box makes generally the replacement of an old program by new one [102] more problematic. From the white-box point of view, users of components may study the source code of the program, so that users may adjust programs. For example, users change a sequence of calls, modify somehow input and output values to obtain a better performance of the program. Components designed as black-boxes are more convenient for future replacement. Since the components' primary goal is to be replaceable, it even more highlights the need for black-box components. One point worth noting here is that there exist several component-based approaches where components designed as grey-box that aim at improving the predictability of extra-functional properties [15]. For instance, the Palladio component designs [12] include Service Effect Specification (SEFF) to model performance-relevant information together with behavior model of the component. Users have access to the SEFF models of the components but not to the source code of the component, so that the component assembler may select convenient components for their design goals, for example, to obtain a component assembly that satisfies some performance requirements of the final system. The selection of components is based on interfaces of components and information from SEFF models.

2.1.3 Component model

The component-based approach consists in designing and developing systems by assembling reusable components. It is composed of an assembly of prefabricated, preconceived, pretested components and interconnected by contracts, and required and provided interfaces. To construct a component-based system, the developers rely on reusable components instead of creating new ones from scratch. The component model gives a uniformity to components and their composition. Its use is to define how a component should look like, how components communicate between each other, and which resources they use. The component model ensures that the components are compatible in terms of deployment, communication. It determines the rules that components must follow to be able to cooperate and its minimal misunderstood assumptions. Another goal of usage of the component model is the predictability. The component model may define typical software requirements, for example, to avoid deadlock, manages race-conditions, synchronization. The component model may also support extra-functional properties such as performance-relevant properties, memory consumption. Paper [8] claims that the component model should provide:

A component type expressed by interfaces that the component implements. When the component implements more interfaces its type is made of the types of all implemented interfaces. In other words the component is polymorphic with respect to all implemented interfaces. A component type is basically a factory for building component instances. The identification and design of the component type are performed by the business experts and the architects of the system following classical software engineering techniques.

An interaction mechanism used by the component model to specify how components are located, which communication protocols are used; it may also define which level of quality of services is achieved.

A resource binding description describing how each deployed component is bound to some resources. A resource is provided by a framework, in which the component is deployed, or by other components. The component model describes which components are available and how the components bind to them. Consequently, the component model drives the life cycle of components and manages resource assignment.

2.1.4 Extra-functional properties modeling of component-based technologies

The main objective of this section is to present the notion of extra-functional properties of software systems. Even when a particular functional requirement of a component is satisfied, the functionality of the component in the target environment may not be guaranteed unless all extra-functional requirements are fulfilled. From the users' point of view, there are two disjunctive parts to be considered:

1. functions, for which the component has been developed for;
2. a set of extra-functional properties of the component.

The correctly working function developed in the component may not be enough in the target environment. It is useful to allow users to take into account extra-functional requirements of the component to fulfill users' needs. For that reason, it is important to express relevant extra-functional properties of the components. While the specification of functional properties is well understood in current industrial models and frameworks, most of them lack the support of extra-functional properties. A functionality of published components does not depend on whether they are able to guarantee their function with regard to specific extra-functional requirements. Let us point out that the term *extra-functional properties* lacks a standardized definition.

The work in [19] does not define precisely extra-functional properties, but the authors define a general model software contract. The extra-functional properties are derived from extra-functional aspects, by specifying constraints over the values of extra-functional aspects. They divide contracts into four levels of increasingly negotiable properties:

Syntactic level This first level of contract is required simply to make the system work. In this level, the designers specify functions that their components can perform.

Behavior level . This second level improves the level of confidence in sequential context by defining the requirements and outcome of operations. For instance, the Eiffel language [77] allows for the definition of pre-conditions and post-conditions to capture conditions that must hold to ensure a result of the computation. The other example is the *assert* command in the Java language. This construct is used to check the validity of input parameters of methods. Behavioral contracts can also be expressed by means of the contract expressed using UML and OCL (Object Contraint Language) [107] formalism.

Synchronization level The third level improves confidence in distributed or concurrency contexts. This level of contract concerns all aspects connected with multi-threaded computation.

Quality of service level The fourth level quantifies quality of service. Quality of service includes attributes such as maximum response time, delay, average response time, memory usage. They are mainly relevant in resolving whether the whole component system will meet extra-functional requirements when executed on a given platform.

This classification does not cover all kinds of quality requirements. Extra-functional requirement includes all expected quality characteristics from a user's point of view. extra-functional requirement could be divided into different sets such as quality of service, behavior, synchronization, etc. Generally, we can describe the association of these sets as below:

Behavior + synchronization + QoS requirements $\subset$ *extra-functional requirement.*

It is useful to precisely and formally describe all relevant extra-functional properties of components since it allows checking whether a component matches extra-functional requirements. An active research activity has been devoted to formal definitions of extra-functional properties such as HQML (Hierarchical QoS Markup Language) [47], TADL (Architecture Description Language for Trustworthy Component-Based systems) [79], QML [41], CQML [2]. When component compatibility checks are improved by extra-functional properties, it leads to more precise decisions on whether one component is suitable as a replacement for the other

one. Chapter 4 will present approaches and their underlying analysis methods that support extra-functional properties expression and prediction.

In short, extra-functional requirements are the requirements that specify criteria that can be used to analyze the operation of a system, rather than a specific behavior. Typical extra-functional requirements are reliability, scalability, and cost. Extra-functional requirements are also known as constraints, quality attributes, and time constrained responses and availability of required service. Temporal properties are considered as hypothesis on the behavior of the system. Figure 2.1 is an example of a quality model presented in [38] to define the quality of software systems. The quality model is explained with the help of six main factors and several subfactors:

Functionality is defined as a set of attributes that deal with the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

Reliability is defined as a set of attributes that deal with the capability of software to maintain its level of performance under stated conditions for stated period of time.

Efficiency is a set of attributes that deal with the relationship between the level of performance of software and the amount of resources used, under stated conditions.

Usability is a set of attributes that deal with the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.

Maintainability is a set of attributes that deal with the effort needed to make specified modifications.

Portability is a set of attributes that deal with the ability of software to be transferred to from one environment to another.

There exist several other quality models such as [22] or [73].

2.1.5 Model-driven engineering

MDE is a software development methodology that focuses on abstracting some aspects of reality for a given purpose by creating and exploiting domain models for this purpose. Figure 2.2 shows the Meta Object Facility (MOF) pyramid-shaped structure [103]. The metamodel describes the concepts and relationships of a given domain. Each model in Figure 2.2 conforms to the well-defined metamodel. The first level model in the pyramid-shaped structure is the meta-metamodel that conforms to MOF, which is self-described. The metamodel is the most important part of a Domain-Specific Modeling Language (DSML). It defines concepts in the domain. For the purpose of designing metamodels, there exists EMF (Eclipse Modeling Framework) [5], which is a *de-facto* standard modeling framework.

From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. We can also use EMFText [10] to create the textual syntax of a DSML or the GMF (Graphical Modeling Framework)

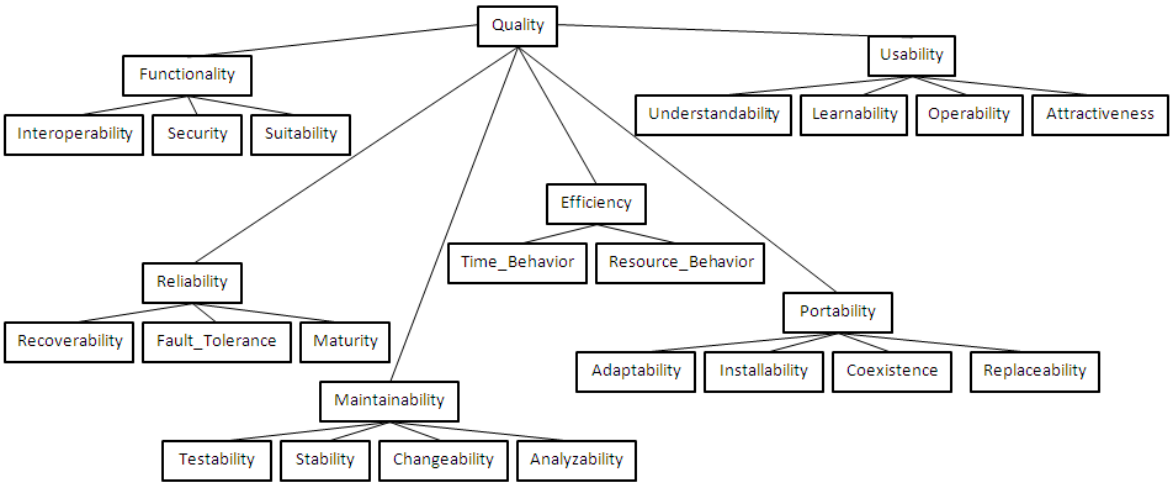

Figure 2.1 – Non-functional properties of software systems

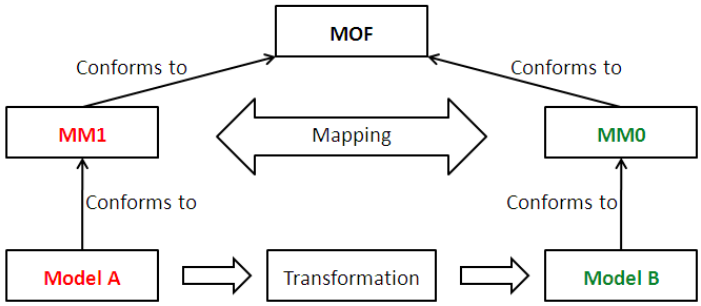

Figure 2.2 – A model transformation

[11] to construct a graphical syntax. Tools and frameworks such as Kermeta [55] provide static and dynamic semantic of metamodel. Kermeta provides OCL-like syntax to specify static semantics, such as the constraints the models must obey. Kermeta allows designing invariants and pre/post-conditions. The conventional modeling framework like EMF does not allow the definition of behavior in operations integrated into a metamodel. Kermeta allows aspect-oriented modeling, which helps design pre/post-conditions, variants and behaviors of a metamodel.

MDE provides a variety of software processes to allow the creation of modeling domains and the transformation of its models. The Model-driven Architecture (MDA™) marketed by the Object Management Group (OMG) presents a model-driven approach to system development. The MDA defines two principle model levels, Platform Independent Model (PIM) and Platform Specific Model (PSM). The former specifies an application independent of the platform technologies, but only the specification of business layer of the application. The latter specifies an application in a specific modeling domain. The platform independent models are incrementally transformed and refined into lower-level platform specific models. The PSMs are then transformed into implementation artifacts such as implementation code. This ap-

proach of automatic construction of systems from high-level models is based on the principle of *model transformation*.

The *Model transformation* is another important aspect of MDE. A *model transformation* takes input models conforming to an input metamodel MM_1 and produces output models conforming the output metamodel MM_0 as illustrated in Figure 2.2. A transformation is basically an ensemble of rules to implement a mapping between elements in source model and elements in target model. The model transformation may transform models within the same modeling domain (endogenous transformations) or between different modeling domains (exogenous transformations). There exists a number of model transformation languages such as Kermeta [85], rule-based ATL [56] [57], graph grammar based AToM [105]. Different types of model transformation can be created using model transformation languages. Such types of transformation are Model-to-Model transformation and Model-to-Text transformation. Model-to-Model transformation is a transformation mechanism that transforms a source model into a target model. The source and target models conform to their respective metamodel. Model-to-text transformation transforms a model into text, which is normally the source code. The next section presents the models@runtime approach, which is based on the model-driven engineering.

2.1.6 Models at runtime

Adaptive systems are often expected to adapt to the changes in their execution environment. Hence, self-adaptive systems require to adapt their behavior and their structure at runtime with little or no human intervention. Models@runtime [81] [21] has emerged as a promising approach to develop adaptive systems. An advantage of using models@runtime is to develop adaptation mechanisms that leverage software models to manage complexity in runtime environments. The basic idea of models@runtime is to apply the model-driven engineering (MDE) approaches to the runtime environment. At a first glance, models@runtime may be considered as a reflection concept. The reflection deals with causally connected, self-representations of an underlying system. The models should mirror the system, its current state and behavior. If the system changes, the models should change and vice versa. However, in the reflection research domain, models are often related to the computation model and hence tend to be based on the implementation space and rather low level. In models@runtime, models are on a much higher level of abstraction, and they are causally connected models related to the problem space. Another key point of models@runtime approach is that models should be intrinsically tied to the models produced from the MDE process. The work in [21] defines the models@runtime concept: a models@runtime is a causally connected self-representation of the associated system that emphasizes the structure, behavior or goals of the system from a problem space perspective. Runtime models provide abstractions of runtime phenomena. There are several benefits of runtime models such as dynamic state monitoring, control of systems during execution, dynamic observation of the runtime behavior to understand a behavioral phenomenon. Now that we have introduced the model-driven engineering and the models@runtime approaches, the next section presents the Kevoree framework that is dedicated to a technique based on models at runtime that enables dynamic adaptation for distributed component-based systems. The metamodel extension proposed in Chapter

4 based on stochastic Petri nets is integrated into Kevoree metamodel as an internal time model for prediction.

2.1.7 Kevoree framework

The Kevoree framework [1] is dedicated to a technique based on models@runtime that enables dynamic adaptation for distributed component-based systems. This framework supports homogeneous continuous design of dynamic distributed component-based software architecture and an intelligent reflection layer that allows evaluating new configurations without stopping the current running system. The Kevoree project was inspired by the EUT-ICT DiVA project (Dynamic Variability in complex, Adaptive systems; www.ict-diva.eu), which is a first attempt to enhance reflection with models@runtime. The rest of this section discusses the kernel of the framework-dynamic adaptation mechanism, and the Kevoree component model that provides concepts to describe the underlying infrastructure of distributed component-based systems.

2.1.7.1 Dynamic adaptation modeling

The approach uses software models at runtime as well as at design time to cope with two main difficulties related to adaptation management and evolution management. The first difficulty is coping with the variability that can lead to explosion of several adaptive artifacts. The set of possible configurations of an adaptive system is specified by identifying variation points, which represents points in the software where variability may occur. The number of configurations explodes in a combinatorial way w.r.t the number of variants, and the number of possible configuration transitions is quadratic w.r.t the number of configurations. The second difficulty is the evolution of the adaptive system. Evolving an adaptive system means dynamically changing the adaptation state machine (adding and removing states and transitions). The modifications to the adaptation state machine leads to the redeployment and restart of the new configuration. The DiVA project uses an adaptation metamodel to assist in modeling the DSPL (dynamic software product line) at design time [83] [36]. Models conforming to adaptation metamodel are the main data manipulated by the runtime infrastructure responsible for dynamically adapting component-based applications at runtime. These models provide a high-level basis for reasoning about relevant aspects of the system and its environment and offer enough details to fully automate the dynamic adaptation process. It is possible to make the design specifications evolve at any time, before initial deployment or while the system is already running. Figure 2.3 presents the concepts of the approach, which includes the architecture of the system in the right hand side and the adaptation layer managing the dynamic variability in the left hand side.

For the architecture model, designers can use existing modeling languages, such as the Unified Modeling Language (UML) or Service Component Definition Language from the Service Component Architecture (SCA), or any architecture description language, to describe the architecture. The approach uses Aspect-oriented modeling (AOM) technique to represent the variability in the architecture model; leaf features of the DSPL model are modeled as aspects, which can be woven or not in the application. The adaptation model captures the

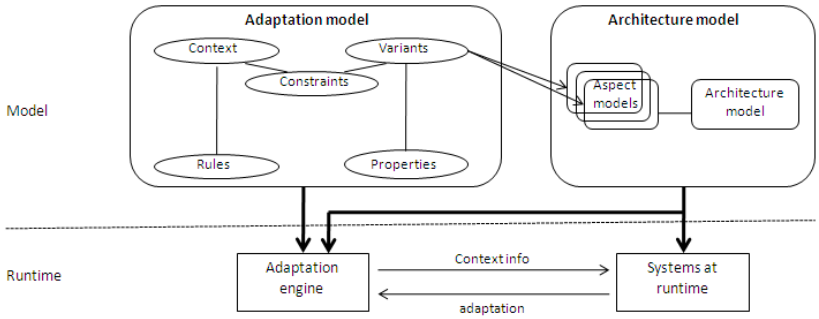

Figure 2.3 – Art dynamic adaptation model

dynamic variability information, *i.e.* which functionality should be used depending on the context. The adaptation model includes four different aspects: variants, adaptation rules, dependencies, and context. The variant aspects and constraints (requires, excludes) among variants are modeled using feature models. The context aspects specify the system's environment. A set of variables specifies those aspects of the environment relevant to adaptation. At runtime, the variable values are provided by context sensors, and these may trigger a system reconfiguration. The adaptation rules part of the reasoning model describes selection of the variability features according to context. The approach investigates two different formalisms to capture the adaptation rules, which are event-condition-action (ECA) [33] [59] rules and goal-based optimization rules [48].

2.1.7.2 Models at runtime

Kevoree uses *models@runtime* to enable intelligent reflection for distributed component-based systems. As discussed in previous sections, models@runtime relies on model-driven approaches to cope with the complexity of dynamic adaptation. The conventional reflection approaches often offer reflection APIs to support the introspection of the system and dynamic adaptation (by applying CRUD operations on elements of the system). In a nutshell, models@runtime is a reflection model that is uncoupled with runtime system and modeled at a higher-level abstraction. This model allows validation of new configurations, checks when changes appear, without modifying the running system. The new configurations are compared with the current configuration, which is a mirror that reflects the running system. The adaptation model represents a set of commands to transform the current configuration into the new configuration. The adaptation engine implements these commands at the platform level. If a concrete action execution fails than the adaptation engine rollbacks the configuration to its previous state. Figure 2.4 shows the adaptation mechanism of Kevoree.

2.1.7.3 Kevoree metamodel

The Kevoree metamodel [32] supports many features to allow models@runtime on top of a distributed component platforms:

1. Coupled and uncoupled modes: the coupled mode is like in conventional reflection

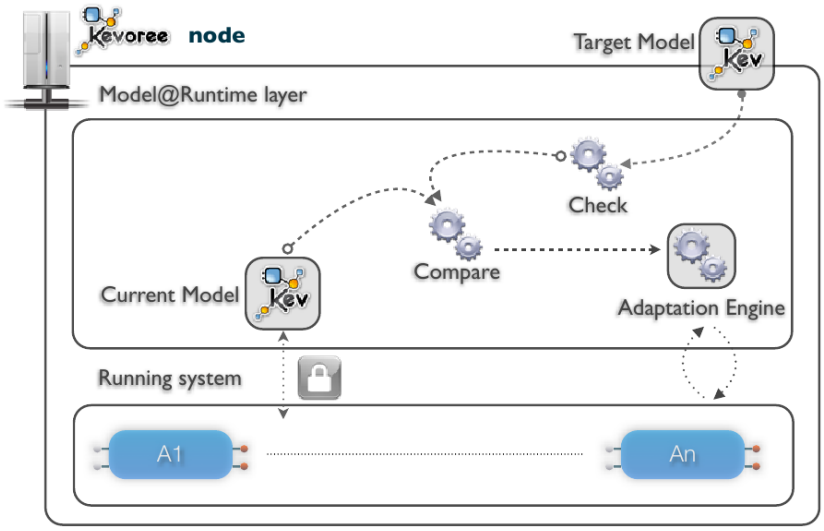

Figure 2.4 – Model at runtime

approaches that synchronize operations on reflection model and runtime. Contrarily, uncoupled mode allows working on reflection models without modifying the running systems.

2. Type and instance separation.

3. Closed isolation of components: closed isolation of components means that for each couple of components binding together, they can not execute the code of each other. This property is to ensure the capacity of halting a component without influences on other processes.

4. Dynamic provisioning: capacity to dynamically integrate executable code to a running system.

5. Distributed topology model.

6. Channel type: to describe complex semantics of component bindings.

This subsection presents a subset of the elements of the Kevoree metamodel.

Node, NodeType: Figure 2.5 shows a part of Kevoree metamodel related to distributed topology model. The Node element describes a logical node in distributed systems. A *Node* can be a container of component instances, channels and other nodes. This allows a hierarchical modeling in Kevoree, whereas the parent node is responsible for starting and stopping its child nodes.

Type definitions, Instances: Figure 2.6 shows elements related to type definitions. A *TypeDefinition* element contains third party dependencies (DeployUnit). *DictionaryType* defines the parameters of the component type. Each instance refers to its type definition. Component instances are bounded together through bindings and channels. *Component* and *Channel* elements capture the complex binding semantics.

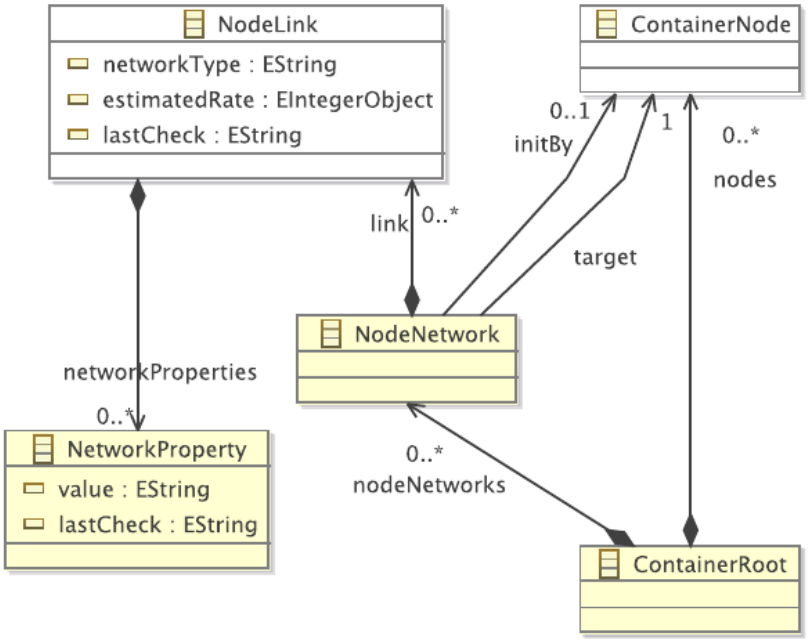

Figure 2.5 – Kevoree topology model

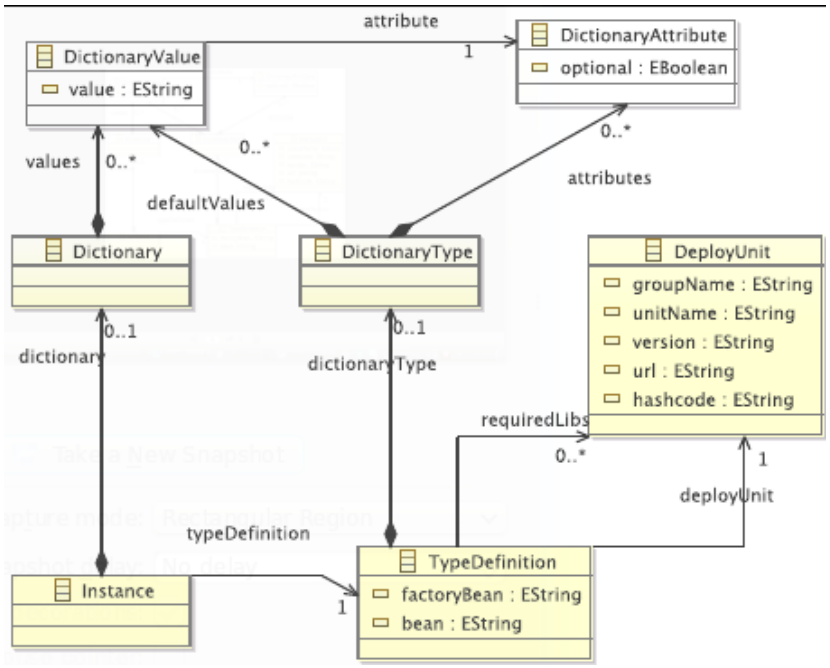

Figure 2.6 – Kevoree type definition

2.2 Methods for performance evaluation of software systems

2.2.1 Analysis and validation of software models

This section discusses validation or analysis of the system model. There are two aspects to consider when analyzing a system model: the qualitative aspects and quantitative aspects. The analysis of the whole system aims at providing qualitative and quantitative information of the system behavior [3] [46] [74] [50] [62] [61]:

- qualitative analysis aims at ensuring that the properties of the system correspond to the expected properties, for instance the absence of bottleneck, the reachability of a specific state;

- quantitative analysis and performance evaluation aims at quantifying the performance of the system w.r.t the objectives of the system. The performance aspects may be the response time of the users requests, the average delay of the message transmission in a communication network, the bandwidth of transactions in data based systems, etc.

In recent years, verification techniques and performance evaluation are used increasingly in industry to analyze a variety of systems such as real-time systems, embedded systems, adaptive systems. These techniques have proved their efficiency. For the purpose of verification of extra-functional properties, including performance-related of a system, the analysis methodology can be initially subdivided into two approaches: measurement-based and model-based approaches. The former comprises three distinct fields: measurements, benchmarks, prototypes. This technique requires the availability of either the system to be suited or its approximation, so that it can be observed. During the design process, before the implementation phase, measurements on real systems are obviously not applicable, and prototype implementations are very difficult because of the necessity of specifying many details that are far from being decided. The state-of-the-art review in this thesis concentrates on the latter one, which can be partitioned into two types: simulation models and analytical models. This technique can be applied during the early system design phase. In the case of analytical models, the description is given in mathematical terms, whereas in the case of simulation models the description is given by means of a computer program. The simulation will almost always deliver results that are less accurate than the ones that can be obtained by using analytical models, and model complexity of very large systems may cause considerable negative impact on simulation time [98]. The advantage of simulation over analytic modeling is that very detailed system behaviors can be captured. The analytic modeling requires much more constraints in modeling. In other words, simulation allows the development of more detailed models, whereas analytically models are normally more abstract. Additionally, models can be either deterministic or probabilistic. It may be simpler and more realistic to model the systems by means of probabilistic assumptions due to the fact that the details are often not known, and even when they are, their inclusion may lead to very complex models. Furthermore, the probabilistic approach may be advantageous because it may provide sufficient accuracy, while yielding more general results.

2.2.2 Qualitative analysis

Quantitative analysis and verification aims at analyzing the qualified properties of the systems [13] [114] We distinguish three classes of verification techniques: model checking, proving, and test.

Properties to verify: there are several types of property that can be verified [30]. The properties listed below are some examples:

- Reachability: to verify that in some conditions, whether a state of the system can be reached or not, for example, whether an initial state may be attained, a state of availability of resources, etc.
- Liveness: the properties that cannot be checked on finite executions (they need to be checked on infinite executions).
- Safety: the properties that can be checked on finite executions. For example, it cannot happen that both processes are in their critical sections simultaneously (mutual exclusion).
- Deadlock free: it ensures that the system never reach a blocked state where it cannot quit.

2.2.2.1 Model checking

The verification or model checking [71] relies on the construction of a finite model of the system, which is then used to check whether a specified property is correct in this model or not. This method is considerably automatic and rapid. In general, the way to perform this verification is to verify that a property expressed in terms of temporal logic is correct in the system. The model checker always proceeds by exploring the state space, whatever the formalism used to model the system. In most cases, the model checkers give the positive response if the property is guaranteed for all the behaviors of the system, and a negative response completed with a counterexample in case of violation of the property.

2.2.2.2 Theorem proving

Theorem proving consists in expressing the system and their properties in terms of a mathematical logic and then researching a proof of these properties. This approach is precise, but it requires user intervention. The theorem proving can be applied to all phases of development of the system [49] [91].

2.2.3 Quantitative analysis

Quantitative analysis aims at computing quantitative performance-relevant parameters. The methods of performance evaluation are based on mathematical fundamentals, more precisely theory of probability and Markovian stochastic processes. The performance evaluation consists of the following steps:

1. describe the static and dynamic aspects of the system under study by using formal models;
2. analyze the system by generating the state space. The state space is computed as a stochastic process, for example, a Markov chain;
3. resolve the obtained chain to compute the probability of each state in the chain;
4. compute the performance aspects based on probabilities obtained.

2.2.3.1 Performance aspects

The Markovian methods in [29, 25, 60] allow different kinds of analysis: transitory and stationary analysis. The calculated performance aspects give information of efficiency and productivity of the system. On the contrary, transitory and stationary analysis answer to questions such as: in a long period, what is the usage of the server (the percentage of time the server is busy). For these two kinds of analysis, there are frequently a set of the calculated performance aspects: the throughput (the number of request per time unit), the average of response time (the average time to execute a user request of the system), the average of waiting time (the average of time the request has to wait in a queue), resource usage (the percentage of time in average the resource is busy), the frequency of the executions of a task, etc.

2.2.3.2 Performance models

Markovian process:

Markovian processes [29, 25] represent a class of stochastic processes. A stochastic process is a family of random variables $\{x(t), t \in T\}$ defined on a given probability space, indexed by the time variable t, where t varies over an index set T.

There are two types of stochastic processes: discrete value and continuous value, and discrete time and continuous time. If each random variable $X(t)$ for different values of t is a discrete random variable, then the stochastic process is a discrete value. If the process is defined only for discrete time instants, then it is a discrete time stochastic process. For a discrete time process, a random sequence X_n is an ordered sequence of random variables X_o, X_1,... Essentially a random sequence is a discrete-time stochastic process.

A Markov chain or Markov process is a process in which the probability of the system being in a particular state at a given observation period depends only on its state at the immediately preceding observation period. A random process whose future probabilities are determined by its most recent values. A stochastic process $x(t)$ is called a Markov process if for every n and $t_1 < t_2 ... < t_n$, we have $P(x(t_n) <= x_n | x(t_{(n-1)}), ..., x(t_1)) = P(x(t_n) <= x_n | x(t_{(n-1)}))$.

This is equivalent to $P(x(t_n) <= x_n | x(t) forallt <= t_{(n-1)}) = P(x(t_n) <= x_n | x(t_{(n-1)}))$

Queueing network

Queuing networks [25] have been used widely for performance evaluation discrete-event systems such as information systems, communication network, etc. A queuing system simply consists of a waiting queue (or a buffer), with one or many servers. Queuing theory answers question like: mean waiting time in the queue, mean system response time (waiting time in

the queue and service time), mean utilization of the servers, distribution of the number of customers in queue, distribution of the number of customer in system, probability that the queue is full or empty. A basic queuing model consists of (a) customer population, (b) n^{th} customer arrives at time τ_n (c) inter-arrival time between two customers: $t_n := \tau_n$ - τ_{n-1}, where t_n is an independent and identically distributed random variable (d) service time x_n is independent and identically distributed random variable. A queuing model is characterized by:

- Input process: the sequence of request for service, often specified in terms of inter-arrival time;
- Number of servers;
- Queue discipline: the disposition of blocked customers. We distinguish different kinds of blocked customer: (a) blocked customers cleared who leave the system immediately (b) blocked customers delayed who wait for service in a queue like FIFO, LIFO, etc. (c) blocked customers held who wait for service in queue (the sojourn time of a customer in queue);
- Distribution of service time;
- Service discipline (random, priority, round robin, ...).

From the informal point of view, a waiting queue is defined in three steps. The first step aims at defining the servers, the number of servers, the customer and the topology applied. The second step consists in defining the parameters of the model, such as inter-arrival time, distribution of service time, the number of clients. Finally, the third step of evaluation allows to obtain quantitative analysis results of the system by measuring the performance aspects. This latter step is similar to the Markov chain by analyzing the system behavior in a short term (transient solutions) or long term (steady state solutions). Steady state is a state where the system has run for a long time and tends to reach a stable state, e.g. distribution of customers in the system does not change.

As a drawback, the capacity of modeling in queuing network for the representation of concurrent and synchronized systems is much limited. In order to extend the capacity of modeling of queuing networks, the scientific community has developed several generalized models to cope with the complexity of concurrent and synchronized systems: stochastic process algebra, stochastic Petri nets, stochastic automata, etc. Recently, several extension models of conventional queuing networks have been proposed to remedy their inconveniences, such as:

- Extended Queuing Network [67, 7, 6] that allow description of the interesting characteristics of real systems, such as constraints of synchronization, concurrence and the finite capacity of waiting queue.
- Layered Queuing Network (LQN) [25] that allows modeling of the client-server communication pattern of distributed systems. LQN allow capturing the impact of multiple

layers of servers. A LQN is expressed by a set of tasks. Each task provides services represented by the input ports. The input ports of a task invoke the input ports of others at lower layers. The tasks are executed on processors. The modeling of sequential and parallel executions are expressed by activities. The activities can send the requests to output ports or accept the requests from other input ports.

We can list some examples of tools for performance evaluation for QN, LQN, EQN: RESQ/IBM [97], QNAP2 [106], LQN tool [25].

Stochastic Petri nets

Stochastic Petri nets (SPN) [9, 80, 37] are extensions of conventional Petri Nets [87]. The ordinary Petri nets allow only qualitative analysis. Stochastic Petri nets have been used widely for modeling and analysis of information systems.

A SPN is a six-tuple $GSPN = (P,\ T,\ I^-, I^+,\ M_o,\ W)$

where

- $PN = (P, T, I^-, I^+, M_o)$
  is the marked untimed PN underlying the SPN
- I^- *is a set of input arcs*
- I^+ *is a set of output arcs*
- $T_1 \in T$ *is a set of timed transitions,* $T_1 \neq \emptyset$
- $T_2 \in T$ *is mentioned as the set of immediate transitions,* $T_1 \cap T_2 = \emptyset,\ T = T_1 \cup T_2$
- $W = (w_1, \ldots, w_{|T|})$ *is a set of real values* $w_1 \in R^+$ where w_i is:
  - is a rate of a negative exponential distribution specifying the firing delay, when transition t_i is a timed transition ($t_i \in T_1$) or
  - is a firing weight, when transition t_i is an immediate transition ($t_i \in T_2$).

A firing delay is associated with each transition. It specifies the time elapsed before the transition can fire. This firing delay is a random variable with a negative exponential as its probability distribution function (pdf). The analysis of an SPN model is usually aimed at the computation of more aggregate performance indices than the probabilities of individual markings. The work in [80] quotes some of the most commonly performance indices that are obtained from the steady-state distribution over reachable markings. Such performance indices are:

1. the probability of an event defined through place markings can be computed by adding the probabilities of all markings in which the condition corresponding to the event definition holds true;
2. the pdf of the number of tokens in a place, can be calculated by computing the individual probabilities in the probability mass function (pmf) as probabilities of the event *place* p_i *contains* k *tokens*;
3. the average number of tokens in a place, can be computed from the pmf of tokens in that place;
4. the frequency of firing of a transition, i.e., the average number of times the transition fires in unit time;
5. the average delay of a token in traversing a subnet in steady-state conditions.

2.3 Performance prediction approaches for component-based systems

2.3.1 Performance prediction methods

In this section we present short summaries of the existing approaches of performance evaluation methods for component-based software systems as illustrated in Figure 2.7. We distinguish between UML-based approaches, proprietary meta-models based approaches, and measurement-based approaches, as illustrated in Figure 2.7. Each approach brings benefits and drawbacks in predicting performance of component-based software systems. The surveyed approaches here focus mainly on domain of embedded real-time systems or distributed systems. For embedded real-time systems, it is necessary to predict the violation of hard real-time deadlines. For example, PECT [52] [65] [101] and ROBOCOP [66] [24] are relevant approaches. For the domain of distributed systems, the violation of performance-relevant agreements normally is not as critical as violating the deadlines of real-time systems. Thus, the approaches that provide a probabilistic answer whether a given design can be expected to fulfill certain service level agreements, give more realistic performance prediction methods. Additionally, the behavioral models of software components in distributed systems can be considerably large, therefore it is not easy to describe them and suitable abstractions has to be made to construct an analyzable model. Palladio [15], and our approach are appropriate for distributed systems.

In general, the performance evaluation methods use model transformations to generate performance models from the component system models. To our knowledge, only our approach and KLAPER [45] use a standardized model transformation framework, such as QVT [89] and Kermeta [55].

Most of the summarized methods use queueing networks as the performance model with analytical analysis or simulation. The analytical methods for Queueing Networks require that users carefully satisfy many of constraints on modeling, because QNs are based on assumptions such as exponentially distributed execution times. However, the simulation

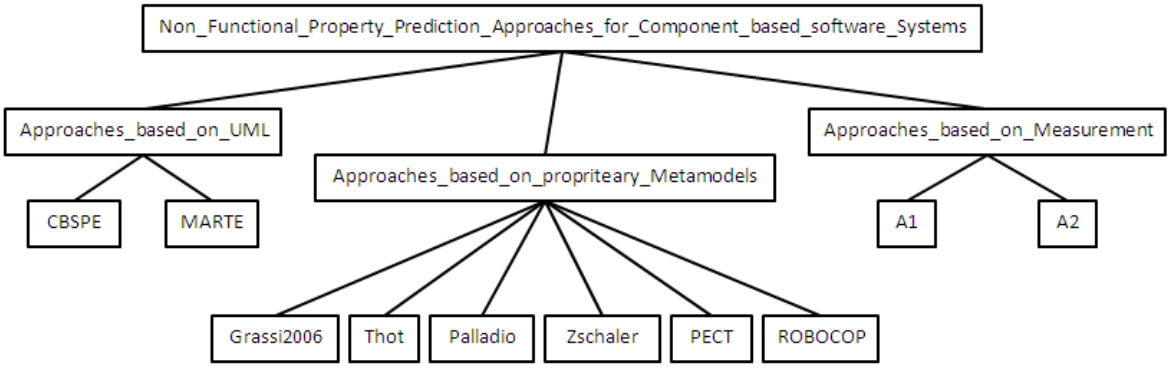

Figure 2.7 – Overview of prediction methods

tools normally require a high amount of execution time, which depends on the precision of prediction results and on the degree of statistical significance desired [53]. From the point of view of dynamic reconfiguration, to our knowledge none of the approaches support feedback of the prediction results into reasoning mechanisms as proposed in our framework. Among surveyed approaches only ROBOCOP offers automated support for prediction results feedback into the design model that help software architect to improve their design based on these feedbacks. ROBOCOP gives ability of evaluating a set of new design alternative configurations and perform a Pareto analysis that compares costs for each alternative with the expected performance [23]. The reasoning mechanism based on comparing performance-relevant feedbacks is out of scope of this thesis but it is a future evolution of our work. One important point is the use of distribution functions to specify resource demands. From the discussion below, several methods allow this ability of resource demands specification. This is considerably important for large software components, and even more for distributed systems, where it is difficult to describe the detailed resource demand as a constant value. The usage of distribution functions for specifying resource demands and extra-functional contracts is more realistic than deterministic contracts and detailed resource demands. For example, Palladio [15] [14] approach allows exponential distribution functions to specify resource demands.

2.3.1.1 Prediction approaches based on UML

The key point for a successful application of quantitative validation of performance properties during component-based software development is the existence of languages allowing performance specification when designing a component-based system both at component and at assembly level. Several studies deal with the use of UML to express extra-functional properties [16] [113, 31, 95]. The main benefit of UML-based approaches is their compliance to OMG standards [62]. These UML-based approaches allow developers to design models by using UML. The developers usually are not familiar with performance models that rely on complex mathematical solution techniques. The use of model transformations can help to hide the complexity of the performance models.

CB-SPE Bertolino and Mirandola [17] introduced a compositional methodology for component-based software performance engineering, the CB-SPE framework. Their approach supports the extra-functional properties modeling such as reliability and performance, which are important properties of realistic components. These properties is captured by the *quality of service* level of component contracts. CB-SPE is based on the concepts and steps of the SPE technology to propose a component-based architecture, performance validation and using OMG's SPT profile [110] as design model and queuing networks as analysis model. The proposed process is decomposed into components level and application level, one dedicated to the parametric performance evaluation of the components in isolation, and one for the predicting the performance of the assembled components on the actual platform. However, the parameterization of component model does not involve service input or output parameters nor explicit required service calls. The CB-SPE framework includes available modeling tools (ArgoUML) and performance solvers (RAQS) and also includes a transformation tool to map the UML model to queueing networks for analytical analysis.

MARTE UML supports modeling component behavior with sequence, activity, and collaboration diagrams. Component allocation can be described with deployment diagrams. However, UML only supports functional specifications, it would be useful to have extension mechanisms (profile, consisting of stereotypes, constraints, and tagged values) to allow modeling performance attributes and also other extra-functional characteristics. MARTE is a UML profile for Modeling and Analysis of Real-Time and Embedded system (MARTE) [70], supporting extra-functional properties annotation. The MARTE profile is not an extension to the UML meta model, but a set of domain profiles for UML. MARTE aims at supporting temporal verification of UML-based models. More precisely, it focuses on schedulability and performance analysis for real-time systems. The background for MARTE comes from two existing profiles, SPT and QoS & FT [88] profiles. The SPT has been proposed as a response to the requirements of introducing in UML diagrams quantifiable notions of time and resources usage, and QoS & FT profile has a broader scope that includes all kinds of QoS properties.

It points out the adoption of some useful structural concepts and qualifiers from UML profile for QoS & FT, as well as it imports annotations in the UML models, which describe the characteristics relative to the target domain viewpoint (performance, real-time, schedulability, and concurrency) for defining domain-specific extra-functional properties, and the analysis techniques can usefully exploit the provided features.

UML profile for CQML Paper [2] introduces a UML profile covering the expressive of CQML [2], that is a language usable for description of extra-functional properties. The authors define a set of stereotypes that correspond to CQML keywords (including QoSStatement, QoSCharacteristics, QoSProfile, QoSQuality as represented below). These stereotypes allow to model extra-functional properties the same way as they are written in the CQM's language. CQML defines basic constructs concerning extra-functional properties:

QoS Characteristics represents an extra-functional property. It contains a unique name and a data type, and additional information if necessary, such as a restrictive intervals for values, measuring unit, etc. It may define invariants for values of the property;

QoS Statement assigns constraints to QoS characteristics by using logical rules. Each statement is enhanced by a name and encapsulates a set of constraints for a set of QoS. A set of QoS with their constraints is then referred to by this name;

QoS profile it aggregates a set of QoS statements into one record with a unique name. A component binds to a profile to attach QoS to itself. Other components can reuse these QoS profile definitions.

2.3.1.2 Prediction approaches based on specific metamodels or formal definitions

The main benefit of these approaches is to allow researchers to create new abstractions of component-based systems, which might capture the performance-relevant properties and also other extra-functional properties more accurately than a UML model. The semantics of the models are often defined more formally than for UML.

ROBOCOP The ROBOCOP component system model [66] [24] allows different layered component models such as specifications, models, and executable code within the component distribution package. The approach adds a performance prediction framework on top of ROBOCOP component system model by combining static analysis and simulation on the executable system model provided by the development framework and the execution framework. ROBOCOP components have *provided* and *required* interfaces with multiple service signatures. Additionally, ROBOCOP components consist of a set of models including a resource model, a simulation model (behavioral specification), as well as an executable model. Extra-functional properties are contained in the resource model. ROBOCOP does not support composite components. ROBOCOP components only support specification of processor and limited resource demands for semaphores and memory on operations, with best, mean, worst cases distinguished. It does not allow probabilistic attributes (e.g. random variables) in the specification. The component specification allows only simple control flow (i.e. sequence and loop).

PECT Papers [52, 65, 101] propose a reasoning framework for performance evaluation of component-based software architectures. They aim at integrating component technology (i.e., component composition language) with analysis models. Thus, they allow the definition of an analytic interface, that consists of an analysis theory (i.e., rate monotonic analysis for predicting schedulability of real-time tasks) to support predictions about quantitative properties of the system. The approach allows synchronous and asynchronous communication with required services. It also supports the specification of resource demands with distribution functions by annotating components. For the purpose of performance analysis, the component architecture models are transformed into a so-called intermediate constructive model, and then from this intermediate model the MAST tool generates models for rate monotonic analysis (RMA) or simulation.

PALLADIO Palladio [14] [15] targets whole development process in component-based development. It includes roles of a component developer, a system architect, a system developer and a domain expert. A system in Palladio is modeled by a set of models where each model covers a different role. A component developer annotates each provided service of a component with an additional specification called Resource Demanding Service Effect Specification (RDSEFF). RDSEFF describe a simplified control flow of the service, and they can express the service's dependencies on input parameters and resource demands on abstract resource types stored in the global resource repository. Service Effect Specifications (SEFF) in [15] describe how a provided service of a component calls its required services at some level of abstraction. In [14], 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. Palladio focus only on performance-relevant extra-functional properties, for whose specification it provides random variables to specify extra-functional properties. Additionally, Palladio takes into account usage profiles. The so-called PCM-Bench tool allows independent graphical modeling for all four developer roles. Model transformations map the whole model into performance models (Queueing Networks, Layered Queueing Networks) which are solved by simulation or analytical analysis.

QML/CS The extra-functional properties of a component or of application are as important as its functional ones. There exists a volume of literature regarding the formal and precise specification of extra-functional properties. Service-level agreements (SLAs) [99] are used in a large domain to formally describe the extra-functional properties. However, SLAs focus only on the specification of the properties required of a service as a whole. In [114] the authors proposed a new specification language QML/CS that can be used to model extra-functional product properties of components and component-based software systems. The formalism of QML/CS is based on extended temporal logic of actions (TLA+) [64]. In this formal semantic framework for the specification of extra-functional properties of component-based systems, they propose formal specification of extra-functional properties for different parts of the component-based systems such as components, resources and containers. They also give formal definition for container strategies and the formal definition of intrinsic and extrinsic extra-functional properties based on intrinsic and extrinsic measurements by using context models. There are two kinds of context model: *Service* defines a service operation and will be used to define the response-time measurement, and *Component* defines a component operation and will be the basis for defining the execution-time measurement. The context models are defined in TLA+ and applied for defining measurements. There are three kinds of measurements defined in the framework: *execution time* is an intrinsic measurement based on the component context model, *response time* is an extrinsic measurement based on service context model, and *inter-request time* is an extrinsic measurement based on service context model.

Thot In [96], the authors apply TCTL (Timed Computation Tree Logic) to describe the QoS (Quality of Service) contract of components. They propose timed patterns to ease the addition of time constraints to behavior, such as, response time, delay, execution time, period of service call, duration. The behavior is defined by a process algebra with In and Out Automaton model[68]. The timed contracts are described by using TCTL semantics and verified by the Kronos tool [4].

However, in this approach and in the QML/CS approach discussed above, there is no support for expressing the time information using probability distributions. In the previous section, we have represented the measurement model to independently construct the monitors for measuring performance aspects of interest. In the following we discuss the expression of extra-functional properties monitoring that are monitoring of constraints on measurement models. We can describe the QoS contracts in terms of probability distributions or statistical nature.

Service Level Agreements (SLAs) [20], or contracts, define the obligations and rights between the client and the provider, with respect to the function and the Quality of Service (QoS). Classically, contracts are formulated as hard bounds on some QoS parameters (like in approaches discussed above). For instance, response times are required to be less than a certain fixed value. When composing hard contracts, the simple composition rules are used such as addition or maximum. In many cases this approach based on hard bounds does not fit the reality well. In fact, users would find it more natural to soften contracts, for instance the average value of a response time is less than T milliseconds, or a response time is less than T milliseconds for 95% of the cases, or accept a throughput not larger than N queries per second for 90% of a time period of M hours, etc. For example, WSLA [58] [75] [100] are examples to specifying contracts of web services. In our work, the use of CPN allows us to construct a simulation monitoring of these soft contracts based on measurement models.

Grassi2006 [46] presents an approach for the prediction of extra-functional properties of component-based systems. The approach uses UML stereotypes to label component connectors, for example, to realize a static or dynamic synchronous client/server interaction. From the model-driven point of view, the approach supports the connector refinement transformation from high-level UML 2.0 architecture models into the KLAPER modeling language [45]. The KLAPER language enables transformation from a variety of design specification types into models for different types of performance and reliability analysis. The construction of a model that supports some specific analysis methodology such as Markov chains and queueing networks for performance analysis is seen as the result of a sequence of refinement steps starting from KLAPER models. The authors propose a library of parametric connector behaviors models in KLAPER for each of the stereotypes. These parameterized behavior models replace the component connectors of the KLAPER model of the whole architecture. KLAPER allows modeling of heterogeneous multi-processor platforms, and provides tools for transformation of the UML-based component and assembly specifications into the models for various types of analysis. In addition, KLAPER supports multiple types of performance analysis techniques and both synchronous and asynchronous communication styles. However, KLAPER has the following limitations:

- The software tools for modeling and computations are not available.

- All the service requests are modeled using a FIFO policy. There are no other available scheduling policy.

- The reconfiguration of bindings is forbidden.

2.3.2 Prediction approaches based on measurement

The main benefit of measurement based approaches is that they can offer accurate measurement results for running systems and give more realistic results. For example, these approaches can help developers to find performance bottlenecks in a given system or to identify implemented performance anti-patterns. An anti-pattern is a solution to a problem that leads to negative consequence. Performance anti-patterns provide a way to detect and correct performance problems as well as building performance intuition in developers. The occurrence of a performance anti-pattern in a model may induce performance problems. An anti-pattern includes bad practice specifications and actions to take to solve performance problems. Based on measurements, each potential anti-pattern can be investigated in detail to detect anti-patterns that have well-known solutions. However, these methods can not be used during system design time but are only applicable after the system implementation and deployment phases.

Paper [26] proposes a performance predicting approach for J2EE systems. The authors take middleware details into account and consider J2EE applications with EJB containers. In the first step, performance measures are collected empirically by creating a product-specific performance profile that describes how components of the middleware product affect performance. Then the results obtained are elaborated to extend their validity to a generic manner that is not related to any particular application requirements. Using the profile collected, it is possible to use a set of generic mathematical models to predict generic behavior of applications. The approach proposed includes a reasoning framework for understanding architectural trade-offs and their relationships to specific technology features. The performance of server side is characterized by inputting into the generic performance models the parameter settings concerning client loads, business logic complexity, database requirements, etc.

Paper [84] proposes a component performance assurance solutions framework for performance monitoring for J2EE systems with EJB components. In the first step proxy components are added for each EJB to send time stamps for EJB life-cycle events (such as startup or invocation) to a central dispatcher. Using proxy components performance measures of the running application can be collected. Based on the performance measures, in a second step the approach allows generation of UML models with SPT annotations. Users then specify different workloads in the models to assess various scenarios. Finally, users get performance predictions of the modeled scenarios by using existing simulation techniques.

2.3.3 Colored Petri nets and CPNtools

2.3.3.1 Colored Petri nets

An ordinary Petri net has no types and no modules but only one kind of token and the net is flat. On the contrary, Coloured Petri Nets (CP-nets) allow hierarchical modeling, data types, and complex data manipulation. Each token has a data value attached, called the token color. The token color can be modified by the occurring transitions.

A Coloured Petri Net is a tuple CPN = (Σ, P, T, A, N, C, G, E, I) satisfying the following requirements:

- Σ is a finite set of non-empty types, called color sets
- P is a finite set of places
- T is a finite set of transitions
- A is a finite set of arcs such that: $P \cap T = P \cap A = T \cap A = \ emptyset$
- N is a node function, which is defined from A into P x T ∪ T x P
- C is color function, which is defined from P into Σ.
- G is a guard function, which is defined from T into expressions such that: ∀ t ∈ T: [Type(G(t)) = Bool *wedge* Type(Var(G(t))) ⊂ Σ].
- E is an arc expression function, which is defined from A into expressions such that: $\forall\ a \in A$: [Type(E(a)) = $C(p(a))_M S$ ∧ Type(Var(E(a))) ⊂ Σ] where p(a) is the place of N(a).
- I is an initialization function, which is defined from P into closed expressions such that: $\forall\ p \in P : [Type(I(p)) = C(p)_M S]$

2.3.3.2 Stochastic Petri nets for component-based system design

Recent research results have shown that queueing networks, stochastic process algebra, stochastic Petri nets are the best quantitative models for performance modeling and analysis used for Component Based Software Performance Engineering (CBSPE) [18]. In this thesis, we apply colored Petri nets to manage time-related performance modeling and analysis.
More precisely, our analysis approach relies on stochastic colored Petri nets such as the ones supported by the CPNTools software [54]. Colored Petri Nets (CPN for short) is a discrete-event modeling language that extends classical Petri nets by allowing definition of actions with the Standard ML functional programming language. By combining classical Petri nets and Standard ML language, the CPN language gives much more power of modeling. Classical Petri nets allow modeling of concurrency, communication, synchronization and conflict. Standard ML aims at defining data types, describing data manipulation, and creating compact, parameterizable models. CPNTools is an open source tool for constructing

and analyzing CPN models.

Petri nets are systems that are inherently concurrent, and they make composition of concurrent systems easier compared with automata composition. Stochastic Petri nets add the possibility to describe alternative behaviors such as failure and recovery behaviors, with a quantitative notion of probability of occurrence of these behaviors.

Hierarchical Modeling: Colored Petri nets allow of hierarchical modeling thanks to substitution transitions. A substitution transition is a transition that can be refined into a subnet, therefore providing multiple levels of abstraction.

As usual, designers can use hierarchical nets to construct systems with either bottom-up or top-down approaches. In a bottom-up approach, designers create the most detailed parts of the net first. Later, these parts created will be replaced by substitution transitions to abstract their details and provide a higher level of abstraction. Gradually, we can eventually construct the top-level net that gives a broad overview of the system. Models for component based design have a hierarchical nature very similar to hierarchical Petri nets. Below is a simple example well-suited to briefly introduce component/CPN mappings. This example describes a simple architecture made of two senders sending packets periodically to a server.

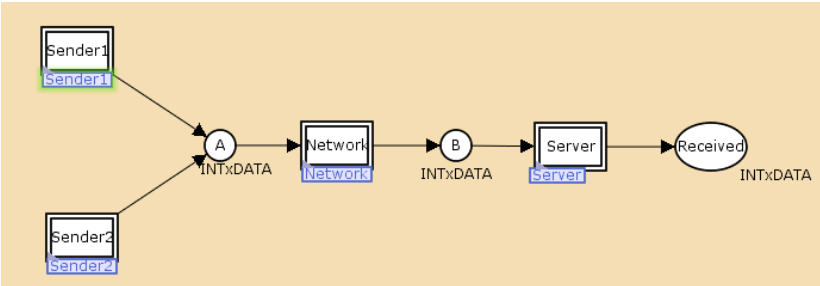

Figure 2.8 – The most abstract model

Figure 2.8 represents the most abstract model of the system. This overview tells us that the system includes two senders: a network and a server. In other words, the two senders periodically send packets that the network broadcasts to the server. To support hierarchical structures, CPN Tools relies on hierarchical tags: these tags are associated with places and transitions in the hierarchical CPNs. There are three kinds of hierarchical tags: subpage tags, port type tags, and fusion set tags. Subpage tags denote substitution transitions, which in turn can be interpreted as sub models when they are detailed in sub pages. The superpage concept can denote a model at higher level of abstraction and it contains subpages. Figure 2.8 shows the superpage, in which there are three substitution transitions named Sender1, Sender2, Network and Server. Port type tags are associated with port places on subpages (or submodels). Subpages and superpages are connected by two places that contain the same kind of colored tokens and anything that happens to each of these two places also happens to the other. Otherwise, these two places are essentially the same. A place is called a port place if it is on a subpage, and a place is called socket place if it is on a superpage. In Figure 2.9, places A, B and C are socket ports, and the places A, B, C in the subpages (e.g places A and B on the

subpage in Figure 2.9 is the subpage associated with the substitution transition Network) are port places respectively. There are three kinds of port type tags: in tags, out tags, and inout tags. Figure 2.9 sketches the network part, which has a transition that can transmit packets to the server, the transmission time being specified by an exponential distribution. The main idea is that we use Petri nets as the abstraction of control flow through the component. Petri nets therefore play the same role than Service Effect Specifications (SEFFs) in [15]: they describe how a required service of a component calls its provided services at some level of abstraction.

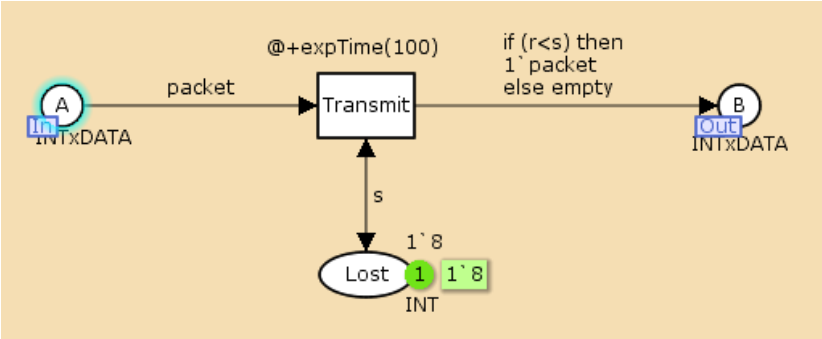

Figure 2.9 – Network

2.3.3.3 Time analysis

CPN-nets can be extended with a time concept. This means that the same language can be used to investigate the logical correctness (obtain the desired functionality, absence of deadlocks, etc) and the performance prediction (for instance, predict mean waiting times and average throughput). In a timed CPN-net each token carries a color (data value) and a time stamp (telling when it can be used). Time stamps are specified by expressions:

- time stamps can depend on color values;
- time stamps can be specified by probability distributions. CPNtools supports several types of probability distributions. Users can specify fixed delays, interval delays and exponential delays.

2.3.3.4 Access/CPN

CPNTools is written in an academic language called Beta that is no longer supported today, and the simulator back-end is written in Standard ML, making integration of CPN models into external applications rather difficult. CPNTools has gone open source and an Access/CPN framework has been developed. Access/CPN consists of an interface written in Java, which provides an object-oriented representation of CPN models. Designers can load models created using CPNTools, with an interface to use the simulator (Access/CPN does not currently support monitoring functionality, but this can hopefully change in near future). Figure 2.10 shows that the architecture of Access/CPN consists of a Java interface (middle)

that provides an object-oriented representation of CPN models, a loader to import models created using CPNTools, and furthermore it also provides ability of performing simulation. The CPN metamodel is split in four different packages: a declaration package, a *cpntype* package, a graphics package, and a model package. The implementation of the CPN metamodel relies on the Eclipse Modeling Framework (EMF). Figure 2.10 shows the CPN model package. In a nutshell, a CPN model contains a Petri net comprising one or more pages. As mentioned earlier, pages show detailed views of a substitution transition. Pages can contain any number of Arcs, Places, Transitions, each containing appropriate Labels, such as name, place type, or arc inscription). Pages also contain *instances*, which are substitution transitions. ParameterAssignements correspond to port/socket assignments. Furthermore, the *cpntype* package holds all the data types supported by CPN tools.

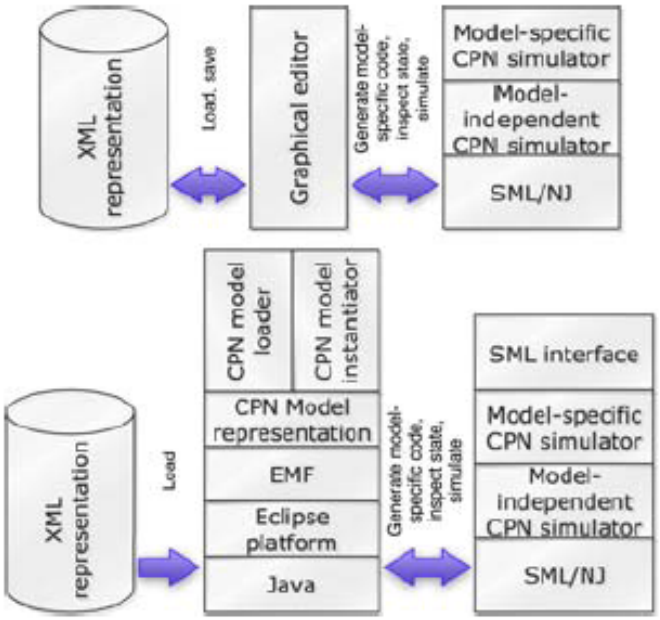

Figure 2.10 – Access/CPN tools

2.4 State-of-the-Art summary

In this chapter we have presented the terminologies in the domain of dynamic component based software and the different performance prediction approaches for component-based systems. The performance prediction approaches presented in this chapter enable design-time property prediction for systems designed from a set of compliant components. Here, we have focused mainly on their working principles, benefits and limitations. However, none of them focused on the performance prediction process for models at runtime. There are only some of them use the Model-Driven Engineering (MDE) for an automatic evaluation process, such as Palladio[15] KLAPPER [44]. In addition, most of these approaches propose the UML-based language or the proprietary metamodel, which lack the concurrency semantics

and the formal techniques. The combination of these aspects are essential to design of the dynamic distributed systems. Apart from the above, the motivation is the framework should be suitable for modeling the distributed systems with a powerful expression language and support the formal techniques. Also, the framework should enable a reasonable time-effect analysis to support the adaptation at runtime.

Consequently, the limitations mentioned above led us to the design of a new framework that features the following properties:

- Modeling semantics for specifications of various performance properties of individual components. A third-party software component could be deployed in different environments. The need to consider its potential execution environments and incorporate them into the system performance modeling makes the problem more difficult. Additionally, modeling of complex systems needs a powerful formalism for modeling, with for instance the capacity to describe concurrency, synchronization, and mutual exclusion of shared resource modeling, etc.

- Well-defined semantics and rules for assembling the performance-related models of individual components in an automatic way. The performance specification of software components and assemblies is a basic problem that must be solved to enable system assembly out of individual components. The description of performance aspects of the system-level mode should be derived from the description of performance aspects of individual components, so that the system-level model could be available for analysis and the performance aspects of the whole system will be measured.

- Reasoning framework allowing extraction and computation on the performance analysis results. The performance results should be used within an appropriate delay to adapt to the continuous changes of highly adaptive systems.

- Time-related performance analysis for evaluation of alternative configurations. Using models-at-runtime, more than one possible configurations must be evaluated before deciding which one should be deployed. This requires to more computation time for evaluation.

The next chapter explains our proposal: an extension metamodel for Kevoree, based on stochastic Petri nets as an internal time model for prediction.

Chapter 3

Performance prediction model for Kevoree

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 [82] 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 fill 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 [39] [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 3.1. 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 sim-

ulated 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 integration into the global models at runtime architecture.

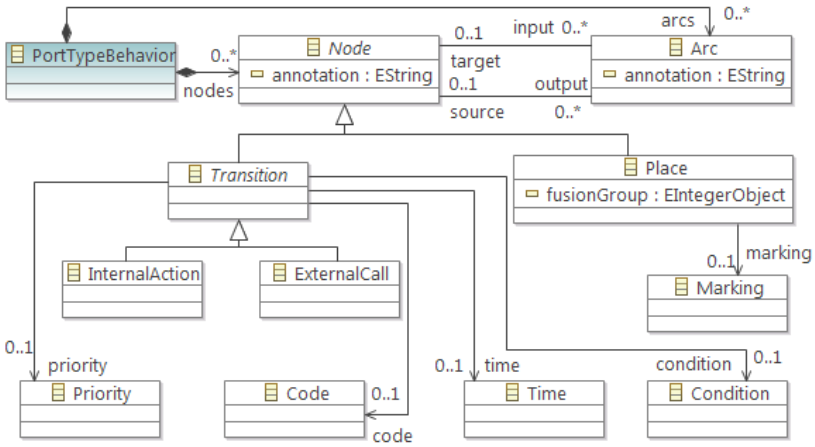

Figure 3.1 – Behavior model specification.

The models for the virtual platform use a specific platform metamodel that eases the transformation from timed component models to simulation models for the CPNtools software [54]. 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.

3.1 Development and validation process model

In the past decade, self-adaptive software system has emerged as one of the most promising approaches to deal with the increasingly complexity of modern software systems and uncertainty of their environments. The system should react to the dynamic changes in order to adapt to the changes in their environments. Such changes includes dynamic changes in availability of memory, network bandwidth, or connectivity and location in mobile systems, and dynamic structural changes, such as hardware, software upgrades. Drastic dynamic changes may impact the performance of the system profoundly. Performance itself is a pervasive property, and is one of the most important property of such evolving systems. Every system aspect can affect the performance, such as resource availability, system's workload, etc. Thus, the designers at design time and the adaptation mechanism need to validate the system model before implementing the assembly model. This task aims at ensuring the correctness and the performance objectives of the assembly models or the configuration generated that attain the design goals. This section discusses about the validation or the analysis of the generated system model. We will give an overview of our approach of continuous design validation, which integrates into the adaptation at runtime process of Kevoree. This approach of the continuous validation aims at helping the reasoning mechanism filtering configurations and

finding the most appropriate one using performance-related analysis results, as illustrated in Figure 3.2.

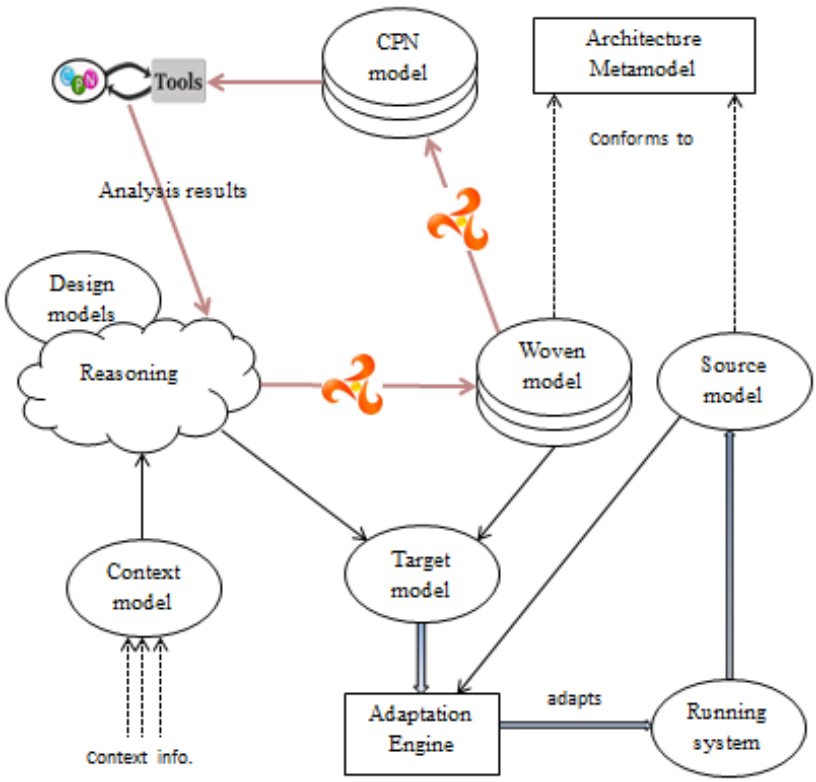

Figure 3.2 – Validation at runtime process

Recently, several approaches have been proposed for the performance analysis of component-based software architectures. However, most of these approaches did not consider the integration into the development process. The work in [63] extended an existing component-based development process model by Cheesman and Daniels to explicitly include early, model-based QoS analyses. In our work we extend the component-based development process proposed in [63] for adaptive distributed component-based systems. Figure 3.3 sketches the main process of our approach. Each box represents a workflow. The thick arrows between boxes represent the change of activity, while the thin arrows point out the flow of artifacts between the workflows. The model allows backward steps into former workflow. We have inherited the requirements, specification, provisioning, assembly, and deployment workflows from the original development process model by Cheesman and Daniels, and the QoS analysis workflow from [63]. To adapt the development process model to the self-adaptive systems from Kevoree, we added the Adaptation specification, Reasoning workflows. The Assembly/Reasoning workflow consists of the assembly at design time by system architect and the automatically generation of new configuration by reasoning engine at runtime. At design time, the QoS analysis workflow is the same as the one in [63]. Component specification, architecture, and use case models are inputs for the QoS analysis workflow. Outputs of

the QoS analysis can be used during the specification to adjust the architecture. The process in Figure 3.3 also includes our extensions to the Adaptation specification and Reasoning workflows that allows to describe the adaptation mechanism at runtime, based on the QoS analysis. In this thesis we mainly focus on performance evaluation, but other QoS properties can also be exploited. The extensions for others QoS properties are out of the scope of this thesis. The detailed description of the adaptation specification workflow has been described in Section 2.1.7. The adaptation specification and reasoning workflows have been described in Chapter 3. Figure 2.3 and 2.4 illustrated the adaptation process.

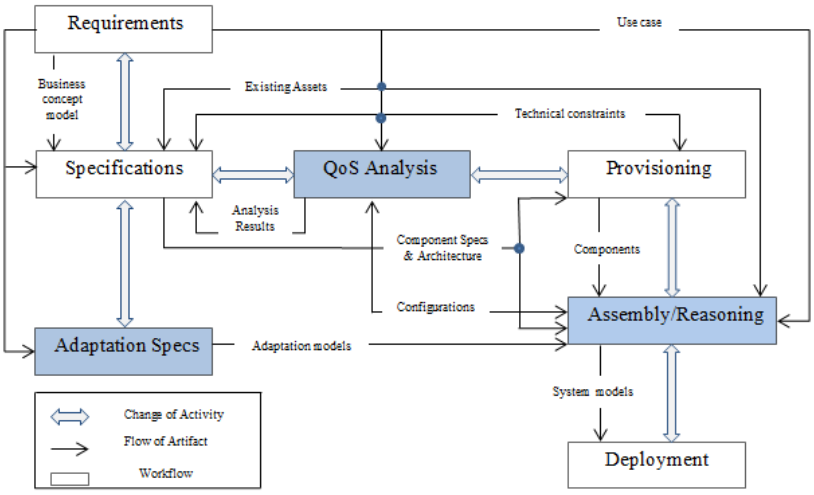

Figure 3.3 – Self-adaptive component-based development process model with QoS analysis

In the following, we will only describe the QoS analysis workflow. The work in [63] specified different roles in the development process, such as component developer, system architecture, system deployer. The workflows of each these roles influence each other. Due to the fact that we mainly focus on the self-adaptive systems, we do not distinguish clearly the roles in development process. Some workflows can be charged by either the system architect or the reasoning engine. Figure 3.4 describes the workflow concerned in QoS analysis development. During the QoS analysis, the software architecture is refined with information on the usage model, the internal structure of components, the deployment context. In offline design process, the QoS analysis model can be constructed by combining the architecture model, the usage model, and the measurement models defined by the system architect. The measurement models aim at defining the QoS metrics of interest. Section 3.3 will detail the concept of measurement model and the aspect-oriented modeling. At runtime, the QoS models can be generated by combining the component models (or new configurations that are generated by reasoning engine) and the usage models (the workload models). The model transformation workflow generates the system-level QoS model (we use CPN-nets as QoS evaluation model). The QoS analysis results can be obtained from simulation on these generated QoS models. Finally, the reasoning engine will select the most convenient and effective configuration w.r.t the QoS objectives.

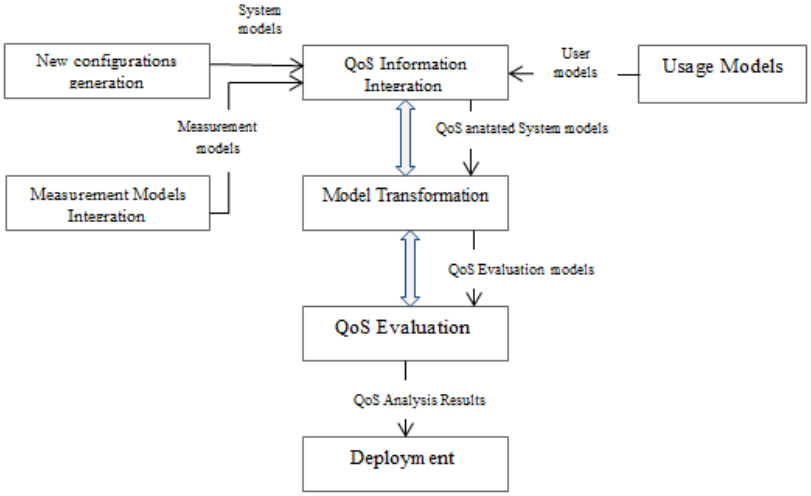

Figure 3.4 – QoS analysis workflow

3.2 Behavior model for Kevoree components

3.2.1 Behavioral CPN meta-model

This section introduces Kevoree Component Model extensions that give developers the ability of modeling component behavior.

The developers will use not only the existing Kevoree Component Model but also the extended behavior model to specify their components including the description of the behavior. This kind of grey box component that was presented in Section 2.1.2 allows developers to specify the contract between components. We will examine thoroughly the specification of our contract meta-model in the following. For the purpose of outfitting the Kevoree meta-model for the contract meta-model, we added the *PortTypeBehavior* class in Kevoree meta-model, as shown in Figure 3.6. Components can describe abstract behaviors through this *PortTypeBehavior* class, which binds to ports. The *ComponentType* meta-class specializes the *TypeDefinition* class, which contains *PortTypeBehavior*.

In the following, we present the proposed formal semantics based on Colored Petri nets to design contracts of Kevoree components. Colored Petri nets are well suited to modeling of concurrent, distributed systems, and moreover they make formal verification more amenable. In the Kevoree component model, direct binding between components is forbidden. Instead, Kevoree provides channel types to describe complex semantics of component bindings especially for N to M bindings in term of multi-cast or distributed FIFO queue. Basically, our contract meta-model must take the following considerations into account:

- timing specification including means for statistical functions such as mean and standard deviation;
- stochastic behavior with several probability distributions;

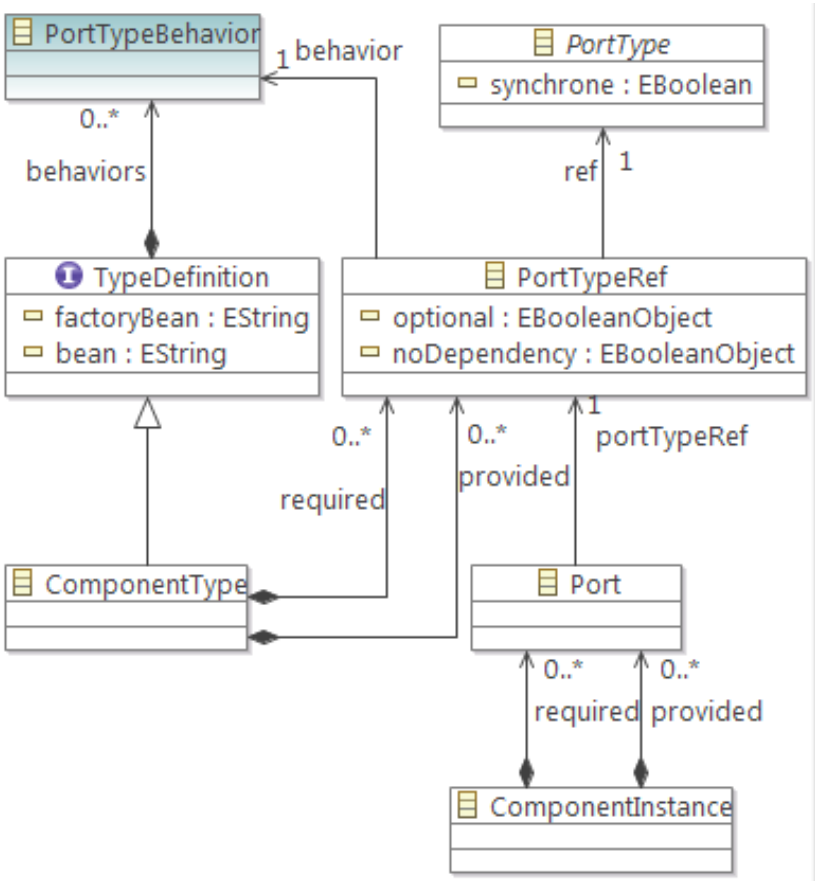

Figure 3.5 – Integration of behavior modeling into Kevoree

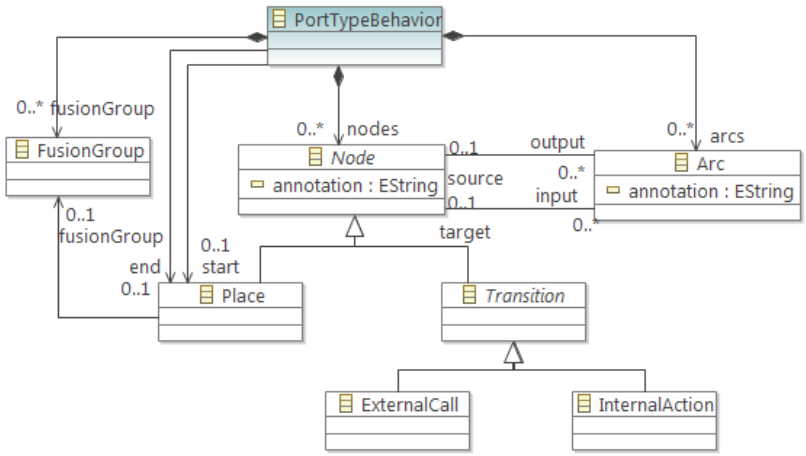

Figure 3.6 – Behavior model for Kevoree

- compatibility with Kevoree channel types, which can involve more than two ports, leading to multiple shareholders in contract definition;
- ease of translation into a CPN tools compatible model.

We now present our formal behavioral notation of Kevoree components. The behavior models may contain start and end places. Each behavior model can contain at most one start place and one end place. Simply speaking, the start and end places can be considered interfaces of behavior models. The behavior models connect to channels through their interfaces. However, from the point of view of developers, their behavior model does not use channels, as they use directly the *external call* element to refer to the required services. However the behavior models connect to channels through their interfaces to implement the communication between provided and required services. Figure 3.6 and 3.5 represents the specification of a behavior model of services.

We can assume that the *PortTypeBehavior* element is equivalent to the *ResourceDemandingSEFF* element of the Palladio Component Model [12] [14]. *PortTypeBehavioral* is the root element of the behavioral meta-model, this behavioral meta-model bears all the modeling capabilities of the CPN language [54, 109]. Our meta-model contains a part for describing types and declarations, as showed in Figure 3.8 and 3.9, and a part for describing the monitor in CPN as showed in Figure 3.7.

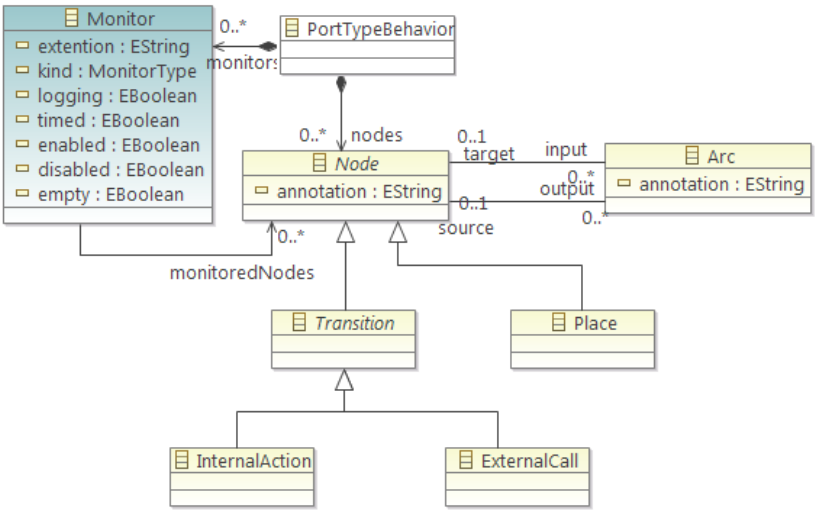

Figure 3.7 – Monitor of CPN nets for capturing QoS measures of the simulation

These parts of meta-model are inherited from the Access/CPN meta-model [108]. A Petri net with a start place and an end place has been considered and defined in several works to model a workflow, such as in [104, 35].

3.2.2 Color type declaration of interface places

The point worth noting here is that there are rules for what constitutes legal interface places (output/input places) of a behavior model. Beyond what is legal, though, we have created

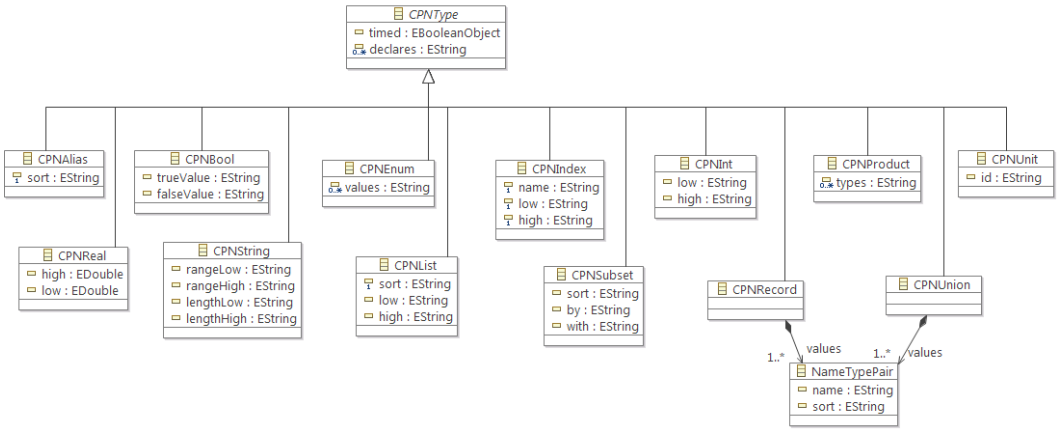

Figure 3.8 – Types used to modeling the CPN behavior models

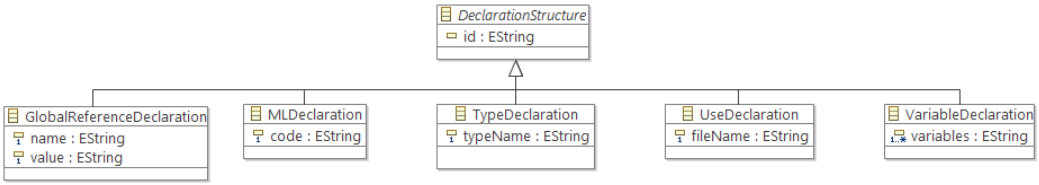

Figure 3.9 – Declarations of color sets and variables defined in the model

conventions for defining interface places. In OOP (Object oriented programming) or CBSE (Component-based Software Engineering) the interfaces are used as a contract of what a class or a component can do, without saying anything about how the class or component will do it, and the interconnected required and provided ports must refer to the same interface. Similarly, the interfaces of the required and provided services in Kevoree behavior models should satisfy conventions for defining interface places. This section provides these conventions for defining interface places.

The definition of interface places is identical to the definition of their color types. The color type declaration varies from designing phase to phase. The interface's color types of components are first declared by third-party developers, and then these declarations of color types may be automatically changed to adapt to the composition of components. The following will represent the rules for declaring color types of interfaces during the component development phase. The model transformation phase is responsible for changing the interface's color types of individual components to make CPN models compatible. We will introduce this model transformation in the next section. In the development phase, the color type of an interface place is defined as a product type of three fields: (a) the source field is the address of the sending service, (b) the data field is described in the format of a record type, which contains the message data and the identity of the message, and (c) the target field is the address of the receiving service. The conventional declaration of the interface place is thus: colset MESSAGE = product ADDRESS * DATA * ADDRESS timed;. The DATA type is a record type that includes an id field providing the identification of the message

and different fields of the message content. The ADDRESS field, which is of String type, specifies the address of the source and target services. To explain these naming conventions, we consider an example of a service as illustrated in Figure 3.10.

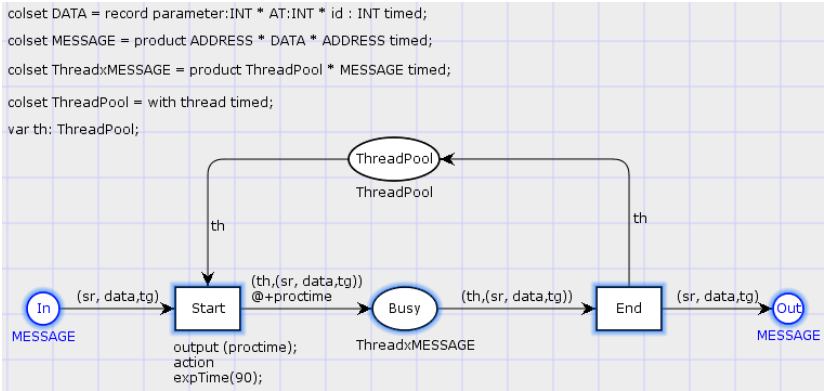

Figure 3.10 – An example of color type declaring convention

The example in Figure 3.10 illustrates how to declare the color type of interface places. New requests arrive periodically in the service implemented in the context of a server that contains a thread pool. If the server is idle when a request arrives then the request is processed immediately by the server. When the server finishes processing a request, it becomes idle. The developer of this service has designed the interface place of type *Data* as follows: *colset Data = record parameter: STRING * AT : INT * id:INT timed;*. As stated in the declaration rules, the color type is a record containing a functional field named *parameter* as an input parameter of the service, and an extra-functional field named AT that could be necessary for evaluating the time related aspects of the system (such as response time). The developers can design many different parameters, and also many extra-functional fields, depending on the information that is necessary to evaluate extra-functional properties of the system. These extra-functional properties are provided by the service, *e.g.* the execution time of transitions, or by security related properties attached to transitions. In the example in Figure 3.10, the developer only provides information about execution time, which could be useful to evaluate the time-related properties of the whole system. Moreover, in this case the developer has declared and used a color named *proctime*. This variable *proctime* is used to evaluate the processing time of the server. The developer can create monitors to evaluate this extra-functional properties of processing time, independently of other elements in the potential composite context. The *AT* variable declared in the interface color type is used to evaluate time-concerned properties, and the monitors used to measure these properties are called *global monitors*. We will present the local monitors and global monitors in detail in the next sections.

3.2.3 Performance indicators injection

Our goal here is to automate the injection of performance indicators, since the simulation is likely to be applied in a model at runtime technique, where performance evaluation is

done on the changing models. This section represents the concept of the monitor, which consists of two types: global monitor and local monitor. The next section explains how to use the defined measurement model and the aspect-oriented modeling to construct the monitors independently of the behavior model.

Definition of monitors: The component developers must describe their component together with a precise description of their extra-functional properties and the description of monitors that will be used by CPN tools to estimate the different measurements of the system. The component developers can not make assumptions on the context in which their components will be used. Hence, the component designers can not define the monitors for measuring the performance metrics that need to observe not only the places and transitions modeled in their component, but also other nodes from other components. In order to tackle this problem, the component developers should first identify and determine all the extra-functional dimensions of interest that can be associated in the component modeling and then provide the monitors for these extra-functional dimensions. Secondly, before modeling the monitors to observe these measurements, the component developers should identify whether the extra-functional dimension is global or local.

To define the monitors for measuring the global extra-functional properties during the modeling of a service specification of a component, the component developer need to take into account the required and provided services of this service. However, the component developer has no knowledge of the definition of these global functional properties and the execution of the required and provided services. Hence, to make these monitors compatible with the CPN net after composing all the components, it is necessary to apply a mapping of these extra-functional properties used in different components. The compositor (or the composition engine) is responsible for this task.

By contrast, to define the monitors for measuring the local extra-functional properties during the modeling of a service specification of a component, the component developer does not need to take care about the required and provided services of this service. Therefore, the composition engine simply adds the color set of these local extra-functional properties into the extra-functional color set.

1. Monitor for estimating the measurements: A monitor is a mechanism in CPN tools that is used to observe, to inspect, and modify a simulation of a CPN net in order to periodically extract information from the markings, binding elements when simulating, and to use this information for different purposes (performance measuring for example). There are several types of monitors:

   (a) Breaking point monitors: used to stop a simulation;

   (b) Data collector monitors: used to extract numerical data from net;

   (c) Write-in-file monitors : used to update files during simulations;

   (d) User-defined monitors: generic monitors used for any purpose.

   For each of the first three monitor kinds, it is possible to save the numerical data extracted from a net during a simulation in a log file. It is also possible to save the

statistics (such as average, max, and so on) that are calculated during a simulation. This kind of performance report is saved as HTML files.

To avoid wasting time to analyze the HTML files, we only use the user-defined monitors to obtain the simulation results directly. We use the Access/CPN Eclipse plug-ins to communicate with the CPN tools simulation. The idea is that instead of using performance report in HTLM format files, we use the user-defined monitors tailored with the modules for communicating with external java processes through the Java sockets, and therefore the monitor will transfer the statistical results after the simulation to the Java processes. All the statistical information will then be transferred to Kevoree reasoners.

2. Global monitor

   A global monitor aims at measuring the global extra-functional dimensions of the system, based on observing and calculating the value of the tokens whose color sets are declared for these global extra-functional dimensions. The global extra-functional properties of a service may be influenced by the execution of their required and provided services. In addition, due to the fact that the components are developed by independent third-party developers, it is worthwhile to apply a mechanism to build a mapping between the different color sets of the same global extra-functional property as defined by different developers. As mentioned before, the composition engine is responsible for carrying out this mapping.

   In the considered use case, we focus on some kinds of extra-functional aspects such as response time, reliability, security, etc. We use the values attached on tokens to specify and evaluate different extra-functional aspects. These extra-functional aspects are represented as numerical attached values on extra-functional token and their values can be calculated cumulatively during execution of the CPN (or simulation of CPN net). For measuring the extra-functional properties, each property should be declared as a field of the extra-functional color set record of the CPN net, so that the value of this property could be calculated during the execution of the CPN net. In doing so, the extra-functional color set is composed of all the extra-functional properties of interest and flow throughout the CPN net of the system assembled after the composition phase.

   As an example, Figure 3.11 visualizes a system model assembled by two components A and B. Once the component assembly (or a configuration) has been made, the reasoner should decide which global extra-functional properties need to be measured before generation of the CPN net of the system derived from component model. In this example, the reasoner needs the measure of the time elapsed between place P0 and P2 to answer the question about the end-to-end response time. Thus, the extra-functional color set is a record that contains a field named AT, which represents the arrival time of the user request. Automatically, the composition engine generate a monitor that is used to measure the time elapsed between P1 and P2. This monitor is 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.

3. Local monitor

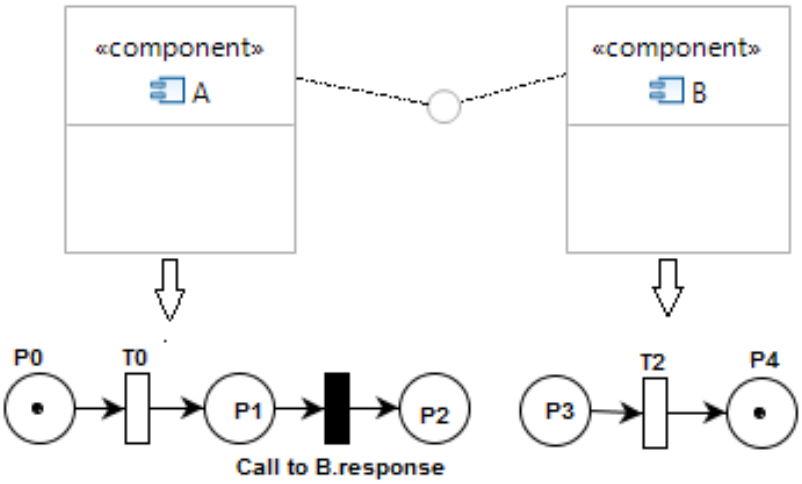

Figure 3.11 – Assembly of A and B components

The monitors of local extra-functional properties of a component service can be modeled in an independent way, during the component implementation. The component developers might predict the necessary of some local performance-relevant properties in the component modeling, they therefore will model their service behavior in a way that includes the monitors to measure these extra-functional properties. The reasoner may decide whether or not to use these local monitors.

3.3 Parameterized templates and aspect-oriented modeling

The preceding sections have illustrated how to use colored Petri nets for creating component performance models. However, constructing performance models of complex systems using colored Petri nets is often not an easy task. Hence, it is necessary to have mechanisms for facilitating performance modeling. We propose to use parameterized templates and aspect-oriented modeling in the development process as presented in the following sections.

3.3.1 Aspect-oriented modeling

Constructing component performance models using colored Petri nets and SML language is not a simple task, specially for complex systems. Adding extra-functional concerns during a design time phase may result in unreadable and complicated Petri nets. It would be useful if colored Petri nets could support the separation of concerns. We propose hereafter a designing technique that relies on the aspect-oriented modeling and the dedicated model transformations supported by the Kermeta tool to improve the component performance model development process. Figure 3.13 and Figure 3.14 represent the aspect-oriented modeling process. Figure 3.12 illustrates the different roles in the development process that based on parameterized templates and aspect-oriented modeling.

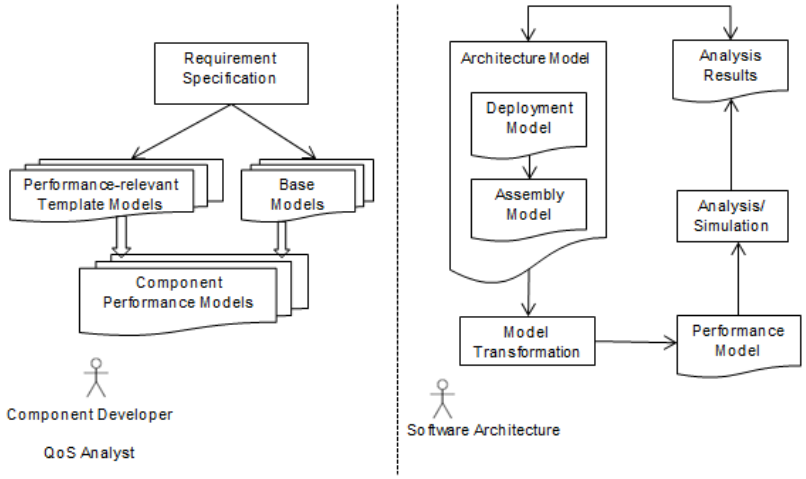

Figure 3.12 – Different roles in QoS-driven development process

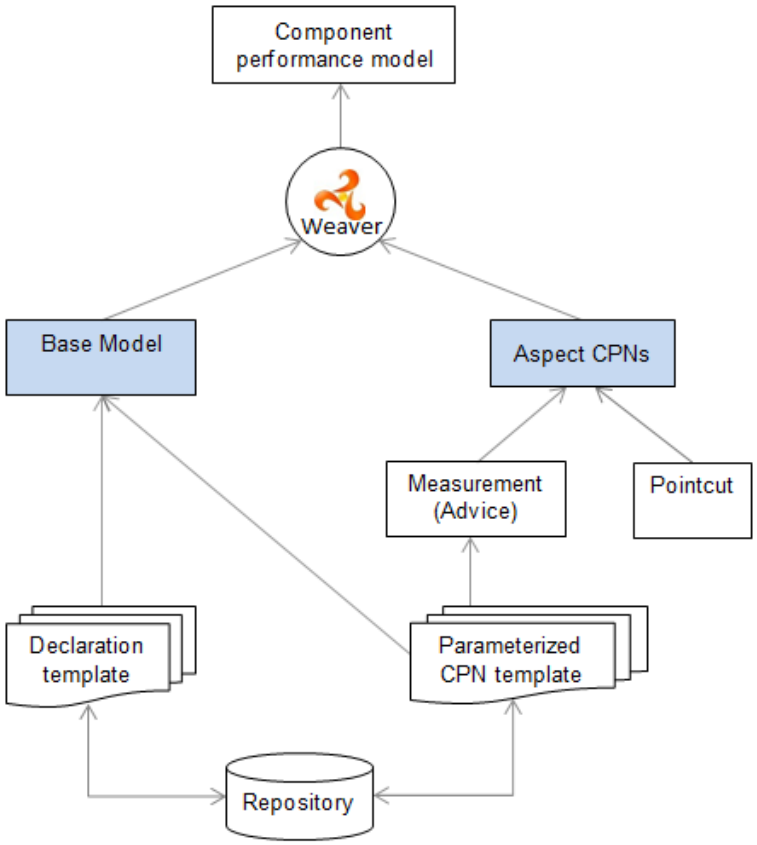

Figure 3.13 – Component performance modeling process based on aspect-oriented modeling

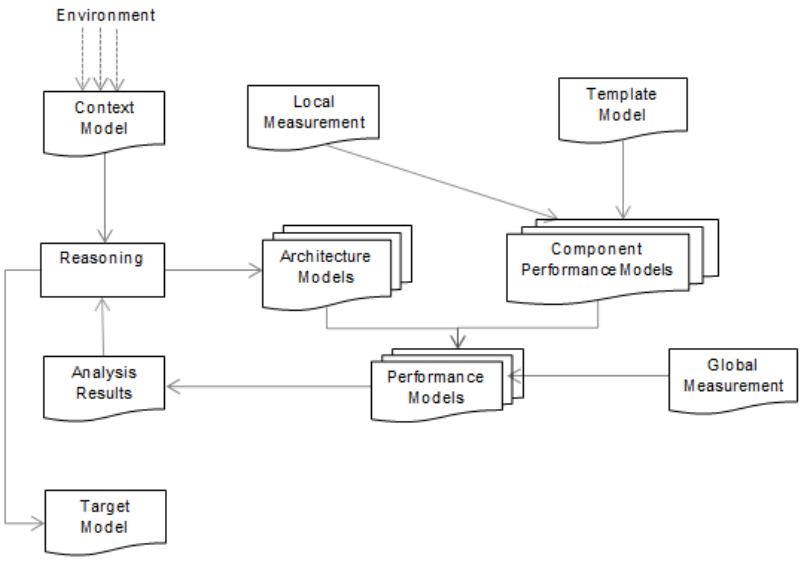

Figure 3.14 – Model composition and validation process on active nodes at runtime

This section discusses the concepts of measurement model and the application of aspect-oriented modeling. A designing model captures both the functional specification and extra-functional specification of the system, by using different conceptual tools. The functional specification is modeled in Kevoree-CPN, and the extra-functional specification is modeled using using parameterised templates. The quantitative measurement models can be specified in and imported from the templates, or they can be created directly in the base model to measure extra-functional aspects.

A measurement model aims at expressing an extra-functional characteristic. It may deal with some functional characterisation of the system such as elements, structures, and behaviors on which the extra-functional characteristic relies on. In [114] the authors use context models to specify measurements independently of their usage in concrete application, the authors defined a context model as a higher level of abstraction of the application model, on which the extra-functional measurement relies. Analogously, we define a specific measurement as an advice model, which is then weaved into the base model. We apply aspect-oriented modeling to allow the separation of concerns and the reusability in Petri nets. In [94] the author applied the aspect-oriented modeling in Petri nets based on the MDA (Model Driven Architecture) [92] approach. The author distinguishes models at CIM (Computational Independent Model), PIM (Platform Independent Model) and PSM (Platform Specific Model) levels by defining base models, advice models and some kinds of join operations that determine the join points to add advice models into base models. In [111] the authors introduce an aspect-oriented approach to model the vulnerabilities by exploring explicit behaviors of security threats as the mediator between security goals and application of security features. They define three different models: (a) base Petri net models, which present the behavior model that contains the intended functions of the system, (b) security threat Petri net models, which explore potential attacks that violate the security goals of system's intended functions,

and (c) threat mitigations of a security design, which are modeled as Petri net-based aspects due to the incremental and crosscutting nature of security features. This approach facilitates verifying correctness of security threats against intended functions and verifying absence of security threats from integrated functions and threat mitigation. In this thesis, we apply aspect-oriented modeling to the continuous QoS evaluation process. We distinguish the base performance-relevant modeling by means of coloured Petri nets, which can use parameterized templates, and the measurement advice models to measure the extra-functional characteristics of the system.

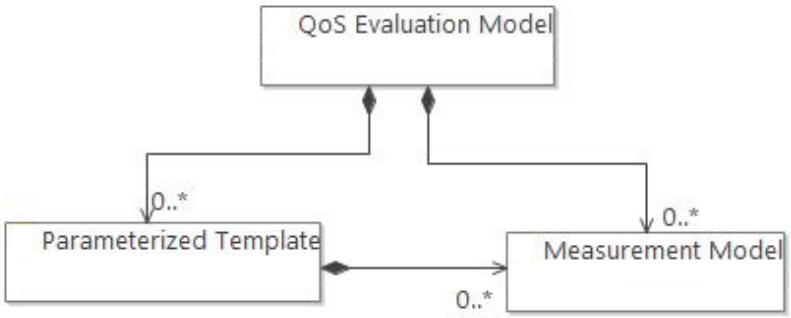

Figure 3.15 – The components of the QoS evaluation model

Figure 3.15 illustrated the relation between the component performance model, the parameterized templates, and the measurement model. One point worth noting here is that instead of defining monitors to capture the performance-relevant information in the simulation, the component developers define the measurement model by themselves or use the templates in the library. By using measurement models, the developers can facilitate the development of their performance model. It would be useful to define measurement models as advice models to be weaved into base models, because the developers can define measurement models independently of the base model and may use these measurement models many times, for different contexts. By simply defining the pointcut, the developers can apply predefined measurement models anytime and anywhere into their base model, without needing to redefine these measurement models. Additionally, the measurement models add the elements, variables, structures needed to capture the quantitative properties, so that the developers do not have to take the definition of monitors that capture the quantitative properties into account when developing their abstract model of its behavior, including timing properties.

The full QoS model of the component could be automatically derived by weaving advice models into the base model of the component and by applying the parameterized templates. The following subsection discusses the fundamental concepts of the semantic framework and the application of the aspect-oriented modeling and the parameterized templates.

3.3.2 Measurement model

A measurement model aims at injecting performance indicators into the base model, by providing an additional model, which is defined by means of a coloured Petri net. Measurement

models are constructed independently of the base behavior model and for the definition of a performance indicator to capture a specific extra-functional characteristic *C*. The measurement model only contains elements that are necessary to explore *C*. Measurement models define the concept of a component, a service, an attribute that are relevant to the extra-functional characteristic. They are more generic and do not model specific components, services, or attributes. Measurement models are defined as advice models and can be weaved into the behavior models by defining pointcuts.

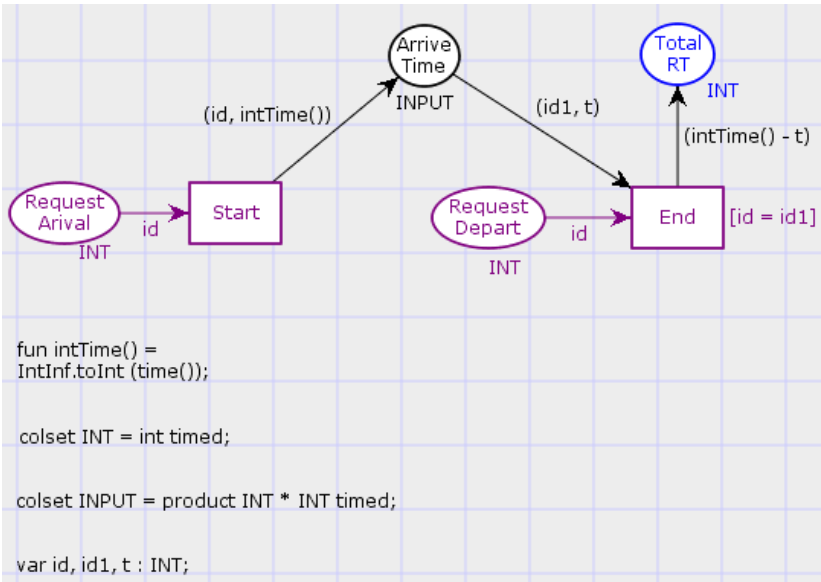

Figure 3.16 – An example of a measurement model defining the response time aspect

Example of Measurement model Figure 3.16 shows an example of a context model defining the response time measurement between two transitions. This model only gives elements that regard response time measurement: there is no reference to any concrete functionality. The context model in this example aims at expressing the response time measurement between two transitions named *Start* and *End*. The place *Arrival Time* stores the arrival time of the incoming tokens that trigger the transition *Start*. As soon as a request with an identity *id* arrives at the place *Request Arrives*, the transition *Start* is fired and a token of type *INPUT* that stores the arrival time of the *id* request will be moved into *Arrive Time* place. The place *RT* contains the elapsed time of the requests between two transitions *Start* and *End*. The function intTime() returns the current time of the simulation:

```
fun intTime() = IntInf.toInt (time());
```

(All code used in a CPN model is written in SML language.) The *id1* variable is the identity of the requests that were transmitted between *Start* and *End* transitions. When the request has been handled completely, the *id* token of the request arrives to the place *Request Depart*. The condition of the *End* transition is to get the arrival time of the request in the place *Request Depart*, and the time elapsed of this request between two transitions can be counted by subtraction of current time to the arrival time of the request:

`(intTime() - t)`

This example of modeling response time measurement presents just one possible way of modeling this aspect of response time measurement. The developers can design the response time measurement model in their own way. Additionally, depending on the goals of QoS evaluation, other extra-functional measurement models may be needed. These measurement models can then be used to measure the quantity properties of the components (local extra-functional properties) and of the components assembly (global extra-functional properties).

```
▼Monitors
  ▼Response_Time
     Type: User defined
    ▼Nodes ordered by pages
      ▼ResponseTime
          End (transition)
    ▼Init
       fun init () =
         if !connected = true
         then (ConnManagementLayer.closeConnection("Conn");
                 connected := false)
        else ()
    ▼Predicate
       fun pred (bindelem) =
       let
         fun predBindElem (ResponseTime'End (1, {id,id1,t})) = true
             | predBindElem _ = false
       in
         predBindElem bindelem
       end
    ▼Observer
       fun obs (bindelem) =
       let
         fun obsBindElem (ResponseTime'End (1, {id,id1,t})) = INT.mkstr(intTime() - t)
             | obsBindElem _ = ""
       in
         obsBindElem bindelem
       end
    ▼Action
       fun action (s1) =
         (if not(!connected)
         then (ConnManagementLayer.acceptConnection("Conn",9000);
                 connected:=true)
         else ();
         send_to_monitor(s1))
    ▼Stop
       fun stop () =
         if !connected = true
         then (ConnManagementLayer.closeConnection("Conn");
                 connected := false)
        else ()
 ResponseTime
```

Figure 3.17 – The monitor for capturing the response time measure

```
▼globref connected = false;
▼fun send_to_monitor(text) =
   ConnManagementLayer.send("Conn",text,stringEncode);
```

Figure 3.18 – The declarations used for the monitor of response time

Measurement models have also an associated monitor that captures the measures of interest during the simulation. The main objective of the measurement model is to define indepen-

dently the monitor that captures the measures of interest during the simulation, and also to increase the reusability and maintainability. Figure 3.18 and Figure 3.17 show the declaration used to define the *Response Time* monitor of the measurement model and the definition of the monitor. The global variable *connected* determines the status of the connection between the monitor and the external Java process. The function *send_to_monitor(text)*sends the text to the external process through the connection named *Conn*.

As explained in the previous section, we only use the user defined monitors of CPN Tools to capture the measures of interest and to communicate with an external process (*i.e.* the reasoner of Kevoree systems). In this example of response time measurement, the monitor is associated with the *End* transition. The monitor is used to measure the amount of time elapsed between *Start* and *End* transitions. The request is represented by a variable named *id*, which represents the unique identity of the request. When the *Start* transition occurs, the request token in the *Request Arrival* place is moved to the *Arrive Time* place. The token type INPUT of the *Arrive Time* place is a Product type: *product INT * INT* that expresses the token's identity and the current time of firing *Start* transition by the token, respectively. When the *End* transition occurs, the response time for the request can be calculated by subtracting the value of the *t* field of the *INPUT* token from the current model time. This monitor can be used to calculate several interesting statistics, such as the average and the maximum time delays.

Predicate function : the predication function for the monitor is invoked each time the transition *End* occurs, and it will return true when the transition occurs.

```
fun pred (bindelem) =
let
  fun predBindElem (ResponseTime'End (1, {id,id1,t})) = true
      | predBindElem _ = false
in
  predBindElem bindelem
end
```

Observation function: this observation function is invoked every time the above predicate function returns true, in other words it will be invoked every time the *End* transition occurs.

```
fun obs (bindelem) =
let
  fun obsBindElem (ResponseTime'End (1, {id,id1,t})) =
  INT.mkstr(intTime() - t)
      | obsBindElem _ = ""
in
  obsBindElem bindelem
end
```

The expression *(intTime() - t)* calculates the time delay for the request that is bound to the variable *id1* when the *End* transition occurs. The observation returns integer values in the format of strings because the parameter of the function

```
send_to_monitor(text)
```

Action function: this action function is invoked every time the above observation function returns a string. It will send the value returned by the observation function through the connection named Conn with the port 9000.

```
 fun action (s1) =
  (if not(!connected) then
    (ConnManagementLayer.acceptConnection("Conn",9000);
     connected:=true)
   else ();
   send_t_monitor(s1))
```

Stop function: This function is invoked to close the connection named Conn when the simulation has already finished.

```
fun stop () =
  if !connected = true then
    (ConnManagementLayer.closeConnection("Conn");
    connected := false)
  else ()
```

As explained above, the measurement models define the concepts of component, service, and attribute that are relevant to the extra-functional characteristic of interest. In the example above in Figure 3.16, the elements in brown are the abstract concepts that are relevant to the extra-functional properties of interest. The transitions *Start* and *End* are the elements for which we want to measure the time elapsed between them. The *Request Arrival, Request Depart* places, the *Start, End* transitions and the variable *id* are used to determine joint points in the behavior models. The *id* element defines the *id* of the token that is transmitted from *Request Arrival* place to the Start transition. The other elements of the measurement model are additional elements used to define the monitor of interest. Developers can apply the measurement models into their behavior model by mapping the abstract elements to elements of a concrete context in their behavior model. In the next sections, we introduce the approach of aspect-oriented modeling to define the measurement models as advice models and to apply the measurement models by defining the pointcuts.

3.4 Parameterized CPN templates for Kevoree

While stochastic colored Petri nets are a powerful means of timed behavior specification, they are too fine 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 specification. In [86] the authors have proposed a set of empirical design patterns for modeling process-aware information systems, communication protocols, embedded systems, distributed systems, 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.

In the previous sections, we have shown that behavioral modeling of software component and software systems in term of colored Petri nets has proved to be a good platform for capturing critical information in real-time, reactive, concurrent, and distributed systems. However, in today's software development, not everyone is familiar with the colored Petri nets. Therefore, in this section we undertake the challenge of usage of a set of patterns in colored Petri nets in the pattern format. The patterns aim at making life easier for developers. Developers and experts working in the same domain experience similar difficulties while solving the same kind of problems. A library of patterns can be collected from the existing experiences. Developers often use patterns in the library to build their models efficiently, while avoiding reinventing already existing solutions to problems. In [86], the authors have proposed a set of 34 empirical design patterns for modeling of process-aware information systems, communication protocols, embedded systems, distributed systems, etc. These patterns serve to resolve problems appearing during modeling by means of CPN, and they have been documented in a format that helps developers to easily understand and apply these patterns into their own problems. The developers have to understand and then reconstruct the CPN models by applying these patterns. However, reconstructing these patterns is not a simple task, particularly for the developers who are not familiar with the CPN language. For the purpose of simplifying the developers' work, we propose a library of CPN patterns whereas the patterns have interfaces and parameters. The interfaces of patterns are transitions and places of the patterns that can be connected with external transitions and places. And the parameters of patterns are information that characterizes the instance of patterns. These parameters can be values of tokens, name of external calls (called services), etc. As a result, the developers do not have to reconstruct the whole pattern, they only connect with interfaces of the patterns and modify parameters of patterns to obtain an instance of the pattern that adapt to their goal. Furthermore, the pattern instances may be generated at runtime in some cases. For instance, in the case of the broadcast channel pattern that we will represent in the next parts, the instances of this pattern are derived from the component model, and it depends on nodes connected to the channel.

In the following, we identify a set of design patterns used to correctly model the timed behavior of basic Kevoree components, and furthermore a set of patterns that are useful for developers in modeling their components with the underlying CPN model. Each of the design patterns has a corresponding template that is a CPN segment. Inspired from the Façade design pattern, as mentioned above, the design patterns are represented as black boxes. They represent an abstraction away from the implementation details of the CPN template that may be complicated for ones not familiar with CPN language. Besides, each pattern contains a set of interfaces, which are the transitions or places used to connect with the usage context. Furthermore, in case of necessity to modify some details designed inside the pattern, developers can adjust the pattern template to reach their goal of modeling.

Using the Kermeta model transformation language we instantiate the used patterns from the Kevoree model. To allow such transformation we have extended the Kevoree metamodel

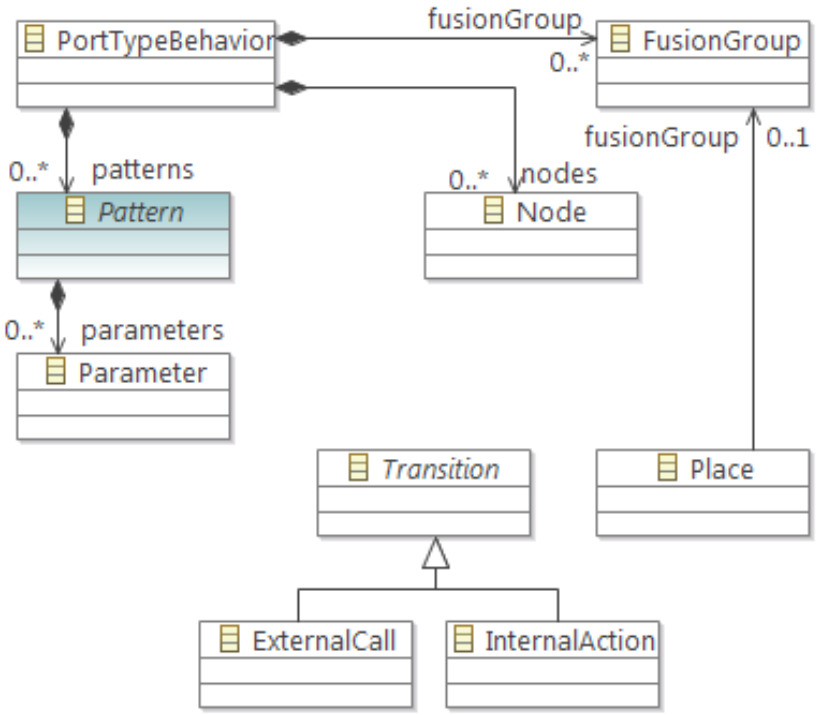

Figure 3.19 – Application of CPN template in Kevoree

with a specific metamodel extension that provides support for pattern definition and reference in component models, as illustrated in Figure 3.19.

3.4.1 Introduction of parameterized templates

In order to facilitate the usage of patterns, each design pattern lies in a template, which represents the generic pattern in using a set of parameters. This means that each template is generic, and independent of any application-specific information. Developers can thus use design patterns template by supplying values for the parameters of the pattern templates. Each pattern template is constructed and stored in a library, an instance of a template will be loaded whenever a developer use the pattern and input the parameters for this pattern. The following section describes a subset of timed behavior templates and how to apply these templates to model timed behavior specification of Kevoree components.

In [69] three kinds of parameterisation were identified: type, expression, and net parameterisation, together with a formal definition of parameterised CPN. In this section we introduce a metamodel extension to create the parameterised CPN contained measurement models, and a subset of templates (or instances of the metamodel) for modeling timed behavior specification. In addition, beyond the three kinds of parameterisation in [69], we identify also some other kinds of parameters, the implementation of these parameters can be realised by transformation tools. The familiarity with Coloured Petri Nets [54] and SML [78] is necessary to describing behavior specification of components, and also the parameterised

templates.

Before going into details of a subset of timed behavior templates, we introduce the concepts of parameters, and the metamodel for patterns with parameters and measurement models. In the following, we will give the ideas of parameterised CPN via a synchronous service call example, and we also provide the metamodel for the parameterised CPN associated with measurement models. Figure 3.19 represents the metamodel for modeling CPN patterns. The pattern instance is implemented by a Kermeta code segment that has parameters of the pattern and a base CPN net as inputs of the transformation, the transformation returns a CPN instance of the pattern. In the following we consider some examples of template used in Kevoree to clarify the concept of parameterized template.

As discussed above, Kevoree defines two different port types: service port type and message port type. The former is a kind of synchronous service, and the latter is a kind of asynchronous service. In the next section we will represent the timed behavior design patterns for synchronous port type.

3.4.2 Synchronous service call template

Objective To allow the transportation of messages between a required service to a provided service, ensuring that the required service which posted a request is blocked until it does not receive the response. This design pattern would be useful while modeling the call to a service that bounds to a service port. Assume that there are two service ports. Required service A needs response messages produced by provided service B (as shown in Figure 3.20). Service A calls service B by sending a requested message, and then the service A should be blocked until it receives the response message from service B. The template could also contain the monitor to measure the response time of the service B.

Solution In order to model the synchronous transportation between two service ports, use two placeholders, which temporally store requested and response messages. The template of this design pattern is given in Figure 3.20. First, the identity token presented in the Token place aimed at ensuring that the recent request can be sent to the service B only after receiving the response for the predecessor request. As interfaces of the pattern, *In* and *Out* places in blue, are marked with *interface* annotation. The *Send* transition consumes both a token representing the external input in the *In* place as well as the identity token in the Token place. The message will arrive to the *Wait* place, and also will be sent to the provided service (that is marked with the *Target* label). The required service will be blocked until a token arrives to *Response* place, and the *Receive* transition will be fired. The response message is then put out to the *Out* place.

Moreover, we furnished the template with a measurement model to measure the response time of the required service.

Measurement model and parameterised pointcut In this synchronous service call template, we also apply the response time measurement model presented in preceding section to measure the response time of the required service. The measurement model is described above in Figure 3.16.

```
Pointcut = (Request Arrival:In, Request Depart:Wait,
Start:Send, End:Receive, id:\# id data, id1:\# id data)
```

Request Arrival and *Request Depart* places in the measurement model are assigned to the *In* and *Wait* places of the template, respectively.

Start and End transitions in the measurement model are assigned to the Send and Receive transitions of the template, respectively.

id and *id1* are assigned to *id* field of the *data* variable of DATA type.

In Figure 3.21, the left-hand side graph represents the *Synchronous Service Call* template, and the right-hand side graph represents the *Response Time* measurement associated with this template to measure the response time of the called service.

Name parameters This kind of name parameter aims at specifying the name of called service. In other words, the name parameter describes the name property of the *ExternalCall* element of the metamodel which abstracts the required service of the behavior specification. For example, in Figure 3.22 and 3.23, the name parameter *requiredService* are assigned to *C*, which implies that the required service is a service named *C*. The expression declarations in the right corner of Figure 3.22 describe the properties required of the synchronous template.

Measurement Modeling parameters The measurement models associated with the template may be also a parameterised model. In this case the developers have to provide values for parameters of measurement models. In this example, the measurement model of the template has two parameters, *id* and *id1* of type *int*; these parameters are used to specify the pointcut of the measurement model, as illustrated in Figure 3.21.

Figure 3.24 shows the *Synchronous Service Call* template used in this example, derived and transformed by the transformation tool, based on the parameters assignment.

3.4.3 Broadcast channel pattern

As mentioned in the previous sections, the Channel entity is responsible for encapsulating inter-component communication semantics. Binding component ports directly is forbidden, as port can be interconnected with channels only. The channel semantics are very diverse, for instance a channel semantics can be broadcast diffusion or distributed transaction, etc. The following part presents some patterns dealing with channel semantics. These patterns include the FIFO queues of channel instances implemented in nodes, and also the communication semantics between nodes. The channel instances will be generated automatically by the model transformation. This transformation will be presented in the next section followed by the introduction of the broadcast channel pattern, and some other kinds of channel.

Pattern name Broadcast Channel Pattern.

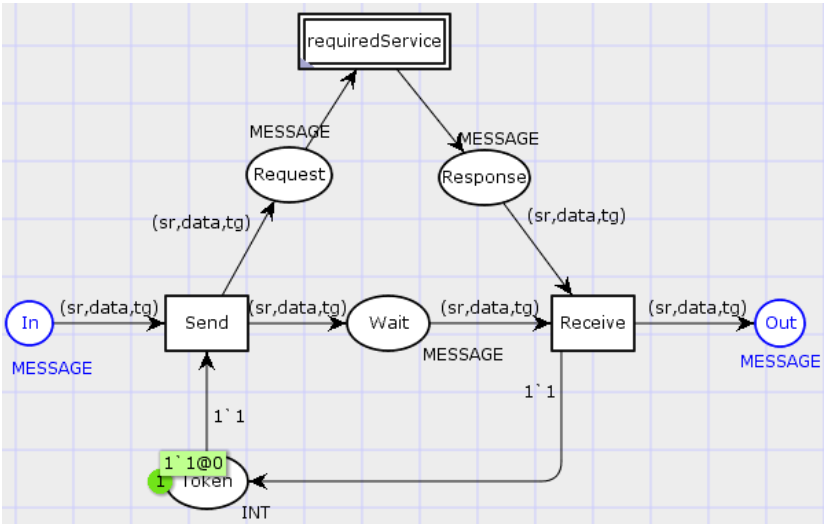

Figure 3.20 – Synchronous service call template

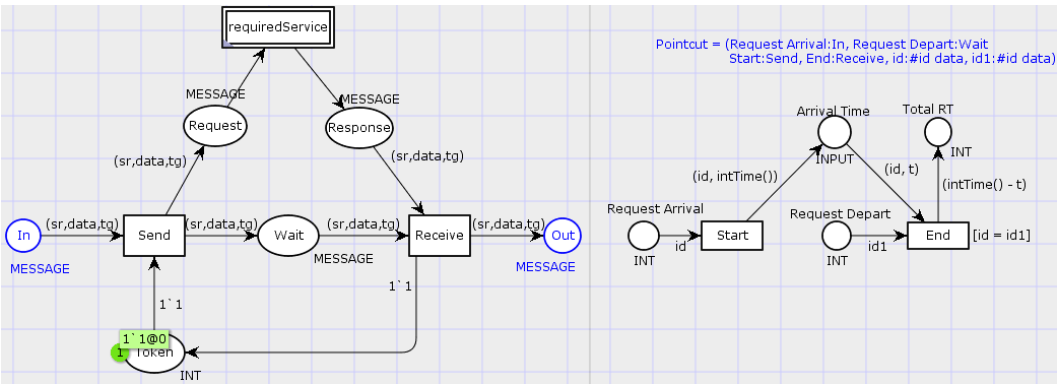

Figure 3.21 – The Response Time *measurement associated with the* Synchronous Service Call *template*

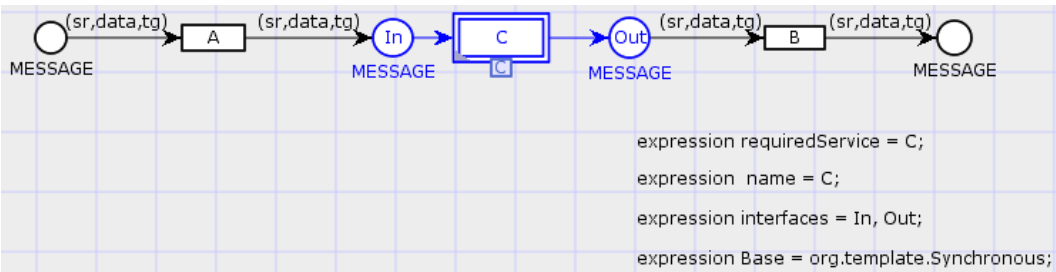

Figure 3.22 – The application of the Synchronous Service Call *template*

Objectives

- To allow broadcasting of messages from single node to multiple targets. In some applications, there is a service hosted on a node, which needs to send a message to other services on the same node or on the other nodes. In this case, we

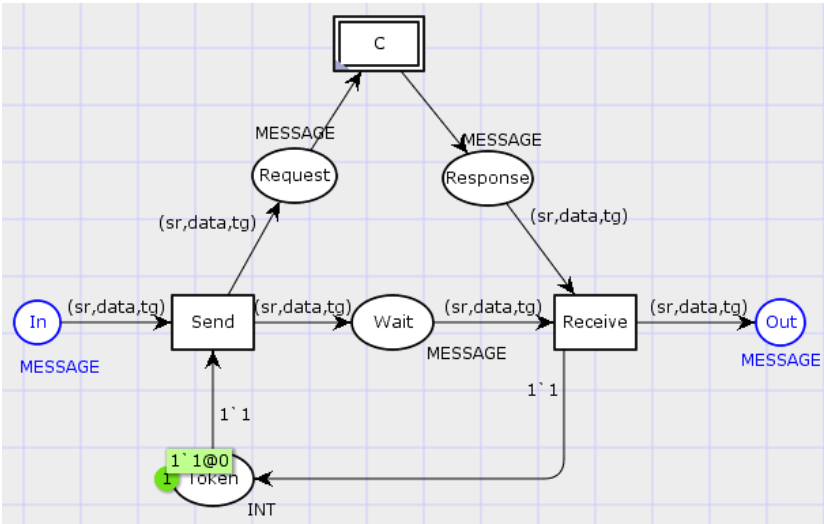

Figure 3.23 – The model after applying Synchronous Service Call *template*

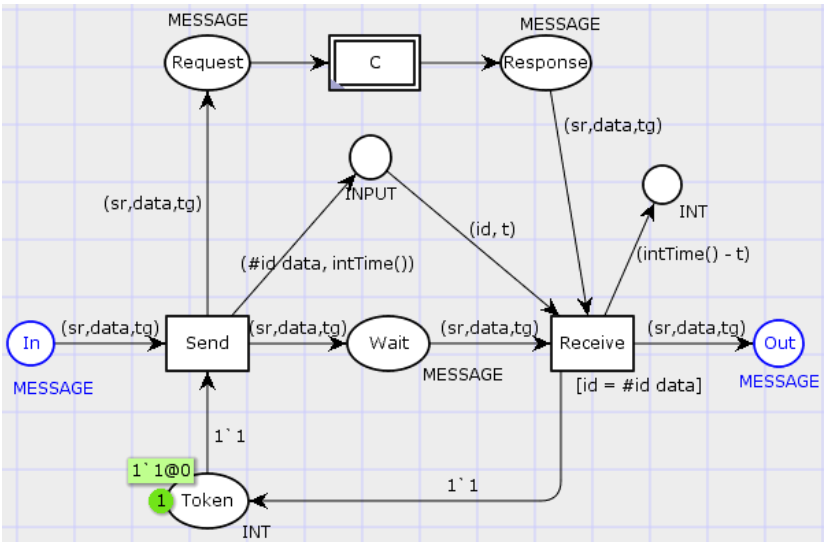

Figure 3.24 – The final result after applying Synchronous Service Call *template and associated* Response Time *measurement*

need to use channels that encapsulate the broadcast communication semantics.

- To provide specification of the latency property of the channel.
- To provide monitor specification to measure the throughput of the channel.

Solution The number of channel instance models generated depends on how many nodes connected to the channel. A channel instance will be generated for each node connected to this channel. Figure 3.25 presents an instance of the broadcast channel pattern. In this instance model, we have four interfaces for which nodes are drawn in blue. In fact,

this instance model is implemented on the node named *host1*. Similarly, there should be also two other instances of the pattern implemented on two nodes *host2* and *host3*. Let us assume that this channel connects a required service named *serviceA* implemented on *host1* and some other required services implemented on two nodes *host2* and *host3*. All provided services that connect to the channel will send messages into *host1* place. Because the instance in Figure 3.25 are implemented on node *host1*, messages will first be sent to services in *host1* that are required services connected to the channel. Hence, the priority of the *serviceA* transition is higher (expressed by the P_HIGH annotation of the *serviceA* transition). Then, the messages will be sent to other nodes. The *host2* and *host3* places are the input interfaces of channel instances implemented on *host2* and *host3* respectively. The type of interface places of the channel instance is *colset MESSAGExPORT = product STRING * MESSAGE * STRING timed;*, which describes the message contents together with the addresses of the required services and provided services. This product type is a triple (source node, message, target port type), the first field of the MESSAGExPORT is a string that describes the source node of the message, the second field is the message content, and the last field is the port type of the received ports. A service receives and processes a message only if its port type is compatible with the port type of the received message.

The instances of the channel pattern will be generated using the aspect programming capabilities of the Kermeta language, and derived from the component model. The details of the transformation will be represented in the next section. Let us explain more detail the instance model of the pattern in this example. The *host1* place is a FIFO (First In First Out) place. The *host1* place will provide messages to the *router* placeholder, and targets will take the message from this place for transfer. The *router* place contains the messages that need to be sent and the targets information. The *router* provides messages to targets through the arcs from the router to targets. The messages are returned back to the router, so that all the targets can get the same information.

The *check* function is responsible for checking the inclusion in the list of nodes information as follows:

```
var host, h: HOST;

fun check(host, []) = false | check(host, h::l) =
    if (host = h) then
        true
    else
    check(host, h);
```

The transition guard

```
[not(check(\"host1\", l))]
```

of the target transitions checks if the message was not yet received by a target. If a target is an element in the targets's list, then this target did not receive this message yet. The transition *del_message* ensures that every target will receive a message once. The total number of targets is derived from the component model. In this example, the number of targets is three. The *len* function returns the length of a list:

```
fun len nil = 0 | len(h::t) = 1 + len t;
```

Parameters

- Latency of the channel: the latency of the network channel is specified by the timed annotation of the *Send* transition, as illustrated in Figure 3.25. This expression parameter can be assigned to a value of a primitive type or a function, for example, in Figure 3.25, the execution time of the Send transition is assigned to expTime(30), which is an exponential distribution function with a mean rate of 30.
- An array of connected services: this array specifies the list of nodes and services that directly connect to the channel. These parameters of the broadcast channel pattern are derived automatically from the component model. In the example above, the array contains services implemented on nodes host1, host2 and host3 together with these nodes.
- A monitor to measure the throughput of the channel: Figure 3.26 represents the monitor to measure the throughput of the channel instance. In Figure 3.26 the socket port os the connection is assigned to 9000 (the connection between the simulation and the external java process, in this case, the simulation connects to the reasoning engine to send the simulation results to the reasoning engine). The monitor is associated with the *Send* transition. The monitor is used to measure the throughput of the *Send* transition. This amount of time the *Send* transition fired the total duration time of the simulation will then be sent to the external process through the port 9000, so that the external process can measure the throughput of the channel instance.

Init function: the Init function for the monitor is invoked at the starting of the simulation, it checks the status of the connection named *Conn*.

```
fun init () =
  if !connected = true then
   (ConnManagementLayer.closeConnection("Conn");
    connected := false)
  else ()
```

Observer function: this observation function invoked every time the above predicate function returns true, *i.e.* it will be invoked every time the Send transition occurs. Each time the Start transition occurs the observation function will return a string of "1".

```
fun obs (bindelem) =
  let
      fun obsBindElem (Broadcast'send (1, {m})) = "1"
      | obsBindElem _ = ""
  in
      obsBindElem bindelem
end
```

Action function: this function is invoked each time the above Observer function returns a value. Each time the Observer function returns a string "1", the Action function will check the status of the connection and then send the value to the external process through the socket port, in this example the socket port is 9000.

```
fun action (s1) =
  (if not(!connected) then
    (ConnManagementLayer.acceptConnection("Conn",9000);
     connected:=true)
   else ();
   send_to_monitor(s1))
```

Stop function: this function is invoked to close the connection named Conn when the simulation has finished.

```
fun stop () =
  if !connected = true then
     (ConnManagementLayer.closeConnection("Conn");
      send_to_monitor(INT.mkstr(intTime()));
      connected := false)
  else ()
```

3.5 Model composition and mapping from CPN-Kevoree model to Access/CPN model

This section presents how to compose component services by means of CPN-Kevoree models using the Kermeta transformation tool in order to achieve a system-level model that can be simulated with CPNtools.

Thanks to the similarity of the semantics of our metamodel and the one of the CPNTools, we only need to focus on the following two problems: how to map our templates onto the CPNTools language structures, and how to manage the possible variable declaration conflicts.

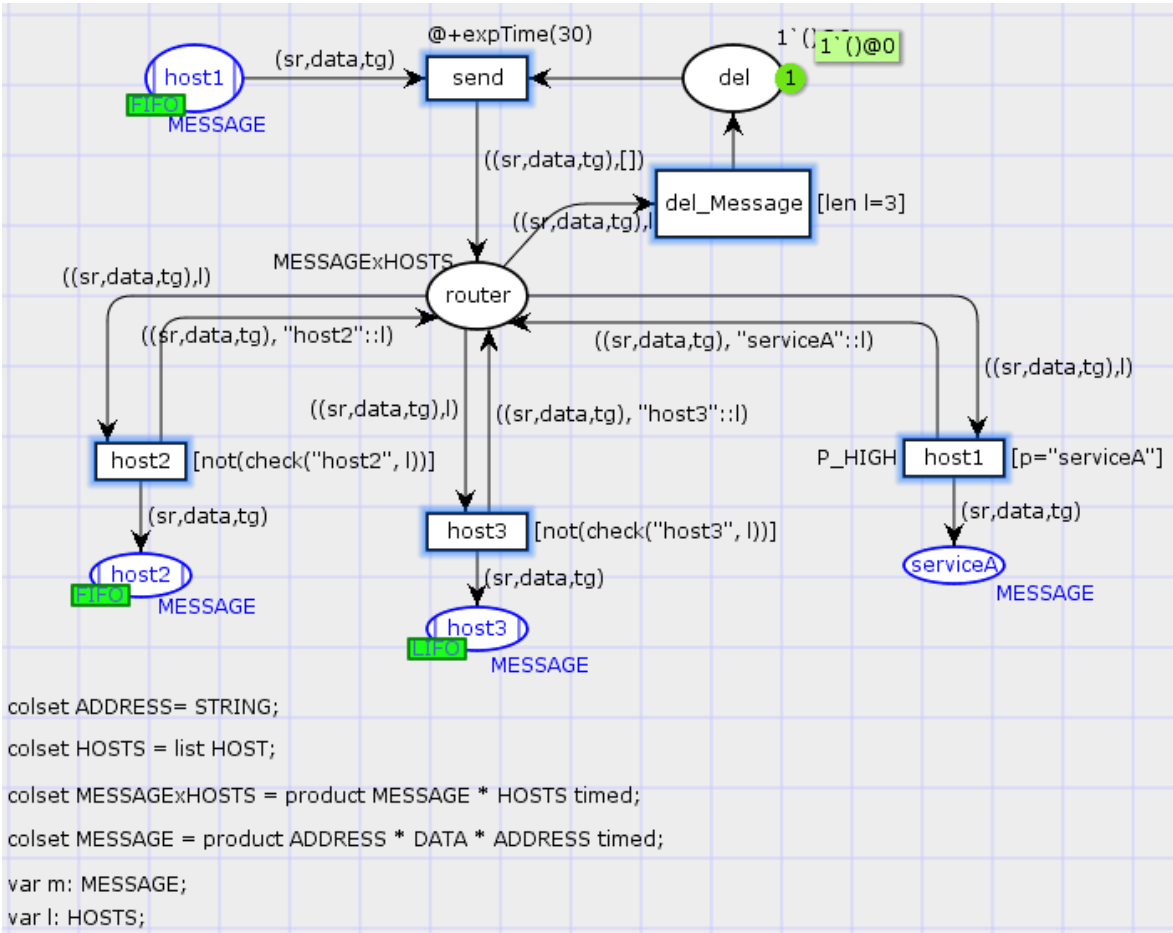

Figure 3.25 – Broadcast Channel Pattern.

The model-to-text transformation is then executed to construct a cpn file that conforms to the DTD file of the CPNTools.

Figure 3.27 shows the transformation process from the Kevoree component model to the system-level CPN net that is ready to be simulated. As discussed in previous sections, the reconfiguration or the generation of new Kevoree component models of the system are realised by the reasoner, based on the DSL model and the reasoning mechanism of the system. Figure 3.28 illustrated the automatic composition of the individual component models and also the generation of the system-level CPN model. This process of generation of the system-level CPN model is equivalent to the first step of the transformation process showed in Figure 3.27. The Kevoree component model generated at runtime by the reasoner conforms to the proposed Kevoree extension metamodel, which strengthens the Kevoree metamodel with the behavioral semantic of the component model by means of our modified CPN language. The modified CPN language has been presented in previous sections. The target CPN model conforms to the Access/CPN metamodel [108]. The Access/CPN [108] metamodel can be found in Chapter 4. The second step (#2) of the transformation process is illustrated in

```
▼Bandwidth
  Type: User defined
  ▶Nodes ordered by pages
  ▼Init
     fun init () =
       if !connected = true
       then (ConnManagementLayer.closeConnection("Conn");
             connected := false)
      else ()
  ▼Predicate
     fun pred (bindelem) =
     let
       fun predBindElem (Broadcast'send (1, {m})) = true
           | predBindElem _ = false
     in
       predBindElem bindelem
     end
  ▼Observer
     fun obs (bindelem) =
     let
       fun obsBindElem (Broadcast'send (1, {m})) = "1"
           | obsBindElem _ = ""
     in
       obsBindElem bindelem
     end
  ▼Action
     fun action (s1) =
       (if not(!connected)
       then (ConnManagementLayer.acceptConnection("Conn",9000);
             connected:=true)
       else ();
       send_to_monitor(s1))
  ▼Stop
     fun stop () =
       if !connected = true
       then (ConnManagementLayer.closeConnection("Conn");
             send_to_monitor(INT.mkstr(intTime()));
             connected := false)
      else ()
```

Figure 3.26 – Definition of a monitor to measure the bandwidth of the channel instance

Figure 3.27, which is the transformation from the system-level CPN net generated in step (#1) to a XML file that is comprehensive by CPNTools. This XML file that CPNtools understands conforms to the Document Type Definition (DTD) of the CPN tools. In this step of model-to-text transformation, we can apply other tasks of post-processing, for example the inclusion of an algorithm of layout (redraw) on the elements of the Petri nets.

The transformation process contains two different types of transformation:

1. model-to-model: The transformation from our proposed metamodel to Access/CPN domain.

2. model-to-text: The transformation to a valid XML that CPNtools understands.

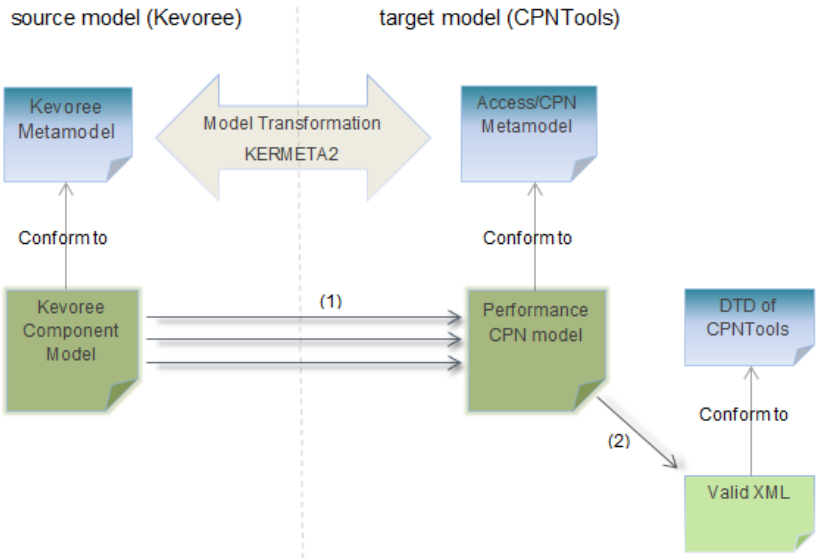

Figure 3.27 – Transformation process from the Kevoree component model to the system-level CPN model

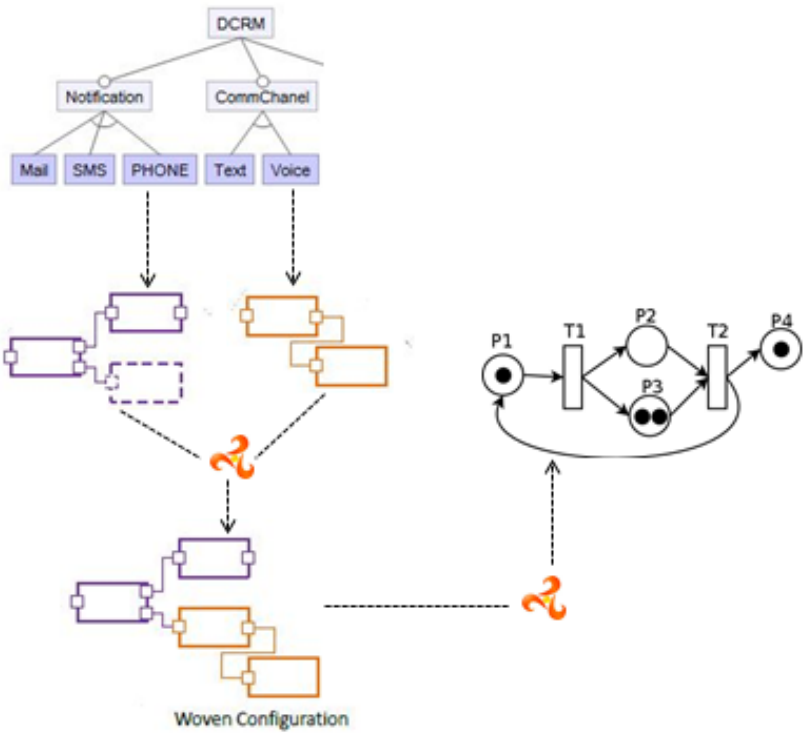

Figure 3.28 – CPN model composition derived from component model

First, we detail the first step of the transformation process. We have defined transformation rules to convert the source domain to the destination domain. Table 3.5 shows the

simplified correspondences between source domain and target domain. The mapping between the Kevoree extension metamodel and the Access/CPN metamodel:

Kevoree extension	Access/CPN
ContainerRoot	PetriNet
PortTypeBehavior	Page
Pattern	Page
ExternalCall	RefTrans
InternalCall	Transition
Transition	TransitionNode

We have implemented these two transformations (model-to-model and model-to-text) with Kermeta [55]. Kermeta is a language built as an extension of EMOF [90] for meta-modeling. Kermeta is an object/model oriented language, allows aspect oriented programming to enrich the model by adding functionality, and also allows design-by-contract [76]. Using Kermeta we have applied the Visitor design pattern [40] to implement our transformation. The code segment below represents the simplified design for the Kevoree-to-Access/CPN model transformation.

```
/*
 * Transformation from Kevoree model to Access/CPN model
 */
package transformation {

abstract class KevoreeTransformation {
operation transform(input: VisitableKevoreeElement): Object is abstract
}

abstract class KevoreeVisitor {
operation visitContainerRoot(k: ContainerRoot): Void is abstract
operation visitPortTypeBehavior(k: PortTypeBehavior): Void is abstract
operation visitPattern(k: Pattern): Void is abstract
operation visitExternalCall(k: ExternalCall): Void is abstract
operation visitInternalCall(k: InternalCall): Void is abstract
operation visitComponentType(k: ComponentType): Void is abstract
operation visitPlace(k: Place): Void is abstract
operation visitChannel(k: Channel): Void is abstract
operation visitArc(k: Arc): Void is abstract

//visit operations for other elements of the metamodel
}
}
```

```
abstract class VisitableKevoreeElement {
operation accept(v: KevoreeVisitor): Void is abstract
}

aspect class ContainerRoot inherits VisitableKevoreeElement{
method accept(v: KevoreeVisitor): Void is do
v.visitContainerRoot(self)
end
}
```

Due to limited space, we only sketch some elements of the Kevoree metamodel as example, the other elements could be designed in the same way. We weave the *accept* method on all the elements that belong to the behavioral extension part of the Kevoree metamodel, an abstract class *VisitableKevoreeElement* describing visitors of the kevoree behavior model, an abstract class *KevoreeTransformation* representing the transformation from Kevoree model to Access/CPN model. Again, due to the limited space, the code segment above only shows the transformation functions of some elements of the Kevoree metamodel. The code segment below shows the simplified transformation code of the Kevoree2CPN class.

```
/*
 * Transformation implementation from Kevoree model to Access/CPN model
 */
aspect class ContainerRoot{
attribute generated: PetriNet

operation getBehaviors(): Sequence<ComponentType> is do
var e: Sequence<TypeDefinition> init self.typeDefinitions.select{e | true}
var res: Sequence<ComponentType> init Sequence<ComponentType>.new
e.each{ type | if type.isInstanceOf(ComponentType) then
                  res.add(type.asType(ComponentType))
            end
       }
result := res
end
}

package transformation{
class Kevoree2CPN inherits KevoreeTransformation{
method transform(input: VisitableKevoreeElement): kermeta::standard::Object is do
var builder: KevoreeVisitorImpl init KevoreeVisitorImpl.new
input.asType(ContainerRoot).accept(builder)

var linker: LinkerVisitorImpl init LinkerVisitorImpl.new
//input.astype(ContainerRoot).accept(linker)
```

```
result := input.asType(ContainerRoot).generated
end
}
```

As can be seen from the code segment above, the *Visitor* of Kevoree metamodel is implemented by two specific classes: *KevoreeVisitorImpl* that will generate the elements of the target Access/CPN model and *LinkerVisitorImpl* that aims to generate links between elements of the target Access/CPN model.

The code segment below shows the simplified transformation code from the *PortTypeBehavior* element to the *Page* element of the Access/CPN model.

```
/*
 * Module for generating the Page element from the PortTypeBehavior element
 */
class KevoreeVisitorImpl inherits KevoreeVisitor{
attribute cr: ContainerRoot

method visitContainerRoot(c: ContainerRoot): Void is do
cr := c
c.generated := PetriNet.new
c.typeDefinitions.each{e | if (e.isInstanceOf(ComponentType)) then
e.asType(ComponentType).behavior.each{f | f.accept(self)}
        end
}
end

method visitPortTypeBehavior(ct: kevoree::PortTypeBehavior): Void is do
ct.generated := Page.new
ct.generated.name.text := ct.name
ct.arcs.each{a | a.accept(self)}
ct.nodes.each{n | if (n.isInstanceOf(kevoree::Place)) then
n.asType(kevoree::Place).accept(self)
else if (n.isInstanceOf(InternalCall)) then
n.asType(InternalCall).accept(self)
else if (n.isInstanceOf(ExternalCall)) then
n.asType(ExternalCall).accept(self)
end
end
end
}
ct.pattern.each{p | p.accept(self)}
end
```

There are two problems when transforming the *PortTypeBehaviorElement* to *Page* element: (a) composing the required and provided services (these are designed by the *ExternalCall* element) and (b) transforming a *Pattern* element into a *Page* element. Firstly, for the services composition, the required service connects to the provided service through the channels. To do so, when transforming an ExternallCall to a Page, for each place that connects to or go out from the ExternalCall transition, we will add an immediate transition between this place and the *start place* of the channel instance model that the required service connect to, as showed in the code segment below:

```
/*
 * Module for generating the place element
 */
method visitPlace(p: kevoree::Place): Void is do
var isFusionGroup:Boolean init false
p.out.each{a | if (a.target.isInstanceOf(ExternalCall)) then
isFusionGroup := true
end
  }
p.in.each{a | if (a.source.isInstanceOf(ExternalCall)) then
end
  }
p.out.each{a | if (a.target.isInstanceOf(Pattern)) then
isFusionGroup := true
end
  }
p.in.each{a | if (a.source.isInstanceOf(Pattern)) then
isFusionGroup := true
end
  }
if (isFusionGroup) then
p.generated := FusionGroup.new.asType(model::Place)
p.generated.name.text := p.name
//other properties of places, like initial marking, color type, etc.
else
p.generated := model::Place.new
p.generated.name.text := p.name
//other properties of places, like initial marking, color type, etc.
end
end
```

The code segment above illustrates the generation of the corresponding Place element in the target Access/CPN model, this place connects to the *start place* of the channel instance through a new added transition.

Secondly, to transform the *Pattern* element to a *Page* element, the places that connect to the used pattern will be translated into fusion places (*FusionGroup* element) in the page generated corresponding to this pattern element, as showed in the code segment above that aimed to generate the fusion place element. If a place connects to a *ExternalCall* instance, it will be translated to a *FusionGroup*.

The second step of the transformation process, as shown in Figure 3.27, could be implemented in the same way. Figure 3.29 represents a simplified view to realise this transformation step to generate the valid XML that the CPNtools understands.

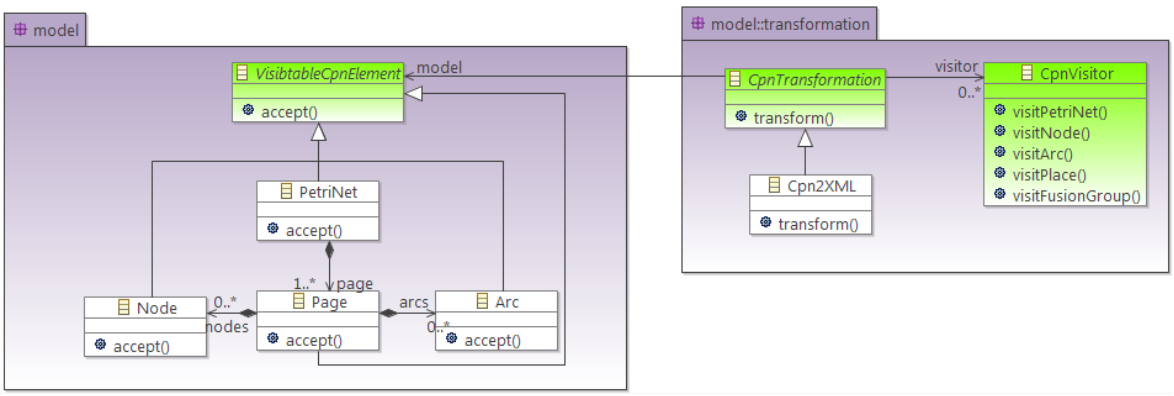

Figure 3.29 – Module for generating the XML code that the CPNtools unerstands

The second step in the transformation process of the framework is the model-to-text transformation, specifically the transformation from Access/CPN model to the XML format that conforms to the DTD of the CPNtools. We can apply an algorithm to redraw (or layout) the generated XML code, but in the scope of our work we only use the default values to layout the generated XML, as showed in the code segment below, which is an example of the transformation from the Place element of the Access/CPN model to a valid part XML.

```
class Cpnvisitor inherits Visitor{
attribute res: StringBuffer

method visitPlace(p:Place):void is do
res:= StringBuffer.new
res.append("<place id=\"\" + Utilities_.random() + "\">")
res.append("posattr x=\"0\" y=\"0\"/> \n")
res.append("<fillattr colour=\"White\" pattern=\"\" filled=\"false\"/> \n")
res.append("<lineattr colour=\"Black\" thick=\"1\" type=\"Solid\"/> \n")
res.append("<texattr colour=\"Black\" bold=\"false\"/> \n")
res.append("<text>" + p.name + "</text>")
res.append("<eclipse w=\"0.00000\" h=\"0.00000\"/> \n")
res.append("<token x=\"0.00000\" y=\"0.00000\"/> \n")
res.append("<marking x=\"0.00000\" y=\"0.00000\" hidden=\"false\"/> \n")
res.append("snap snap_id=\"0\" anchor.horizontal=\"0\" anchor.vertical=\"0\"/> \n")
```

```
res.append("<type id=\"\" + Utilities_.random() + "\">")
res.append("posattr x=\"0\" y=\"0\"/> \n")
res.append("<fillattr colour=\"White\" pattern=\"\" filled=\"false\"/> \n")
res.append("<lineattr colour=\"Black\" thick=\"1\" type=\"Solid\"/> \n")
res.append("<texattr colour=\"Black\" bold=\"false\"/> \n")
res.append("<text tool=\"CPNtools\" version=\"3.5.1\">MESSAGE</text>" \n")
res.append("</type> \n")

res.append("<initmark id=\"\" + Utilities_.random() + "\">")
res.append("posattr x=\"0\" y=\"0\"/> \n")
res.append("<fillattr colour=\"White\" pattern=\"\" filled=\"false\"/> \n")
res.append("<lineattr colour=\"Black\" thick=\"1\" type=\"Solid\"/> \n")
res.append("<texattr colour=\"Black\" bold=\"false\"/> \n")
res.append("<text tool=\"CPNtools\" version=\"3.5.1\">MESSAGE</text>" \n")
res.append("</initmark> \n")
res.append("</place> \n")
end
}
```

Chapter 4

Experiment and validation

4.1 Component model of the application

In this section we present a simplified example based on our ongoing experimental platform, the temperature monitoring part of the firefighters application. We will use this example as a running example to demonstrate the usage of our metamodel by means of the coloured Petri nets language in modeling timed behavior model of components and channels in Kevoree, as well as the performance prediction of timed behavior model of the system as a whole after composing individual behavioral models of components and channels.

This example describes a network of sensors that monitors temperatures outside and inside of the firefighters personal protective equipment. Periodically, sensors send data to the remote server through XBee, Ethernet or 3G networks, which are modeled as a channel in Figure 4.1. In this example, we focus on the following quantitative properties: data transmission time between sensors and server, packet loss rate, throughput of incoming data that comes into the server node. Let us assume that the reasoner has to consider a configuration as illustrated in Figure 4.1 among other different configurations, based on evaluating some performance aspects and verifying the correctness of the behavioral model of the configuration. Periodically the sensors send information on the latest temperature readings to the server, in particular, the *Notifier* component, and then the *Notifier* will extract the abnormal information and forward it to the *Processor* component, which will proceed to react against these abnormal situations. To be more specific, the *Notifier* will verify whether the temperature is in the safety interval. If the value is out of bounds and some situation dependent conditions are met then the *Notifier* will transfer this information to *Processor* component , which monitors abnormal conditions for the firefighters team. Such abnormal information, for instance, is when the external temperature is greater than 100 or the internal temperature greater then 40 and the firefighter did not move for more than 20 seconds. Figure 4.2 shows the class diagram of the application.

The CPN net of the whole system should be derived from the component model of the system. The Kevoree component model in Figure 4.1 tells us that the system includes three sensors, a network, and an emergency management server. Each sensor contains a *captureTemperature* functionality to capture the internal and external temperatures and also the movement of the firefighter. The *Channel 1* models the communication semantics between sensors and *Notifier* component, it also provides the network information, for example, the data rate. The *Notifier* component provides the *notify* service that verifies the abnormal information and send it to the *Processor* component. The message channel *Channel 2* encapsulates the communication semantics between the *Notifier* and *Processor* components.

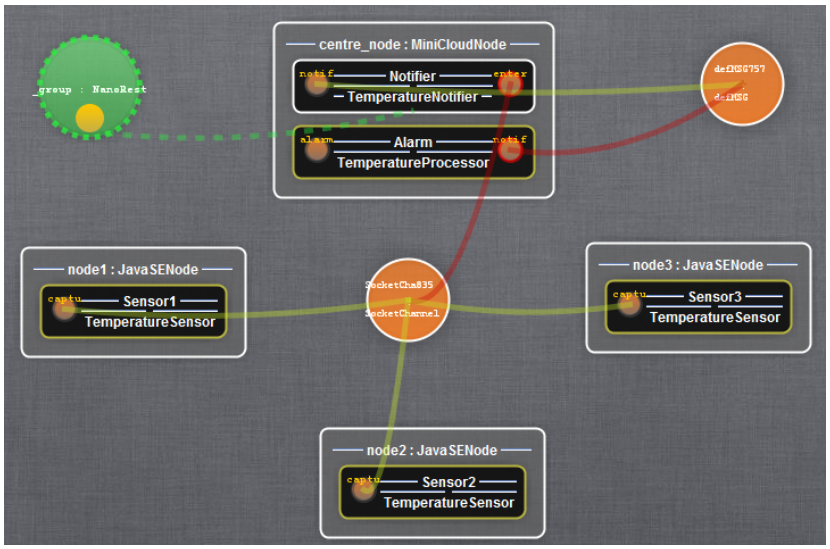

Figure 4.1 – Kevoree component model of the temperature's firefighters application.

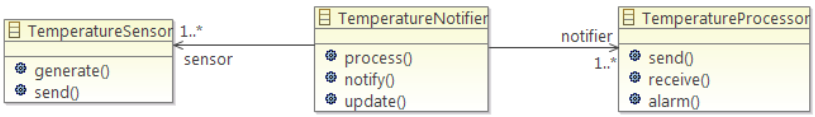

Figure 4.2 – The class diagram of the temperature's firefighters application.

4.2 Behavior models of individual components and of the whole system

Figure 4.3 defines the system-level behavioral model of the application, which is derived from the component model. In the following we will describe in more details the behavioral modeling of services of the application. Firstly, we will represent the description of the behavior model of the alarm service of the alarm component to understand how we can use the template in modeling. Then we will represent the behavior model of the *notify* service of

the *Notifier* component, and then the workload modeling. For clarity's sake, from now on we represent the modeling of timed behavioral semantics by means of two different languages, the first one conforming to our meta-model and constructed by developers, and the second one conforming to the CPN language, generated from the component model.

Before we go into details, let us take a look again at the convention in naming the declaration during modeling the behavior models of services. As mentioned above, the main coloured type is the MESSAGE product color type:

```
colset MESSAGE = product ADDRESS * DATA * ADDRESS timed;
```

The ADDRESS color type is used to describe the source and the target of the token. The DATA color is defined by means of record color type, provided by developers in their model, for example, the DATA color defined in sensor components in this case study :

```
colset DATA = record externalT:INT * internalT:INT * moved:BOOL * ID timed;
```

The first field in DATA record, the *externalT* is the external temperature captured by the sensor, the *internalT* field is the internal temperature of the firefighter, and the *moved* field is of boolean type, true if there was a move. The ID field provides a unique identity for the token. In any case, the *start place* and *end place* of behavioral models are of main type MESSAGE.

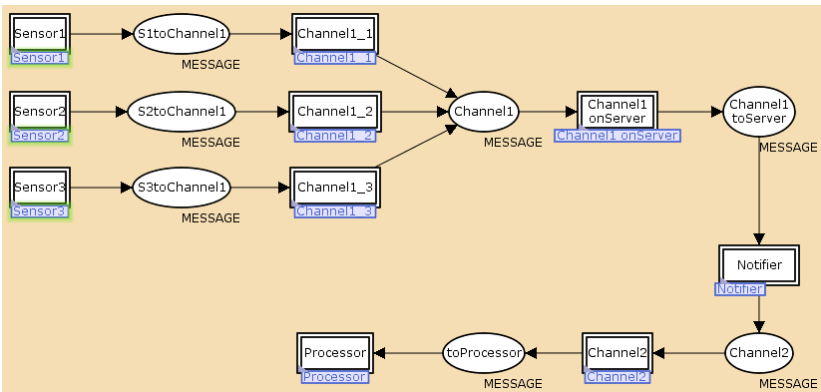

Figure 4.3 – The CPN model at top-level generated by Kermeta and derived from the Kevoree component model of the system.

Behavior model of the *alarm* service port

As shown in Figure 4.4, the behavior model of the alarm service port makes use of the *synchronous service* template to model the synchronous call to the *notify* service. The lower part in Figure 4.4 shows the properties and parameters of the template. It resembles a class import in Java, and declares and instantiates a java class. The *base* property of the *synchronous* template indicates the path of the parameterized template CPN associated with the Kermeta transformation code segment, which will be used to translate the template instance to the CPN language. The *interfaces* property determines the interface nodes of the template instance. In this example, the *Synchronous* pattern instance has two interface places named *In* and *Out*. The *In* place herein is also the *Start* place of the *alarm* service.

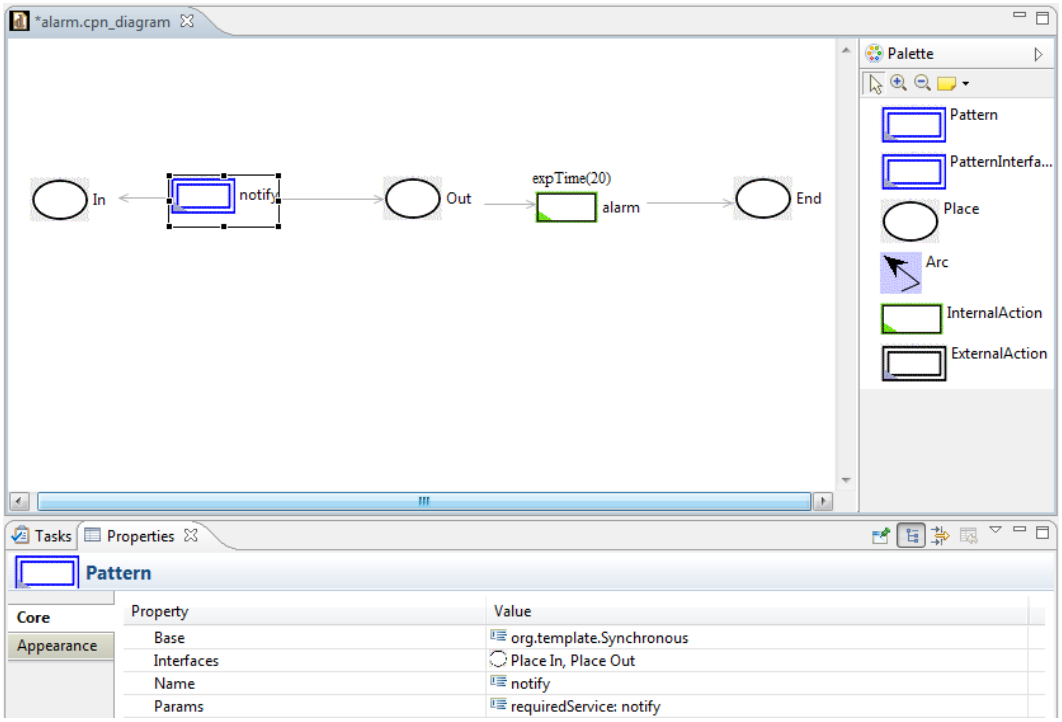

Figure 4.4 – The timed behavior model of the alarm *service of the alarm component, developed by third parties.*

The *name* property defines the name of the called service. Here, the *alarm* service calls the *notify* service to get the recent abnormal information received from the sensors. Hence, the *name* property is assigned to *notify*, which is the name of the called service. Finally, the params properties list the values of the parameters of the template. For instance, in this example, there is a parameter named *requiredService*, which is ñotify:

Figure 4.5 shows the CPN of the behavior model of the *alarm* service. This CPN of the *alarm* service is derived from the behavior model in Figure 4.4, by the model transformation.

Timed Behavior of the *broacast channel*

Kevoree does not allow the direct binding between ports. Instead, the channel instances carry the binding semantics. Channel instances are developed by fragments, one fragment on each node that connects to the channel using the FIFO semantic. This means that one *ChannelFragment* communicates with local ports, and other *ChannelFragment* of remote ports. In Figure 4.3 we have seen that 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. In the Kevoree framework, the channel instances are responsible for binding ports. The channel instances are deployed by fragments on each node that has connection with the channel. The channel fragments communicate with local ports and remote ports. The *Channel 1* is a broadcast channel, therefore, as we can see in Figure 4.6, the CPN model of the *Channel 1* instance on the *Node 1* is derived from the component model using the *Broadcast Channel Pattern* that was mentioned in Chapter 3. The behavior model of the *Channel 1* instances are generated automatically at runtime from

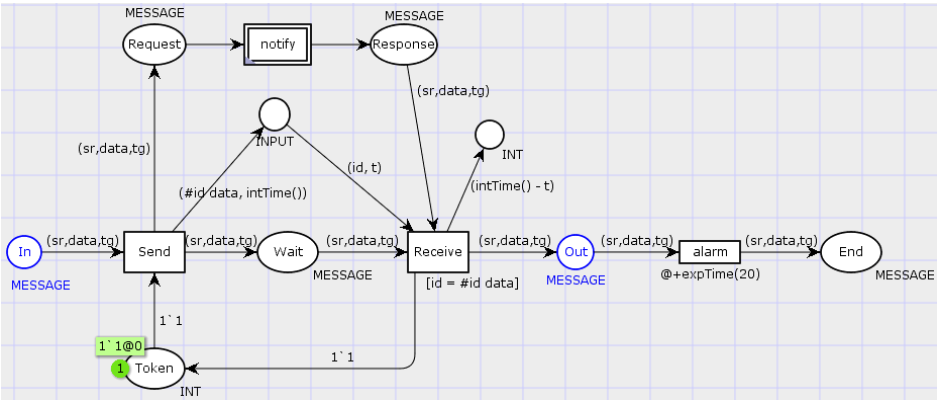

Figure 4.5 – The CPN model of the alarm *service derived after the transformation step.*

the component model using the *Broadcast Channel Pattern*. The architect provides the time delay of the channel only, in this example, the time delay is provided as a distribution function *expTime(20)*. The translator will generate the behavior models of the channel instances. Figure 4.6 shows an instance of the *Channel 1* on the emphNode 1, other channel instances models on emphNode 2, emphNode 3 should be generated in the same way. We can see that this instance of the *Channel 1* has only one target node, which is the *centre_node* place because the channel instance on *Node 1* has only one required port on the server that connects to the channel. The transmission delay between *Node 1* and *centre_node* is expressed by the distribution function: *expTime(20)* marked on the *send* transition.

Behavior model of the *notify* service port

We now represent the behavior model of the *notify* port. Figure 4.7 is the CPN model of the *notify* port, derived from the component model. This behavior model has a start place and an end place, which are *Channel1toServer* and *Channel2* places in Figure 4.3, respectively. Let us consider in more detail the semantic of this CPN model. Note that the *notify* service is responsible for verifying the abnormal information and sending this abnormal information to the *Alarm* component by calling the *alarm* service. Firstly, we should look again the declarations of this model:

```
colset MESSAGE = product ADDRESS * DATA * ADDRESS timed;

colset COUNT= product ADDRESS * INT;

colset COUNTS = list COUNT;

colset DATA = record externalT:INT * internalT:INT * moved:BOOL * id : INT timed;

var sr, tg : ADDRESS;

var i : INT;
```

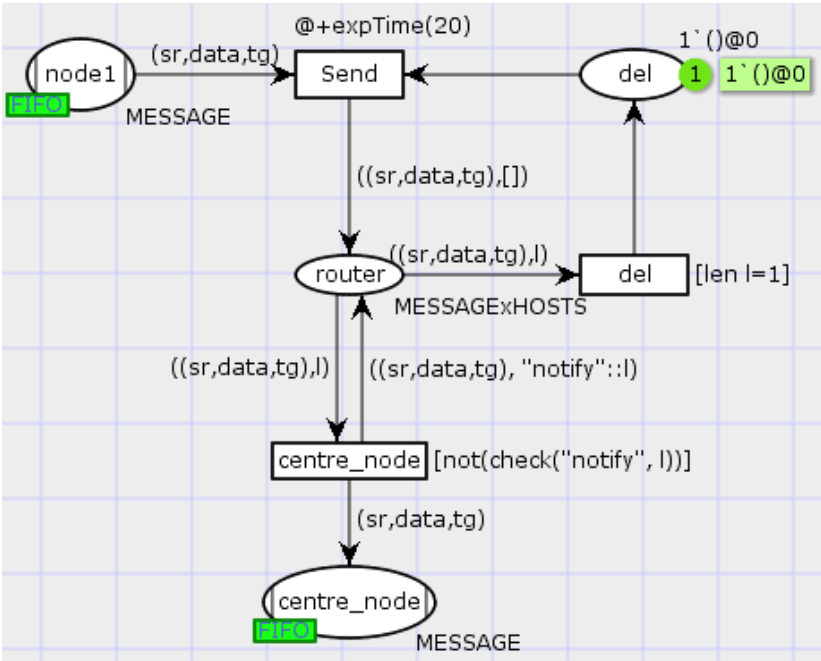

Figure 4.6 – The CPN model after the transformation of the Channel 1 *instance implemented in node 1.*

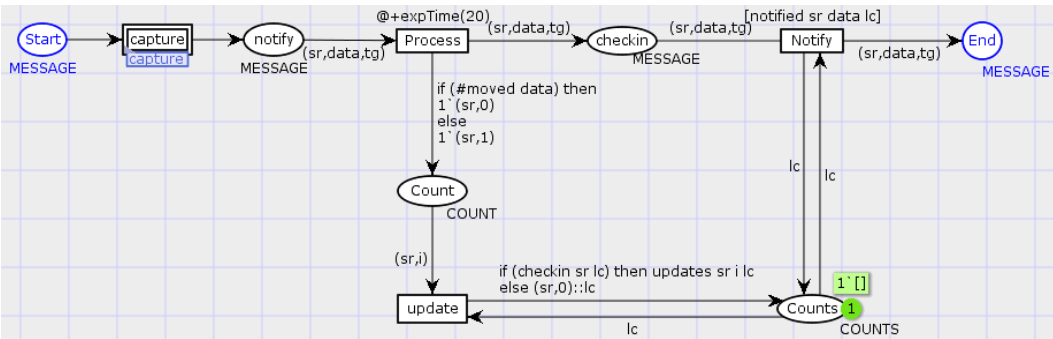

Figure 4.7 – The CPN model of the notify *service port.*

```
var lc : COUNTS;

var data : DATA;
```

The MESSAGE and DATA color sets are the same as the ones defined in previous elements that we have presented above. *sr, tg* are the variables of ADDRESS type, *data* is a variable of DATA type. The COUNT and COUNTS are counting the number of firefighter movements. The following will explain in more detail this CPN net. First, the *notify* service receives messages in *Channel1toServer* place, and then the *process* transition transfers the message to the *checkin* place to be than checked and determine the message whether the value is abnormal or not. The address information of the source and the movement *(sr, #moved)*

contained in the message is sent to the *COUNT* place, then the number of movement is updated and this information is saved in *COUNTS* place. The *checkin*function checks if the source token (or the sensor address) is already in the *COUNTS* place:

```
fun checkin ad nil = false |
checkin ad ((a,b)::l) = if ad=a
then true else checkin ad l;
```

Then, if the source is already in the *COUNTS* place, the function *updates* below will update the movement number corresponding to this source, otherwise the movement number of this source will be initialized to 0.

```
fun updates ad c ((a,b)::l) = if ad = a
then (a,b*c+1)::l else (a,b)::updates ad c l;}
```

Finally, the transition *notify* will be fired if the condition of abnormal messages is fulfilled. More precisely, if one of the conditions hereafter is fulfilled: the external temperature is greater than 100, or the external temperature is greater than 70 and the firefighter has not moved during the last 20 seconds, or the internal temperature is greater than 40 and the firefighter has not moved during the last 20 seconds. The *notified* function is responsible for verifying abnormal messages, this function used as the condition mark of the *notify* transition. The code snippet below represents this function:

```
fun notified sr (data:DATA) l =
if (noMove sr l)
    andalso ((#internalT data)  > = 40) then true
    else if (noMove sr l)
    andalso ((#externalT data) > = 70) then true
    else if ((#externalT data) > = 100) then true
    else false;
```

The *noMove* function checks whether or not there is a sensor that has not moved for more than 20 seconds in *COUNTS* place.

```
fun noMove  source nil = false |
noMove source ((a,b)::l) = if (source=a)
andalso (b >= 20) then true else noMove source l;
```

When the *notify* transition fires, it sends the abnormal messages to *channel 2* and then it begins to transfer to the *alarm* service port. This *alarm* service will then react to the abnormal messages received.

Behavior model of the *capture* service port

Figure 4.8 describes the timed behavior model of the *capture* service port of the sensor component, the behavioral models of other sensors are modeled similarly. In this model, the developer models the arbitrarily generation of the temperature by means of an exponential distribution that is defined as a ML function:

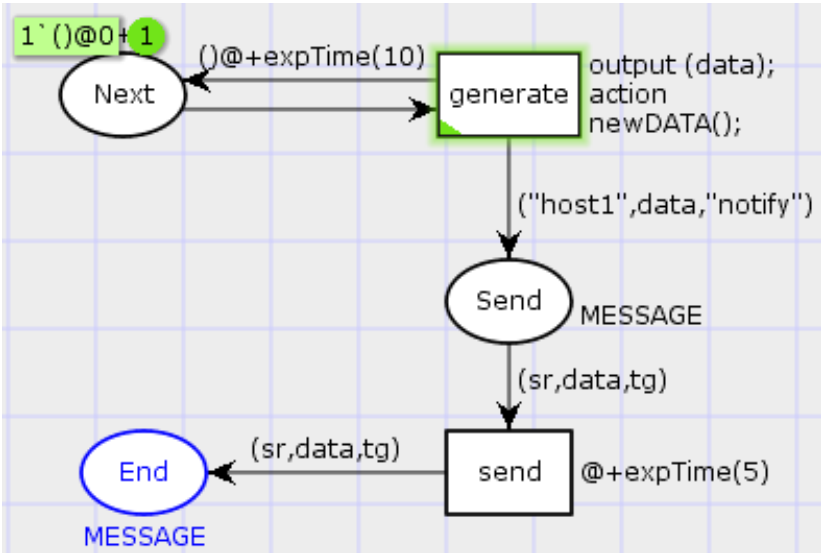

Figure 4.8 – The timed behavior model of the capture *service port.*

```
fun expTime (mean: int) =
let
    val realMean = Real.fromInt mean
    val rv = exponential((1.0/realMean))
in
    floor (rv+0.5)
end;}
```

The left hand side of Figure 4.8, comprising the generate transition and the Next and Send places, models the arbitrarily generation of the temperature. The mean value, or the rate of the exponential distribution function is described as expTime(10), which is the period to capture the temperatures and movement of the firefighter and then sent to the server. The second parameter is the name of the host in which the sensor component is implemented, this parameter will be adjusted in the transformation step, the host' s name is derived from the component name in the component model. Finally, the developer has also to describe the function named *newData* to generate data values. The function below is an example, the distribution of the internal temperature, external temperature, and the movement being 37, 70 and hasMove(2), respectively.

```
fun newDATA() = {externalT = expTime(40) ,
internalT = expTime(70), moved = hasMove(expTime(5)), id = ran()}}
```

thus function

```
hasMove(s : INT, r : Ten) = (s > = r)

 colset Ten : int with 1..10}
```

Figure 4.8 is the CPN model of the *capture* service that is generated from the component model, the mean time between two continuous capturing is 10 and this time obeys exponential distribution *expTime(10)*. The name of the node that carries this sensor is *host1*. *sr, tg* are variables of type ADDRESS, *data* is variable of type DATA.

4.3 Measuring performance aspects

4.3.1 Notification delay time monitor

As mentioned earlier, for the purpose of measuring performance aspects of the behavior model, developers should define monitors to collect data from simulation, so that the statistic results can be calculated. In the previous section, we have mentioned that there are different kinds of monitors, but the *User defined* monitor is the most convenient, because it fits our need for a mechanism to communicate with external processes to allow for the computation of statistic results by the external reasoner. The following describes the monitor that measures the delay from the moment that sensors send messages until the moment that the *notify* transition fires to transfer the abnormal messages. As explained earlier, a *User defined* monitor has different parts:

Type User defined

Nodes ordered by pages This part should be generated by model transformation from the component model. In the firefighter example, the pages of the monitor are only: *Notifier* page, which is *notify* service, and the transition related to this page is *notify* transition. The developer only has to indicate the transitions and places that relate to the monitor in their behavior model.

Init the Init function is instantiated before the simulation start. In the firefighter example, we want to check if the connection between the simulation and the external process for this monitor is already established.

```
fun init () =
    if !connected = true then
     (ConnManagementLayer.closeConnection("Conn");
      connected := false)
    else ()
```

Predicate This function is the precondition of the monitor. In this example, the predicate function for the monitor is invoked each time the transition *notify* occurs, it returns true if the transition occurs.

```
fun pred (bindelem) =
let
  fun predBindElem (Notifier'Notify (1, {data,lc,sr,tg})) =
```

```
    true
    | predBindElem _ = false
  in
    predBindElem bindelem
  end
```

Observer This function is invoked every time the Predicate function returns true; in the firefighter example, the Observer function is invoked each time that transition *notify* occurs.

```
fun obs (bindelem) =
let
fun obsBindElem (Notifier'Notify (1, {data,lc,sr,tg})) =
INT.mkstr((intTime() - (#AT data)))
| obsBindElem _ = ""
in
  obsBindElem bindelem
end}
```

In this function, note that we want to count the time elapsed between capturing of abnormal messages by sensors and notifying abnormal messages by *Notifier*. The function *intTime()* returns the actual simulation time, and the field AT of the DATA record is the time of capturing information.

Action This Action function is invoked every time the Observer function returns a value. In the firefighter example, the Action function is responsible for transferring the value returned by Observer function to the external process through the Connection *Conn* and the port 9000.

```
fun action (s1) =
(if not(!connected)
then (ConnManagementLayer.acceptConnection("Conn",9000);
          connected:=true)
else ();
send_to_monitor(s1))
```

In the code snippet above, *s1* is the data returned by Observer function. The function send_to_monitor(s1) transfers s1 to the external process.

```
fun send_to_monitor(text) =
ConnManagementLayer.send("Conn",text,stringEncode);
```

4.3.2 Conclusion

For the configuration described above, we have launched a simulation of 10000 steps to measure two performance aspects. The first one is the amount of time elapsed between the apparition of abnormal information and the notification of these abnormal information, this measurement is executed by the notification delay time monitor presented in previous section. And the second one is the queue delay on the server side, it measures the amount of time that messages have to wait in the queue of the server (or centre_node).

Name	Count	Avrg	Min	Max
Queue_Delay	712	124.991573	1	386
Notification_Time	33	130.818182	10	352

The launching of simulations executed with help of the Access/CPN tool [108], and the Comms/CPN library makes it possible for CPN Tools to communicate based on TCP/IP with external application and processes [42]. Our tools take Access/CPN as dependencies to realize the simulations and collect the statistical results from the simulations through socket communication. However, when launching simulations, we should pay attention to the confidence intervals of the results, furthermore the independent and identically distributed estimates of performance measures must be collected from independent and terminating simulations. CPN Tools provides support for performance analysis using *Simulation replications* of independent and terminating simulations. Therefore, we can consider the simulation replications configuration in case that the accuracy of the confidence interval of the average values measured could be considered as an importance aspect. We have used the our process to design and predict the different performance aspects of the Kevoree component model. However, our current toolchain lacks the optimization algorithms to select the best configuration, based on the performance measurements.

As mentioned above, in order to validate the presented Kevoree extension framework and the process development, we have performed design and analysis case study on a firefighter application. Besides the validation of the performance prediction technique, we have additionally applied the patterns in component behavior modeling and the aspect-oriented modeling for defining the measurement models. The case study proved the suitability of our process 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 plugin), a set of generated monitors created from the patterns instantiated in the Kevoree model and Kevoree wrapper components that interface with the models et runtime engine. Running the tool chain on a Kevoree model of dozen 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 [108]. Running the tool chain 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 performs this kind of models at runtime computation.

We also consider the effort that is involved performing the architecture analysis, we have measured this effort calculating the time periods for the architecture analysis activity. The

architecture analysis process consists of component implementation, component behavior modeling, and simulation. In the case study, we have used the Broadcast channel pattern to model the communication semantic of the channels. In addition, we have applied the measurement models to define the monitors for measuring the performance aspects in the application, such aspects are the notification time and the queueing delay. The usage of the library of patterns and measurement models lead time efforts reduction in modeling behavior models.

Chapter 5

Conclusion

Performance is often a central and challenging issue in design, development and configuration. In the context of soft real time systems, time-related performance properties are vital and most challenging properties to predict, enforce and measure. In this thesis we focused on the adaptive component-based distributed systems that must support major architectural changes at run-time, without stopping but instead by hot-deploying. An example of such kind of system is a car-to-car scenario, two cars approaching the same intersection should be able to synchronize in a reasonably short delay to share information about their own context and configuration, then take distributed decisions, e.g. on the precedence order to cross the intersection. In such kind of systems, performance failures may cause serious consequences. In order to avoid these failures, the performance properties should be predicted and analyzed at early design phases and at the run-time reconfiguration to making a choice between alternative configurations. In component-based systems, the functional and extra-functional properties of individual components are used to reason about those functional and extra-functional properties of component composition. In fact, a solution for performance-centric predictable assembly of adaptive distributed systems should satisfy the following characteristics. First, it enables an automated synthesis of system performance properties from the related component properties; second, it requires the separation of concerns in modeling of functional and extra-functional properties of the components and the system; third, it provides a time-effect performance analysis.

This thesis presents a framework that features three aforementioned characteristics. The framework defines a development and validation process which guides a component developer and a QoS analyst through an iterative design cycle, while focusing on performance properties. The iterative cycle contains the following phases. Firstly, construction of a number of alternative component models from the available libraries of Kevoree components. Secondly, for each alternative, assembling the behavior models of these individual component models into a system-wide performance-related model. Thirdly, analyzing these performance models with respect to the requirements. The latter phase helps the reasoning engine in selecting a

performance optimization of the promising alternative. Fourthly and finally, the adaptation mechanism of the system selects the best architecture alternative for implementation.

The Kevoree framework is dedicated to design based on models at runtime that enables dynamic adaptation for distributed component-based systems. At runtime, the system's environment fluctuation provided by context sensors, and may trigger a system configuration. The adaptation rules part or reasoning model describes selection of the variability features according to context. The existing Kevoree infrastructure investigates two different formalism to capture the adaptation rules, which are event-condition-action (ECA) rules and goal-based optimization rules. The work in this thesis focuses on a technique based on the continuous performance-related validation that supports the reasoning engine to select a performance optimization architecture. The behavior models of individual components are specified at the component-development time and are shipped in a component library. At design time, the models of the constituent components are automatically synthesized into an executable system model. Designers rely on quantitative analysis techniques to validate implementation 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 at run time. Simulation of the system architectural model provides performance measures.

From a development and validation process point of view, the framework features the following principal benefits.

- At design time, an architect needs to obtain only the corresponding behavior models of the components, design system-wide performance-related models and conduct the performance analysis. The design and analysis can be performed prior to the purchase or implementation of the components. At runtime, multiple system models that are generated by the reasoning engine could be analyzed to select the best architecture model. There is no system implementation needed within the iteration cycle.

- The performance analysis can be carried out for systems constructed from a set of third-party components. Multiple architecture models generated by the reasoning engine can be analyzed for performance in a reasonable time to select the best configuration. An assumption is that each individual component has to be accompanied with predefined behavior models specifying the internal properties of that component. Synthesis of component models allows to characterize system behavior in a consistent and automated way.

- Rapid component prototyping with help of a library of design pattern. Instead of using pure CPN language for the development of component behavior model, the parameterized templates and aspect-oriented modeling facilitate the construction of performance-related behavior models.

- The method provides a powerful language that based on Colored Petri Net for performance modeling of complex systems.

However, the proposed approach has a number of limitations. The method requires that designers have some experience in modeling performance-related component behavior with colored Petri nets. Developers must follow the convention (as mentioned in Section 3.2) in modeling of interface places. Otherwise this could affect the capacity of the modeling language. The automatic synthesis of individual CPN models of third-party components implicit the incorrectness of the composed system-wide model. The method assumes that the workload models are available for every simulation.

In Chapter 3, we have proposed a library of design patterns that provides parameterized templates to facilitate the modeling of distributed systems, process-aware information systems, communication protocols, etc. Due to the space limitation, we only proposed some commonly used patterns in Kevoree systems, such as broadcast channel template, synchronous service call template. The application of the aspect-oriented modeling approach augmented the re-usability and separation of concerns in modeling with our proposed CPN language for Kevoree. In our opinion, the proposed CPN language can be applied to a broad set of systems for performance validation and expression.

In order to validate the presented Kevoree framework and the continuous validation method, we have performed a case study on a firefighter application. The case study contains all development and validation process, starting from components development phase and ending with the profiling of the selected system model on the actual infrastructure.

5.1 Discussion on Research Questions

This section aims at reviewing the four research questions that we selected in Chapter 1 of this thesis. The main goal of this is to describe how we address these research questions.

RQ1: How should the functional and performance properties of individual independent-developed components be specified in order to enable automated composition of these properties and to capture all environment aspects that may influence the performance of the components?

The behavioral properties of components can be expressed by behavior models, and the performance properties can be specified directly by using timed transitions or a resource component model. Instead of specifying the processor in terms of the number of instruction cycles that an operation requires from a processor to be executed, in the scope of this thesis the number of instruction cycles are mapped directly into the execution time metric. Time stamps can be specified by probability distributions. The CPN language supports several types of probability distribution. The component resource model that specifies other software and hardware claims (e.g. thread pool, semaphore) for each operation provided by a component can be expressed by means of an aspect model (as mentioned in Section 3.3). The behavior component models introduced (as mentioned in Section 3.2) describe a component behavior by specifying actions that the provided operations of the component are performing upon their invocation. These actions specify the sequence of *ExternalCall* operations called by interfaces of neighboring components. At design time these neighboring components are not known. However, a component developer does not need to have this information, because it is sufficient to know the signatures of provided interfaces. At the time of creating a component

assembly, when the component model is known, the behavior and performance properties of individual components can be combined into system-wide properties.

RQ2: How to evaluate performance properties of combined system architectures at runtime in an automatic way?

At runtime, the environment context may trigger a system reconfiguration. The reasoning engine generates a set of configurations based on adaptation rules. The results lead to a list of synthesized component models, with detailed behavioral and performance characteristics of individual components. The architectural component models and behavior models of involved components are input for the synthesis of the performance system-wide models. The component model contains the specification of the composition of the service instances, the involved component instances, the channel instances, and behavior models of involved components. The model transformation done using Kermeta [55] tool in Chapter 3 results in the constructed performance system model, which can be simulated to obtain the predictions of various performance properties of the system.

RQ3: How can the reasoning engine compare architectural alternatives and select an optimized with respect to multiple quality attributes?

To construct a system that fulfills all its performance-related requirements is a challenging task. As a consequence, the reasoning engine has to consider several generated architectural alternatives and identify a solution that satisfies most quality objectives, and where the optimal balance between different quality attributes is achieved. The reasoning algorithms for this optimization problem is out of scope of this thesis, we consider this issue as a future work. The selection of an optimized configuration may contain different steps, such as generation of a set of appropriate architectural alternatives, simulation to obtain multiple of performance properties values of these alternatives, a trade-off analysis of the alternatives with respect to multiple performance properties (e.g. response time, resource usage), and finally, identification of an optimal alternative to be instantiated in nodes of the system.

RQ4: Can this approach proposed in this thesis be applied to other systems, such as embedded real-time systems?

The Colored Petri Nets used in the framework is a discrete-event modeling language that extends classical Petri nets by allowing definitions of actions with the Standard ML functional programming language. By combining classical Petri nets and Standard ML language, the CPN language has proved its powerful capacity of modeling. Therefore, the CPN language in general and the refined CPN language of our framework are suitable for modeling most of the complex systems, such as distributed systems, or embedded systems. For instance, to model and evaluate performance properties of embedded systems, we address behavioral properties and the performance properties of passive and active components. These aspects can be modeled by the behavior metamodel and the aspect modeling proposed in this thesis.

RQ5: How can other extra-functional properties like security, availability, etc be expressed and evaluated?

The other extra-functional concerns such as security, availability can be expressed by using the aspect modeling proposed in this thesis. Each extra-functional concern can be constructed by an aspect model and then be weaved into the behavior component model by defining the joint points in the behavior component model. The aspect modeling allows the separation of concerns and re-usability in extra-properties-related behavior modeling.Moreover, there are

several advantages of using a formal method like coloured Petri nets in system design such as simulation, verification of properties. We have shown the approach of validation, performance analysis to support the reconfiguration of the reasoning mechanism. However, performance evaluation based on simulation can not used to prove correctness of structure composition. It could be useful to apply formal model checking techniques to verify all relevant properties of components and systems. A promising approach would be to use a branching-time logic ASK-CTL [28], which is integrated into CPNtools, to evaluate other system properties such as safety, liveness and precedence properties. It is as important as performance evaluation to verify these system properties. The solution proposed here is to use a state-space formal model checking technique to verify if the generated CP-net satisfy the properties given as ASK-CTL statements. Figure 5.1 gives an overview of the state-space formal model checking process to verify the generated CP-net. The ASK-CTL toolkit is provided with CPNtools to analyze state spaces by means of a CTL-like temporal logic. ASK-CTL allows to formulate queries about states, and queries about state changes (e.g., the occurrence of certain transitions).

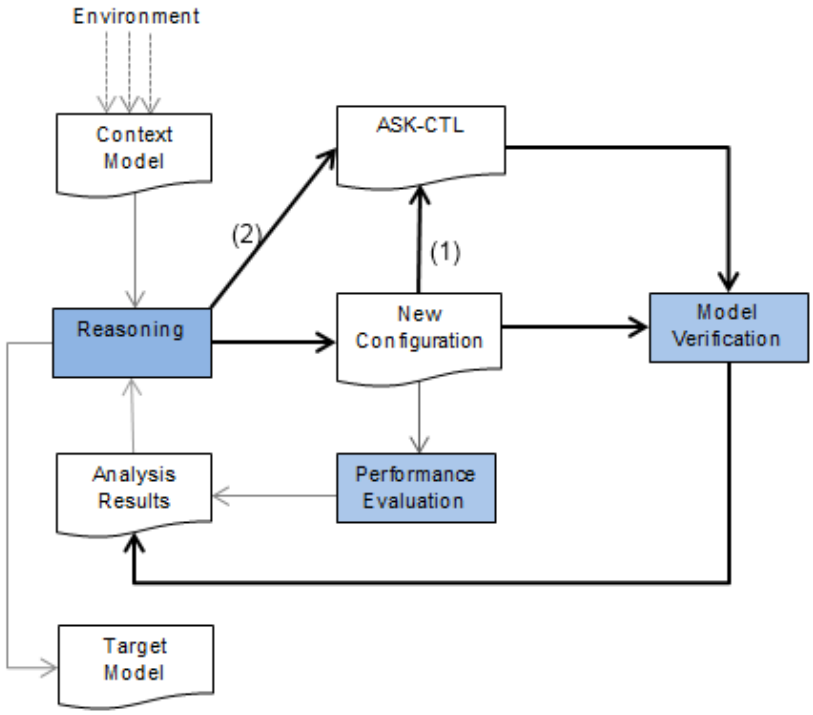

Figure 5.1 – The state-space formal model checking to verify the generated configuration

5.2 Open Issues and Future Work

At present, the reasoning engine of the Kevoree framework has not yet investigated a performance-based goal algorithm to resolve the optimization problem that we mentioned above in the research question #3. In order to provide the Kevoree framework with reason-

ing algorithms that allow resolving the optimization problem, we aim at introducing extra-properties goal-based adaptation rules part of the reasoning model. The method should also provide a trade-off analysis of the alternatives with respect to multiple performance properties (e.g. response time, resource usage). Pareto-based trade-off analysis [72] is a promising approach to be considered.

In addition to the performance properties expression and evaluation, the analysis of other extra-properties would be useful to select the best configuration. The expression of the other extra-functional properties ca be achieved by using aspect modeling. Each extra-functional concern can be modeled by an aspect model, and then these models can be weaved into the base model. Moreover, as mentioned above in the research question #5, the branching-time logic ASK-CTL [28] is integrated into CPNtools to evaluate other system properties such as, safety, liveness and precedence properties. However, as an obvious drawback, the modeling of aspect models would require substantial effort for modeling by means of CPN language.

In addition, we focus on the enhancements of modeling expression facilities. Instead of using directly CPN language, it would be useful to use design-oriented models, the translation from these design-oriented models to analysis-oriented model (our CPN language) with help of the model transformation methodology. The point worth noting here is the usage of a specific design metamodel for each domain. For instance, the MARTE profile [70] can be used to model components of embedded systems. Model driven approaches and technologies can help to connect the design world and the analysis world by building transformation rules between them.

Finally, it would be interesting to take into account the process mining approach [34]. Process mining aims at automatically generating process models from event logs. The real-life event logs captured at run time by a Kevoree system can be used to form an executable model that reflects the real system. The model discovered can be used as feedback mechanism to check if the prescribed models (the behavior model of the system) fit the executed ones. This feedback mechanism allows to refine and improve the behavior models of involved components in the system. Another objective is to automatically construct the behavior models for new developed components.

Bibliography

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

[2] Jan Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[3] H.O. Almeida, L.D. da Silva, E. Oliveira, and A. Perkusich. A formal approach for component based embedded software modelling and analysis. In *Proceedings of the IEEE International Symposium on Industrial Electronics,*, volume 4, pages 1337–1342, 2005.

[4] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *LICS '90, Proceedings., Fifth Annual IEEE Symposium on Logic in Computer Science.*, pages 414 –425, jun 1990.

[5] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Transactions on aspect-oriented software development i. chapter An Overview of Caesarj, pages 135–173. Springer-Verlag, Berlin, Heidelberg, 2006.

[6] André Arnold. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[7] André Arnold. Nivat's processes and their synchronization. *Theor. Comput. Sci.*, 281(1-2):31–36, June 2002.

[8] Felix Bachmann, Len Bass, Charles Buhman, Santiago C. Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical report, Software Engineering Institute Carnegie Mellon, 2000.

[9] Gianfranco Balbo. Introduction to generalized stochastic petri nets. In *Proceedings of the 7th International Conference on Formal methods for Performance Evaluation*, SFM'07, pages 83–131, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. Mocas: A state-based component model for self-adaptation. In *Proceedings of the 2009 Third IEEE International*

Conference on Self-Adaptive and Self-Organizing Systems, SASO '09, pages 206–215, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Elisa Baniassad and Siobhan Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering*, pages 158–167. IEEE Computer Society, 2004.

[12] Steffen Becker. The palladio component model. In Alan Adamson, Andre B. Bondi, Carlos Juiz, and Mark S. Squillante, editors, *WOSP/SIPEW*, pages 257–258. ACM, 2010.

[13] Steffen Becker, Wilhelm Hasselbring, Alexandra Paul, Marko Boskovic, Heiko Koziolek, Jan Ploski, Abhishek Dhama, Henrik Lipskoch, Matthias Rohr, Daniel Winteler, Simon Giesecke, Roland Meyer, Mani Swaminathan, Jens Happe, Margarete Muhle, and Timo Warns. Trustworthy software systems: A discussion of basic concepts and terminology. *SIGSOFT Softw. Eng. Notes*, 31(6):1–18, November 2006.

[14] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, WOSP '07, pages 54–65, New York, NY, USA, 2007. ACM.

[15] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, January 2009.

[16] Simona Bernardi and Dorina C. Petriu. Comparing two uml profiles for non-functional requirement annotations: the he spt and qos profiles. In *THE SPT AND QOS PROFILES, UML'2004*, 2004.

[17] Antonia Bertolino and Raffaela Mirandola. Cb-spe tool: Putting component-based performance engineering into practice. In *PROC. 7TH INTERNATIONAL SYMPOSIUM ON COMPONENT-BASED SOFTWARE ENGINEERING (CBSE 2004*, pages 233–248. Springer, 2004.

[18] Antonia Bertolino and Raffaela Mirandola. Software performance engineering of component-based systems. In *Proceedings of the 4th International Workshop On Software And Performance*, WOSP '04, pages 238–242, New York, NY, USA, 2004. ACM.

[19] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[20] Preeti Bhoj, Sharad Singhal, and Sailesh Chutani. Sla management in federated environments. In *Computer Networks*, pages 293–308. IEEE Publishing, 1999.

[21] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@runtime. *Computer*, 42(10):22–27, 2009.

[22] Barry W Boehm, John R Brown, Hans Kaspar, and Myron Lipow. *Characteristics of software quality.* TRW Softw. Technol. North-Holland, Amsterdam, 1978.

[23] Egor Bondarev, Michel R. V. Chaudron, and Erwin A. de Kock. Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, pages 153–163, New York, NY, USA, 2007. ACM.

[24] Egor Bondarev, Peter de With, Michel Chaudron, and Johan Muskens. Modelling of input-parameter dependency for performance predictions of component-based embedded systems. In *IN PROCEEDINGS OF THE 31TH EUROMICRO CONFERENCE*. Society Press, 2005.

[25] Pierre Bremaud. *Markov Chains : Gibbs Fields, Monte Carlo Simulation And Queues*. Springer, New York, USA, 1999. ISBN: 0-387-98509-3.

[26] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of component-based applications. *Journal of Systems and Software*, 74(1):35–43, 2005.

[27] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[28] S. Christensen and T.B. Haagh. *Design/CPN Overview of CPN ML Syntax*. University of Aarhus, 1996.

[29] Erhan Cinlar. *Introduction to Stochastic Processes*. Prentice-Hall, Englewood Cliffs, NJ, [nachdr.] edition, 1975.

[30] Ivica Crnkovic. Component-based approach for embedded systems. In *In Proceedings of 9th International Workshop on Component-Oriented Programming*, 2004.

[31] Luiz Marcio Cysneiros and Julio César Sampaio do Prado Leite. Using uml to reflect non-functional requirements. In *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '01, pages 2–. IBM Press, 2001.

[32] Erwan Daubert, François Fouquet, Olivier Barais, Grégory Nain, Gerson Sunye, Jean-Marc Jézéquel, Jean-Louis Pazat, and Brice Morin. A models@runtime framework for designing and managing service-based applications. In *Software Services and Systems Research - Results and Challenges (S-Cube), 2012 Workshop on European*, pages 10–11, 2012.

[33] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006.

[34] A. K. Alves De Medeiros and Christian W. Günther. Process mining: Using cpn tools to create test logs for mining algorithms. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 177–190, 2005.

[35] Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning workflow petri nets. *Fundam. Inform.*, 113:205–228, 2011.

[36] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Models in software engineering. chapter Modeling and Validating Dynamic Adaptation, pages 97–108. Springer-Verlag, Berlin, Heidelberg, 2009.

[37] Gerard Florin and Stéphane Natkin. Generalization of queueing network product form solutions to stochastic petri nets. *IEEE Trans. Softw. Eng.*, 17(2):99–107, February 1991.

[38] International Organization for Standardization ISO. Iso 9126-1, software engineering - product quality, part 1: Quality model. Technical report, 2001.

[39] Francois Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 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.

[40] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[41] Svend Frølund and Jari Koistinen. Quality of services specification in distributed object systems design. In *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*, COOTS'98, pages 1–1, Berkeley, CA, USA, 1998. USENIX Association.

[42] Guy Gallasch and Lars M. Kristensen. Comms/CPN: A communication infrastructure for external communication with design/CPN. In *3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01) / Kurt Jensen (Ed.)*, pages 75–90. DAIMI PB-554, Aarhus University, August 2001.

[43] Kurt Geihs, Mohammad Ullah Khan, Roland Reichle, Arnor Solberg, Svein Hallsteinsen, and Simon Merral. Modeling of component-based adaptive distributed applications. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 718–722, New York, NY, USA, 2006. ACM.

[44] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. Klaper : An intermediate language for model-driven predictive analysis of performance and reliability. *The Common Component Modeling Example*, pages 327–356, 2008.

[45] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. From design to analysis models: a kernel language for performance and reliability analysis of component-based

systems. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 25–36, New York, NY, USA, 2005. ACM.

[46] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems, 2007.

[47] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu. An xml-based quality of service enabling language for the web. *Journal of Visual Language and Computing, Special Issue on Multimedia Language for the Web*, 13:61–95, 2001.

[48] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[49] Joseph Y. Halpern and Moshe Y. Vardi. Model checking vs. theorem proving: A manifesto, 1991.

[50] J. Happe, H. Koziolek, and R. Reussner. Facilitating performance predictions using software components. *Software, IEEE*, 28(3):27–33, 2011.

[51] George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[52] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging predictable assembly. In *COMPONENT DEPLOYMENT, IFIP/ACM WORKING CONFERENCE, CD 2002*, pages 108–124. Springer-Verlag, 2002.

[53] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1 edition, April 1991.

[54] Kurt Jensen, Lars Michael Kristensen, and Lisa 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.

[55] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[56] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, June 2008.

[57] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1188–1195, New York, NY, USA, 2006. ACM.

[58] Er Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:2003, 2003.

[59] Jeffrey O. Kephart and Rajarshi Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, January 2007.

[60] Leonard Kleinrock. *Theory, Volume 1, Queueing Systems.* Wiley-Interscience, 1975.

[61] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *Software Engineering, IEEE Transactions on*, 32(7):486–502, 2006.

[62] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634 – 658, 2010. <ce:title>Special Issue on Software and Performance</ce:title>.

[63] Heiko Koziolek and Jens Happe. A qos driven development process model for component-based software systems. In *Proceedings of the 9th international conference on Component-Based Software Engineering*, CBSE'06, pages 336–343, Berlin, Heidelberg, 2006. Springer-Verlag.

[64] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[65] Magnus Larsson, Kurt Wallnau, Scott Hissam, John Hudak, James Ivers, Mark Klein, Gabriel Moreno, Linda Northrop, Daniel Plakosh, Judith Stafford, and William Wood. Predictable assembly of substation automation systems: An experiment report. Technical Report, Mälardalen University, September 2002.

[66] Ronan Mac Laverty. Robocop: Robust open component based software architecture for configurable devices project. Initial specification, ITEA PROJECT 00001 Deliverable, 2002.

[67] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[68] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[69] Thomas Mailund. Parameterised coloured petri nets, 1999.

[70] Frédéric Mallet and Robert De Simone. MARTE: A Profile for RT/E Systems Modeling, Analysis (and Simulation?). In *First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems SIMUTools'08*, pages 1–8, Marseille, France, 2008. ICST, ACM. The original publication is available from ACM Digital Library (http://portal.acm.org/citation.cfm?id=1416222.1416271).

[71] Markus Markus Muller-Olm, David Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In *In Proc. SAS*, 1999.

[72] C. A. Mattson and A. Messac. Pareto Frontier Based Concept Selection Under Uncertainty, With Visualization. *Optimization and Engineering*, 6(1):85–115, 2005.

[73] J McCall. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*, volume 1-3. General Electric, November 1977.

[74] P. Meier, S. Kounev, and H. Koziolek. Automated transformation of component-based software architecture models to queueing petri nets. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 339–348, 2011.

[75] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: Metrics, Models, And Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[76] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[77] Bertrand Meyer. *Eiffel: the Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[78] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[79] Mubarak Mohammad and Vasu Alagar. Tadl - an architecture description language for trustworthy component-based systems. In *Proceedings of the 2nd European conference on Software Architecture*, ECSA '08, pages 290–297, Berlin, Heidelberg, 2008. Springer-Verlag.

[80] M. K. Molloy. Performance analysis using stochastic petri nets. *IEEE Trans. Comput.*, 31(9):913–917, September 1982.

[81] Brice Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability*. PhD thesis, Université Rennes 1, Septembre 2010.

[82] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[83] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 782–796, Berlin, Heidelberg, 2008. Springer-Verlag.

[84] Adrian Mos and John Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *in: Proceedings of the Third International Workshop on Software and Performance (2002*, pages 235–236. ACM Press, 2002.

[85] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *in: International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005*, pages 264–278. Springer, 2005.

[86] Nataliya Mulyar and Wil M.P. van der Aalst. Towards a pattern language for colored petri nets, 2005.

[87] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989. NewsletterInfo: 33Published as Proceedings of the IEEE, volume 77, number 4.

[88] OMG. UML Profile for Schedulability, Performance and Time (SPT).

[89] OMG. *MOF QVT Final Adopted Specification.* Object Modeling Group, June 2005.

[90] OMG. Meta object facility (mof), 2010.

[91] Martin Ouimet and Kristina Lundqvist. Formal software verification: Model checking and theorem proving. Technical Report, Mälardalen University, March 2007.

[92] John D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[93] Fatemeh Rahimian, Thinh Le Nguyen Huu, and Sarunas Girdzijauskas. Locality-awareness in a peer-to-peer publish/subscribe network. In *Proceedings of the 12th IFIP WG 6.1 international conference on Distributed Applications and Interoperable Systems*, DAIS'12, pages 45–58, Berlin, Heidelberg, 2012. Springer-Verlag.

[94] E.E. Roubtsova and M. Aksit. Extension of petri nets by aspects to apply the model driven architecture approach. In *Preliminary Proceedings of the 1st International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB)*, 2005.

[95] Guadalupe Salazar-Zárate, Pere Botella, and Ajantha Dahanayake. Uml and the unified process. chapter Introducing Non-Functional Requirements in Uml, pages 116–128. IGI Publishing, Hershey, PA, USA, 2003.

[96] Sébastien Saudrais, Noël Plouzeau, and Olivier Barais. Integration of time issues into component-based applications. In *Proceedings of the 10th international conference on Component-based software engineering*, CBSE'07, pages 173–188, Berlin, Heidelberg, 2007. Springer-Verlag.

[97] Charles H. Sauer, M. Reiser, and Edward A. MacNair. Resq: a package for solution of generalized queueing networks. In *AFIPS National Computer Conference*, volume 46 of *AFIPS Conference Proceedings*, pages 977–986. AFIPS Press, 1977.

[98] L. Schruben. Analytical simulation modeling. In *Simulation Conference, 2008. WSC 2008. Winter*, pages 113–121, 2008.

[99] James Skene, D. Davide Lamanna, and Wolfgang Emmerich. Precise service level agreements. In *In: Proc. of 26th Intl. Conference on Software Engineering (ICSE*, pages 179–188. IEEE Press, 2004.

[100] Hyung Gi Song and Kangsun Lee. spac (web services performance analysis center): performance analysis and estimation tool of web services. In *Proceedings of the 3rd International Conference On Business Process Management*, BPM'05, pages 109–119, Berlin, Heidelberg, 2005. Springer-Verlag.

[101] Judith Stafford and Kurt C. Wallnau. A technology for predictable assembly from certifiable components. Technical report, Universitaet Karlsruhe, June/September, 2003.

[102] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[103] The OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[104] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[105] Hans Vangheluwe, Ximeng Sun, and Eric Bodden. Domain-specific modelling with atom3. In *In Proceedings of the th OOPSLA Workshop on Domain-Specific Modeling*, 2004.

[106] M. Veran and D. Potier. QNAP 2: A Portable Environment For Queueing Systems Modelling. Technical Report RR-0314, INRIA, June 1984.

[107] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[108] Michael Westergaard. Access/cpn 2.0: a high-level interface to coloured petri net models. In *Proceedings of the 32nd International Conference On Applications And Theory of Petri Nets*, PETRI NETS'11, pages 328–337, Berlin, Heidelberg, 2011. Springer-Verlag.

[109] Michael Westergaard. Rfc: Conveniences in cpn tools, 2012.

[110] Murray Woodside and Dorina Petriu. Capabilities of the uml profile for schedulability performance and time (spt). In *SPT) WORKSHOP SIVOES-SPT RTAS'2004*, 2004.

[111] Dianxiang Xu and Kendall E. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Trans. Softw. Eng.*, 32(4):265–278, April 2006.

[112] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.

[113] Liming Zhu and Ian Gorton. Uml profiles for design decisions and non-functional requirements. In *Proceedings of the 29th International Conference on Software Engineering Workshops*, ICSEW '07, pages 41–, Washington, DC, USA, 2007. IEEE Computer Society.

[114] Steffen Zschaler. Formal specification of non-functional properties of component-based software systems, 2010.

ANNEXE 2 (Modèle dernière page de thèse)

VU :

Le Directeur de Thèse
(Nom et Prénom)

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)