Towards Formal Certification of Software Components

Erik Burger
Institute for Program Structures and Data Organization, Faculty of Informatics
Karlsruhe Institute of Technology, Germany
E-mail: burger@kit.edu

***Abstract*—Software certification as it is practised today guarantees that certain standards are kept in the process of software development. However, this does not make any statements about the actual quality of implemented code. We propose an approach to certify the non-functional properties of component-based software which is based on a formal refinement calculus, using the performance abstractions of the Palladio Component Model. The certification process guarantees the conformance of a component implementation to its specification regarding performance properties, without having to expose the source code of the product to a certification authority. Instead, the provable refinement of an abstract performance specification to the performance description of the implementation, together with evidence that the performance description reflects the properties of the component implementation, yields the certification seal.**

The refinement steps are described as Prolog rules so that the validity of refinement between two performance descriptions can be checked automatically.

***Index Terms*—Components, Certification, Performance Engineering.**

I. Introduction

The term *software certification* usually denotes a process during which an external authority attests certain properties of the software development process in a company. These properties concern the development process or the qualification of the people envolved with it. Despite the fact that a neutral third party is involved, this kind of certification does not guarantee the quality of an actual software product, but, as Voas [1] puts it, rather "make[s] software publishers take oaths concerning which development standards and processes they will use." However, this type of certification is the most common, in contrast to product certification [2].

The process of *software verification*, on the other hand, proves the functional correctness of a piece of software, based on formal verification methods. However, software verification does not cover non-functional requirements such as performance or safety. The advantage of formal verification over a testing and simulation based approach is the exhaustive coverage of all system states, which cannot be reached with classical testing [3].

In our proposed approach, software certification is a formal process of product certification, concerning the non-functional properties of software. This process includes the definition of *Service Effect Specifications (SEFF)* [4] for the specification documents as well as for the implementation. The software issuer can then certify the conformance of the piece of software to the specification document. In combination with techniques that certify the implementation against its SEFF description, the quality of a software product can be certified through the complete chain of software development, from the specification document to the actual application. This is especially important for distributed software development processes, where components of the product are developed by external software suppliers in a globalised market. In order to guarantee the non-functional properties of a piece of software, the quality of the externally developed parts has to be determined. Since software suppliers are often not willing to have their source code inspected by a certification authority or the customer, the definition of non-functional properties of the software provides a level of abstraction that hides implementation details of the product, but can still be used to gain a certificate about the quality of the software.

By using formal methods for the certification of software, the actual process may also be performed by the issuer of the piece of software himself with the support of semi-automatic certification tools, which are provided by a certification authority. The result of this kind of self-certification is reproducable and can be comprehended by the customer of the software, or an external authority. However, the "classical" certification scenario, with the authority performing the certifcation, is also supported. In both situations, human interaction is inevitable, but can be reduced through the introduction of certification tools. In one possible scenario, the compliance of product and specification is specified by the user, and the validity of this compliance is checked by the tool, yielding the certification of the product.

Our approach focuses on non-functional properties, such as performance and resource usage. The envisioned method shall be fully parametric regarding usage profile, deployment and assembly. For the initial version, we will assume that some of these properties are fixed in order to reduce the complexity of the problem.

This paper is structured as follows: In Section II, an overview about the performance modeling constructs in the Palladio Component Model and the reverse engineering methods is given. Section III surveys related work, while the certification scenario is outlined in Section IV. The definition

This work is granted by the GlobaliSE project, a research contract of the state of Baden-Württemberg which is funded by the Landesstiftung Baden-Württemberg gGmbH.

of refinement rules and a formal representation is presented in Section V together with an example in Section VI. The limitations of the presented approach are discussed in Section VII. After an outlook on future work in Section VIII, the paper concludes with Section IX.

II. Foundations

In this section, we will describe the parts of the Palladio Component Model that are relevant for our approach, and outline the reverse engineering approach for the extraction of performance properties from existing component implementations.

A. Palladio Component Model

The *Palladio Component Model (PCM)* [5] is a meta-model for the description of component-based software architectures. The model is designed with a special focus on the prediction of Quality-of-Service attributes, especially performance. Furthermore, a component-based development process is described that contains four types of developer roles: *component developer*, *software architect*, *system deployer* and *domain experts*. The part which is of interest for this paper concerns the *component developer*, who specifies the components and their behaviour.

Service Effect Specifications (SEFF) describe the relationship between provided and required services of a component. In the PCM metamodel, they are defined in the form of *Resource Demanding Service Effect Specifications (RDSEFF)*, which are used for performance prediction contain a probabilistic abstraction of the control flow. RDSEFFs use a notation stemming from UML activity diagrams, i.e. activities are denoted by nodes. For each RDSEFF, a *resource demand* can be specified as well as dependencies of transition probabilities and resource demands on the formal parameters of the service. RDSEFFs can be annotated to each provided service of a component. They describe

- how the service uses hardware/software resources;
- how the service calls the component's required services.

Resource demands in RDSEFFs abstractly specify the consumption of resources by the service's algorithms, e.g., in terms of CPU units needed or bytes read or written to a hard disk. Resource demands as well as calls to required services are included in an abstract control flow specification, which captures call probabilities, sequences, branches, loops and forks.

RDSEFFs abstractly model the externally visible behaviour of a service with resource demands and calls to required services. They present a grey box view of the component, which is necessary for performance predictions, because black box specifications (e.g., interfaces with signatures) do not contain sufficient information. RDSEFFs are not white box component specifications, because they abstract from the service's concrete algorithms and do not expose the component developer's intellectual property. Component developers specify RDSEFFs during or after component development and thus enable performance predictions by third parties.

```
void execute(int number, List array) {

      requiredService1();

      // internal computation
      innerMethod();

      if (number>=0)
            for (item in array)
            requiredService2();
      else
            requiredService3();
}
```

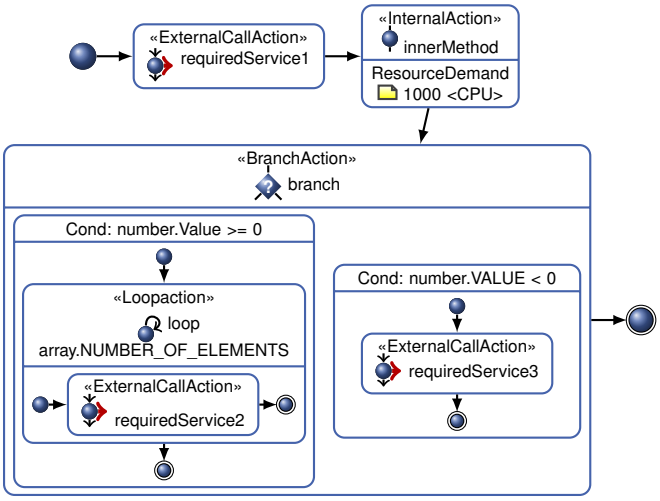

Figure 1. SEFF Example (from [5])

To get an initial idea of RDSEFFs, consider the example in Figure 1. The top part depicts the simplified code of the service execute. The bottom part shows the corresponding RDSEFF. It includes calls to required services as ExternalCallActions, and abstracts computations within the component's inner method into an InternalAction. Control flow constructs are modelled only between calls to required services, while control within the internal computations is abstracted. The example includes parametric dependencies on the branch transitions and the number of loop iterations.

A single InternalAction can potentially subsume thousands of instructions into a single modelling entity as long as these instructions do not interact with other components and perform only component-internal computations. In many cases, an RDSEFF consists only of a few InternalActions and ExternalActions while at the same time modelling large amounts of code.

In addition to internal and external actions, RDSEFFs model control flow (Figure 2): StartAction and StopAction are the beginning and end points of the action chain of a ResourceDemandingBehaviour. Additionally, RDSEFFs may contain branches, loops, and forks.

Branches are expressed by BranchActions, which model XOR control flow alternatives. The contain a number of AbstractBranchTransitions, which can be either Guarded-

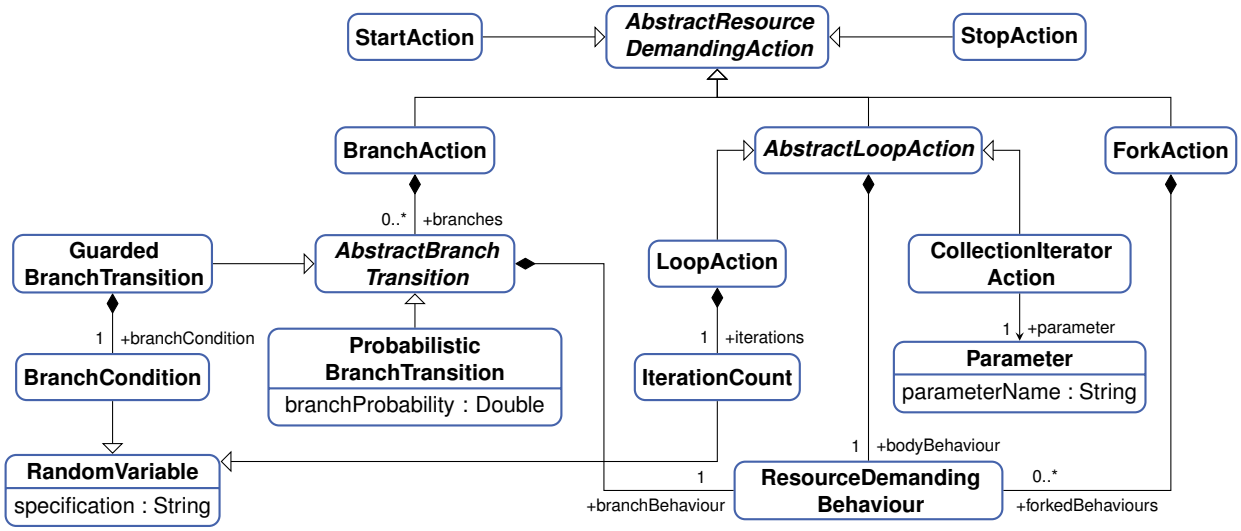

Figure 2. RDSEFF control flow primitives (from [5])

BranchTransitions, i.e. depending on a parameter, or ProbabilisticBranchTransitions. Since the value of an input parameter is not known at design time, guarded branches can also have transition probabilities if the input parameter is specified as a stochastical function in the usage model. The branch probabilities of guarded branches are determined once the usage model is defined. Probabilistic branches can be used to model probabilistic behaviour without dependencies to the input values. Both kinds of branches contain a ResourceDemandingBehaviour, which includes actions executed in the body of the branch.

Loops can be modeled in two ways: LoopAction and CollectionIteratorAction. Both contain inner actions to model the behaviour of the loop body. A LoopAction includes the number of loop iterations as a RandomVariable, which consists of constants or abitrary distribution functions. The RDSEFF metamodel only allows to model loops explicitly, without backward references in the chain of actions within a ResourceDemandingBehaviour. For this reason, the graph structure of an RDSEFF is always a tree, since cycles or loops are modeled explicitly. CollectionIteratorActions are a special construct for loops iterating over a collection, so that the number of repetitions depends on the size of the collection.

ForkActions model AND control flow alternatives. The contained ResourceDemandingBehaviours are executed in parallel, so the succeeding action of a ForkAction is not executed until all forked behaviours have terminated.

B. Reverse Engineering

The performance predictions that can be performed with the Palladio Component Model are applicable in early stages of development as well as in the case of existing software. In order to obtain performance models from black-box components, Krogmann et al. [6] have developed a reverse engineering approach that uses genetic algorithms, static and dynamic analysis and benchmarking. The approach has been validated for Java-based code.

III. Related Work

A. Formal Description of SEFFs

1) Finite Automata: In [7], Firus and Koziolek describe a formal model for *Service Effect Specifications* that is based on annotated finite automata. The automata are enriched by annotations that contain stochastical expressions for transition probabilities, depending on the usage profile. The basic elements are serialization, loops and branches.

However, the actual SEFF metamodel specified as part of the PCM posesses a higher expressive power than finite automata. For example, the number of iterations in a loop may depend on an input variable, which is not expressable by a regular language.

2) Queueing Petri Networks: The semantics of PCM models have been described in [8] in terms of *Hierarchical Queueing Petri Nets (HQPN)* as defined in [9]. The performance-relevant parts of the PCM such as the usage model and RDSEFFs are modeled with the Petri nets, but not the component-related concepts like interfaces and roles. The transformation presented by Koziolek also contains limitations regarding the stochastical expressions such that it is assumed that all distribution functions can be expressed by phase-type distributions and that all random variables are stochastically independent. Furthermore, composite and incomplete component types are not supported yet.

B. Refinement Calculi and Petri nets

In [10], rule-based modifications are applied to high-level Petri nets in order to reach a refinement that preserves safety properties, which are expressed in temporal logic. In contrast to other refinement calculi on Petri nets, the approach does not operate on low-level Petri nets, but on algebraic high-level nets. However, it is concerned with safety rather than performance properties of systems, like most refinement calculi for petri nets [11].

C. Certification of software implementation against SEFF

In [12], Groenda suggests a test-based validation of performance requirements, i.e. resource demands, against a deployed component implementation. With this approach, the validity of SEFF descriptions for component-based systems can be tested for certain parameter ranges that can be specified for the testing scenario. Based on this, automatic testcases are generated, which check the performance results against the specification and vice versa.

In Groenda's approach, the certification step is performed by an external certification authority. It is not intended for self-certification.

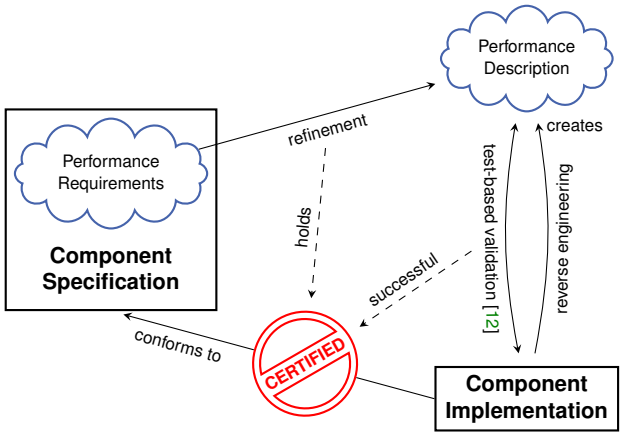

Figure 3. Certification Idea

IV. Scenario

The scenario of certification can be seen in Figure 3. In the proposed component-based software development process, a specifications document for components is created and enriched by non-functional requirements concerning the performance of a component (depicted as a cloud symbol on the left hand side). These requirements have to be laid down formally in the specifications document, using the formalism of *Service Effect Specifications (SEFF)* (see Subsection II-A).

The performance requirements serve as a contract which has to be fulfilled by the implementing party. But Service Effect Specifications can not only be used in the specification of a software system, but also as a means of description for an actual implementation of that system, so we are also going to use them for that.

Based on the specifications document, the implementation of the component is created, usually by a third party supplier. The resulting component is shipped with a description of its performance properties (depicted as a cloud symbol on the right hand side). This description can be determined by the implementing party in two ways: In the first case, the developer of the component creates the performance description manually. The conformance of these descriptions to the actual implementation has to be validated by the approach described in Subsection III-C. In the second case, the reverse engineering techniques mentioned in Subsection II-B are used to create the performance descriptions a posteriori from the implemented component. Assuming the correctness of these reverse engineering techniques, the resulting performance description can be used for a comparison with the requirements.

The availability of both the performance requirements and the performance description in the form of Service Effect Specification is a necessary precondition for the approach proposed in this work. If both artefacts are present, it is to be determined if the implementation SEFF (which serves as the performance description of Figure 3) is a *refinement* of the requirements SEFF (which serves as the performance requirements). For this purpose, a formal refinement definition is specified that allows both parties to check the conformance of implementation to specification regarding the performance properties, on the level of the abstract descriptions in the form of SEFF annotations. With the help of a checking tool, which could be provided by a certification authority, it is then checked if a refinement relation between the two SEFF artefacts holds, and if positive, the certificate can be issued. In the case this performance description has been created manually, a validation has to be performed, which is indicated by "test-based validation" in Figure 3. If the refinement relation holds and the test-based validation is successful, this means that the implementation complies with the performance requirements.

The separation of performance requirements and performance description on the implementation side is necessary for several reasons:

- If one were to attempt a validation of performance requirements against the implementation directly, the approach of Groenda (see Subsection III-C) would not be applicable. The implementation of a component can have performance properties that are very different from the requirements in the sense of providing a much better performance, but still fulfilling the requirements. Thus, the test-based validation would return a negative result which does not reflect the intended purpose of the certification.
- If the performance descriptions are determined by reverse engineering techniques, the resulting description will not be identical with the performance requirements, so that the refinement is necessary for a conformance check.

In addition to these necessary reasons, the separation of performance requirements and performance description has the following advantages:

- In a scenario where the component is developed by a third party, it may not be desireable that all implementation details of the component, i.e. its sources, are published to the certificying party or the customer. In order to protect intellectual property, the performance description can be used as an abstraction of the software.
- The refinement relation can also be used to find an existing component implementation from a repository that meets the performance requirements of the component specification.

V. RDSEFF REFINEMENT

A. Idea

For the definition of refinement between RDSEFFs, two things are required. Firstly, the notion of refinement itself has to be described, and secondly, a formal representation of RDSEFFs and the rules has to be found in order to check if the refinement relation holds for actual RDSEFF instances.[1]

In the following subsection, we will provide a set of refinement rules that defines the refinement relation constructively: If there is an application of rules for two given SEFFs, then the refinement relation holds. The proof consists of a set of valid rule applications.

For the top refinement relation, we propose a definition of refinement that is dependent from the resource demand of the SEFF:

Definition 1: A Service Effect Specification $SEFF_1$ is refined by $SEFF_2$ if, for each resource, the resource demands of $SEFF_1$ are greater or equal than those of $SEFF_2$:

$$SEFF_1 \geq_{RD} SEFF_2 \tag{1}$$

Whether the relation $\geq_{RD}$ is given depends on the elements contained in the SEFFs, since the total resource demand of a SEFF is calculated by the actions contained in it. In the following subsections, the single refinement steps for the different types of actions will be described. In order to express the resource demand of an action a, we write RD_a. A comparison on resource demands means that for every type of resource, the resource demands are compared.

This definition is quite coarse at the moment. The resource demand of a SEFF or of the actions contained in it can be parametric regarding the input parameters. It can be a complex stochastical expression for which the "greater than or equal" comparison of Definition 1 may be difficult to define. At the moment, we assume that the comparison of two stochastical expressions is possible and focus on the structural possibilities of refinement.

B. Refinement steps

An internal action is the simplest building block of a SEFF, since it represents an action that has no externally visible behaviour. It can contain a resource demand. For an internal action, we define the following refinement steps:

Definition 2: An internal action a is refined by

- an internal action b if $a \geq_{RD} b$
- a branch action c containing[2] inner actions d_i, $i \in \mathbb{N}$ if $\forall d_i \in c : a \geq_{RD} d_i$
- a loop action l with a maximum number of iterations $p \in \mathbb{N}$ containing an action e if $RD_a \geq \sum_p RD_e$

Note that the refinement for the branch action does not take into account the branch condition. For this estimation, it is only important that the inner action fulfills the comparison of resource demands, so that $\geq_{RD}$ holds in any case.

The following two definitions are derived from Definition 2, but consider the opposite direction of refinement: It is also possible to refine an action with a less complex internal action. As a consequence of this, the refined SEFF is not necessarily smaller in its structure, which has serious influences on the complexity of checking for a valid refinement relation (see Section VIII).

Definition 3: A branch action b containing inner actions c_i is refined by an internal action a if $\forall c_i \subset b : c_i \geq_{RD} a$.

Definition 4: A loop action l with maximum number of executions $p \in \mathbb{N}$ containing an action c is refined by an internal action a if $\sum_p RD_c \geq RD_a$.

As we can see, all these refinement rules are "worst case" rules: it is assumed that the resource demands are always less or equal, independent of the circumstances, i.e. the usage profile. This is why we do not take the branch condition into account and only regard the maximum number of loop iterations. It is possible to define different notions of refinement, so that for example the resource demands are lower or equal on average, and not in all cases. For the certification scenario outlined in Section IV, this weaker refinement would however not be useful, so we stick to the stronger definition for the moment.

C. Formal definition of refinement

After having defined the refinement steps in Subsection V-B, a formal representation of these rules has to be determined. In the following subsections, we will present several possibilities for the representation of SEFFs and the refinement relation.

1) SEFF metamodel: The definition of Service Effect Specifications is laid down in the Palladio Component Model [5]. There, an EMF metamodel for *Resource Demanding Service Effect Specifications (RDSEFF)* is described. This is a formal definition which serves as a basis for the modeling tools of Palladio and for various model transformations that can be performed on PCM instances.

However, for our purposes, this definition is not sufficient. In order to check whether a refinement between two instances of a SEFF exists, a series of applications of the refinement rules described in Subsection V-B has to be provided. If such a series is found, it is possible to define the concrete refinement relation as a model transformation in terms of the model-driven development. However, the general definition of valid refinement would require a type of higher-order transformation, of which the actual refinement relation would be an instance. But even then, the determination of such an instance from two existing SEFFs would still be a problem which cannot be solved with the mechanism of model transformations. For this reason, we are looking into different formalisms to express the refinement relation.

2) Petri Nets: Higher forms of Petri nets such as Stochastical Petri nets are a possible formalism, since a transformation of SEFFs in to Hierarchichal Queueing Petri nets (HQPN)

[1] In the following, we write SEFF instead of RDSEFF for the sake of brevity.

[2] The containment relation is expressed with set operators here. By containment, we mean the relation depicted as ◆— in Figure 2.

has been performed with limitations in [8]. The process of refinement can be described in terms of Petri net refinement as mentioned in Subsection III-B, but the problem here is also that the source and target net already exist. Refinement usually describes a transformation that is to be applied on a source net, which creates the target. Here, the refinement relation has to be created for existing nets. Furthermore, a class of refinement steps has to be defined which represents our desired refinement relation, and a proof method has to be established that checks if a given refinement complies to this definition.

3) Prolog encoding: Since the refinement steps are defined as single refinement rules, a rule-based approach can be used to define the refinement relation between SEFF instances and also to find the sequence of rule applications that refines the source instance to the target instance. Rule-based languages like Prolog [13] can be used to formally describe the rules of Subsection V-B, provided that an encoding of Service Effect Specifications in Prolog is defined.

```
1  % checks if the demand of action A is greater
2  % or equal than those of the actions in a list
3  demand_geq(A,[Head|Tail]) :-
4        demands(A,D1),
5        demands(Head,D2),
6        D1 >= D2,
7        demand_geq(A,Tail).
8  demand_geq(A,[]).
9
10 % refinement of internalAction to branchAction
11 refinesIntBranch(I1,B1) :-
12       intaction(I1),
13       branchaction(B1),
14       demands(I1,D1),
15       branches(B1,L),
16       demand_geq(I1,L).
```

Listing 1. Refinement of internal action to branch action in Prolog

In Listing 1, we can see the Prolog representation of the second part of Definition 2 (refinement of an internal action by a branch action). For the refinement, we need the recursive helper function `demand_geq()` that compares an action `A` with a list of actions. The clause `branches(A,B)` indicates that action `A` demands a number `B` of resources (for the sake of simplicity in this example, `B` is a number). Lines 12-15 describe the preconditions for refinement, such as the correct types of actions. If the clause in line 16 also evaluates to true, then the refinement relation holds.

VI. Example

A. Description

In the example of Figure 4, two different SEFFs are shown. For the purposes of this example, it is assumed that these SEFFs describe the behaviour of the same service, for example in a scenario where $SEFF_1$ is part of the component specification, whereas $SEFF_2$ is adjacent to the implementation. In order to prove that $SEFF_2$ is a refinement $SEFF_1$, one has to provide a mapping of the single actions of $SEFF_1$ to $SEFF_2$.

In our case, determining a mapping is quite straightforward, since the SEFFs are not complex. In the example,

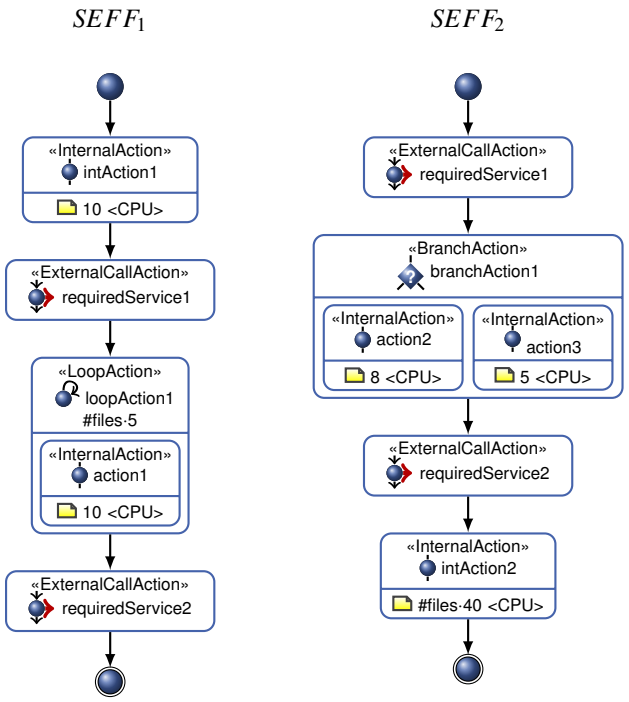

Figure 4. Refinement Example

we have calls to two external services: requiredService1 and requiredService2. To both services, calls occur in $SEFF_1$ as well as in $SEFF_2$; the ordering is the same in the sense that requiredService1 is always called before requiredService2. Since the internal actions between the external calls do not affect the execution time of the external calls in this case, the refinement association is valid for these actions.

For the other actions, matching is more complex. Since we have different kind of actions on both sides, we must investigate all possible combinations. For example, if we compare intAction1 with branchAction1, we can see that branchAction1 contains two elements of the type InternalAction, which are easily comparable with intAction1. In this case, each of the internal actions contained in branchAction1 has a smaller resource demand than intAction1. This means that a refinement relation holds between intAction1 and branchAction1. This comparison is independent of the conditions under which the branch in branchAction1 occurs, since in every case the resource demands will be lower than the one in intAction1.

In comparison, no refinement relation exists between intAction1 and intAction2, since the resource demand of intAction2 grows linearly with an input parameter (the number of files) and is thus not necessarily lower than the fixed resource demand of intAction1. For the same reason, loopAction1 is not refined by branchAction1, since the internal action of loopAction1 has a resource demand that depends on an input parameter, but those of branchAction1 do not, and so it cannot be guaranteed that the resource demand of branchAction1 is always lower than that of loopAction1.

So the only two elements that remain are loopAction1 and

intAction2. The number of iterations of the loop is dependent from the number of files, and the resource demand of the internal action is fixed, so we can calculate that loopAction1 has an overall resource demand of #files·50. The resource demand of intAction2 is also linearly dependent from the number of files, but with a smaller factor, so the refinement relation holds between loopAction1 and intAction2.

B. Prolog

As an example of how such a SEFF can be coded in Prolog, we describe the actions intAction1 and branchAction1 from Section VI in Listing 2. The clauses should be quite self-explaining; as mentioned in Subsection V-C3, we are using natural numbers in the resource demands clause `demands()` for the sake of simplicity here.

```
1 intaction(intaction1).
2 demands(intaction1,10).
3 branchaction(branchaction1).
4 intaction(action2).
5 intaction(action3).
6 demands(action2,8).
7 demands(action3,5).
8 branches(branchaction1,[action2,action3]).
```

Listing 2. Example encoded in Prolog

Now that we have the SEFF instances, we can pose a query, which together with the refinement rule of Listing 1 gives us the following result:

```
1 ?- refinesIntBranch(X,Y).
2 X = intaction1,
3 Y = branchaction1 ?
4 yes
```

Listing 3. Query on refinement

With `refinesIntBranch(X,Y).`, we are querying if there are two entities that fulfill our refinement rule (which, in this example, only applies to the refinement of internal actions to branch actions). The interpreter delivers `intaction1` and `branchaction1` as the correct result.

VII. Assumptions/Limitations

The approach presented in this paper can only be performed if the component specification as well as the component implementation contain performance descriptions in the form of Service Effect Specifications. The correctness of these descriptions is assumed. For the implementation side, this can be checked by the techniques mentioned in Subsection III-C or by reverse engineering techniques that can guarantee the conformance of performance description to implementation. Since the final certificate states that the implementation conforms to the performance requirement laid out in the specifications document, both the refinement check and the conformance check are necessary.

The formal refinement check is only correct under the assumptions that the refinement rules that are used are also correct. The preservation of resource demands or the fulfillment of performance requirements is not checked directly like in [10], but is encoded in the refinement rules: if there is a valid application of rules, then the refinement relation holds. The rules themselves are not formally proved to be correct. The reason for this is that the performance semantics of PCM model instances are not specified formally either, so the correctness of refinement can not be checked formally at the moment.

VIII. Future Work

A. Comparability of Stochastical Expressions

In the presented approach, it is assumed that the resource demands contained in the SEFFs are comparable, so that a "greater than" relation can always be specified. However, resource demands are specified with the *Stochastical Expressions* language of the PCM, and these expressions can reach a complexity for which a comparison is not yet defined. For this reason, the comparability of Stochastical Expressions has to be researched further so that the approach presented in this paper can be applied to SEFFs containing any kind of resource demand definition, and also to take the usage profile into account.

B. Refinement rules

The definition of rules in Subsection V-B only covers some SEFF elements (branch, loop, internal action) and must be extended for all elements of the Palladio Component Model. Further aspects of refinement are not yet covered in the initial version presented in this paper. These aspects include

- rules for the matching of actions, so that all elements in the SEFF are part of a refinement rule
- ordering of actions (especially external actions)
- combining of actions so that multiple actions are refined by a single action
- preconditions for refinement, e.g. comparibility of certain types of stochastical functions

Furthermore, it may be possible that a refinement can be found between two SEFF instances that may require several applications of refinement rules. This would require the definition of intermediate states, i.e. SEFF instances that are neither part of the source nor of the target model. With the current approach, this is not possible yet, but can be included in a preprocessing step that would determine the intermediate states and add them to the set of SEFF instances for which a rule application path must be determined.

C. Formal representation

In order to check the refinement property for existing systems, the encoding of SEFFs in a formal language, e.g. Prolog, has to be specified. For this purpose, a model-to-text transformation can automatically generate the necessary code for an interpreter, which can than determine whether the refinement relation holds. For cases in which the refinement relation can not be found, a mechanism should be developed that points out the problems so that the parts which prohibit a valid refinement can be identified.

D. Validation

For the validation of the approach presented in this paper, a case study would have to cover the development process of a component from specification to implementation. From the implemented component, performance properties can be extracted through reverse engineering techniques. Then, the check for a refinement relation is performed for the SEFF contained in the specification and the generated SEFF, first manually following the refinement rules defined in natural speech, then automatically via the formal checks. This way, it can be determined in which cases a refinement relation can be found and in which not. From this experience, the set of refinement rules can be improved in order to meet the requirements.

IX. Conclusion

The proposed approach extends the notion of software certification into two directions. Firstly, the subject of certification is shifted from functional correctness to the fulfillment of non-functional properties such as performance. Secondly, the certification process is based on formal methods that correlate the specification documents to the actual implementation for component-based systems, so that an actual product certification can take place.

The process requires that software specification documents are enriched with descriptions of performance properties, which can later be checked formally and thus yield a certification that is concerned with the quality of software. The formal foundations of this process make it possible to use certified checking tools for the conformance checks and do not force the software developer to reveal intellectual property in the form of source code to the certification authority, but still gives the customer the possibility to comprehend the check of compliance between the requirements and the delivered piece of software.

Since the process is based on the Palladio Component Model, existing tools and mehods can be used to determine performance requirements and the performance properties of existing systems. For the latter case, reverse engineering methods make the approach also accessible if no performance descriptions in the form of design documents exist for the component implementation.

References

[1] J. Voas, "Developing a usage-based software certification process," *Computer*, vol. 33, no. 8, pp. 32–37, Aug 2000.

[2] K. C. Wallnau, "Software component certification: 10 useful distinctions," SEI, CMU, Tech. Rep. CMU/SEI-2004-TN-031, September 2004.

[3] C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer, "Integrating Verification and Testing of Object-Oriented Software," in *Tests and Proofs. Second International Conference, TAP 2008, Prato, Italy*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Berlin/Heidelberg: Springer, 2008, pp. 192–191.

[4] R. H. Reussner, S. Becker, H. Koziolek, J. Happe, M. Kuperberg, and K. Krogmann, "The Palladio Component Model," Universität Karlsruhe (TH), Interner Bericht 2007-21, October 2007. [Online]. Available: http://sdqweb.ipd.uka.de/publications/pdfs/reussner2007a.pdf

[5] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2008.03.066

[6] K. Krogmann, M. Kuperberg, and R. Reussner, "Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction," *IEEE Transactions on Software Engineering*, 2010, accepted for publication, to appear.

[7] H. Koziolek and V. Firus, "Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation," in *Proc. of the 5th Int. Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'06)*, ser. ENTCS, J. Kuester-Filipe, I. H. Poernomo, and R. H. Reussner, Eds., vol. 176, no. 2. Elsevier Science Inc., March 2006, pp. 69–87. [Online]. Available: http://sdqweb.ipd.uka.de/publications/pdfs/koziolek2006e.pdf

[8] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, Universität Oldenburg, Uhlhornsweg 49-55, 26129 Oldenburg, 2008.

[9] F. Bause, P. Buchholz, and P. Kemper, "Hierarchically combined queueing petri nets," in *11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, ser. Lecture Notes in Computer Science, G. Cohen and J.-P. Quadrat, Eds., vol. 199. Berlin/Heidelberg: Springer, 1994, pp. 176–182.

[10] J. Padberg, M. Gajewsky, and C. Ermel, "Rule-based refinement of high-level nets preserving safety properties," *Science of Computer Programming*, vol. 40, no. 1, pp. 97 – 118, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6V17-42815TS-5/2/8c7f94a0a6e23e6eacebf11014b31312

[11] W. Brauer, R. Gold, and W. Vogler, *A survery of behaviour and equivalence preserving refinements of petri nets*, ser. Lecture Notes in Computer Science, Berlin/Heidelberg, 1991, vol. 483.

[12] H. Groenda, "Certification of software component performance specifications," in *Proceedings of the Fourteenth International Workshop on Component-Oriented Programming (WCOP) 2009*, ser. Interner Bericht. Fakultät für Informatik, Universität Karlsruhe, R. Reussner, C. Szyperski, and W. Weck, Eds., vol. 2009-11, 2009, pp. 13–21. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012168

[13] E. Y. Sterling, Leon ; Shapiro, *The art of prolog : advanced programming techniques*, 2nd ed., ser. (MIT Press series in) Logic programming. Cambridge, Mass: MIT Press, 1994.