Interoperability of Non-functional Requirements in Complex Systems

Norbert Siegmund, Maik Mory, Janet Feigenspan,
Gunter Saake, Mykhaylo Nykolaychuk
University of Magdeburg, Germany

Marco Schumann
Fraunhofer Institut Magdeburg, Germany

***Abstract*—Heterogeneity of embedded systems leads to the development of variable software, such as software product lines. From such a family of programs, stakeholders select the specific variant that satisfies their functional requirements. However, different functionality exposes different non-functional properties of these variants. Especially in the embedded-system domain, non-functional requirements are vital, because resources are scarce. Hence, when selecting an appropriate variant, we have to fulfill also non-functional requirements. Since more systems are interconnected, the challenge is to find a variant that additionally satisfies global non-functional (or quality) requirements. In this paper, we advert the problem of achieving interoperability of non-functional requirements among multiple interacting systems using a real-world scenario. Furthermore, we show an approach to find optimal variants for multiple systems that reduces computation effort by means of a stepwise configuration process.**

I. Introduction

In many areas of life, embedded systems play an important role. An embedded system is a computing device usually embedded in a larger system to control specific functions. These devices sense environmental phenomena (e.g., pressure or temperature), scan and profile goods (e.g., to allow tracking), and communicate with each other. Often, multiple systems cooperate to fulfill a higher goal (e.g., observing quality of goods in a logistic hub). A key requirement to operate a network of interacting devices is *interoperability* [1].

Interoperability ensures that all systems can work together [2]. Important aspects are related to communication. That is, interacting devices must use the same medium to transfer data (i.e., hardware components and communication protocols) and must speak the same language [3]. The language defines syntax (i.e., a uniform data format) and semantics (i.e., a consistent model). There are many approaches to address interoperability at different levels [2], such as standards for communication protocols, middleware techniques [4], or tailored solutions for special application scenarios [3].

Today, most solutions focus on functional requirements of software only [5]. That is, they aim at overcoming differences in *functionality* provided by different programs among embedded devices (e.g., different protocols). However, to enable a reliable and secure interoperability, an approach must consider also requirements on *non-functional properties* of a program. A non-functional property is "... a property, or quality, that the product must have, such as an appearance, or a speed or accuracy property." [6] For example, a sensor network must address the problem of limited energy of its devices by developing a network design that respects non-functional (i.e., energy) requirements. For example, sensor networks use aggregation nodes, which collect and aggregate sensor data, to decrease communication cost. If a node fails due to exhausted energy, other nodes maintain can change their role such that the overall life-time of the network increases.

Such a network design is common for sensor networks, because it meets non-functional requirements of all systems. But how should we design a general network of interacting systems with non-functional requirements different than energy consumption? First, we need programs that can be configured to satisfy different functional and non-functional requirements. This allows us to develop different designs of networks with different non-functional properties whereas the provided functionality stays the same. Second, we have to find a program configuration for each participating system that fulfills all requirements.

Program families, such as software product lines, allow us to tailor a program to requirements of the application scenario and hardware by specifying a *configuration*. A configuration reflects decisions made by stakeholders to ensure that all requirements are met. In practice, it can be a set of preprocessor flags (e.g., #define statements in C/C++), configuration files, scripts, command-line parameters, etc. With a mapping from configuration to implementation, a corresponding program (called a *variant*) is generated (e.g., using #ifdef preprocessor flags in a C/C++ programs) or configured. The remaining challenge is to find the program configurations that yield to valid programs meeting non-functional requirements of the deployed system as well as fulfilling global quality requirements.

In this paper, we propose a staged configuration process to find suitable configurations in a reasonable amount of time. We advert the problem of missing interoperability of non-functional requirements and make the following contributions:

- Show challenges when ensuring non-functional interoperability using a real-world scenario.
- Present novel techniques from software-product-line engineering that enable measurement and configuration of non-functional properties, which is a prerequisite for non-functional interoperability.

978-1-4673-1853-2/12/$31.00 © 2012 IEEE

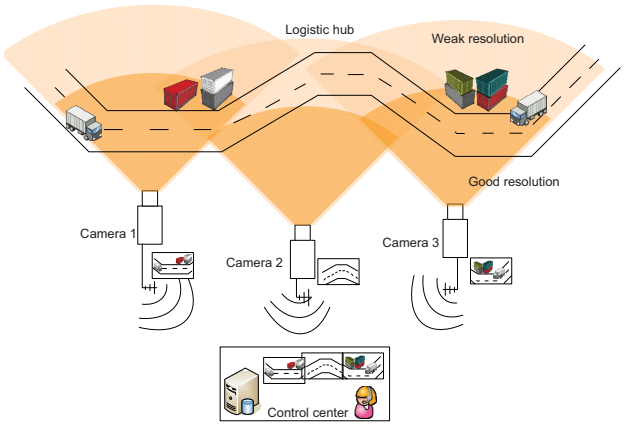

Figure 1. Overview of a camera-based cargo tracking-system of a logistic hub. Picture quality depend on the configured picture resolution, environment, and distance to target. Ethernet bandwidth limits picture quality and frequency.

- Describe a staged configuration approach that provides interoperability of programs within the deployed systems.
- Envision the usage of virtual-reality techniques to enable evaluation and analyses of configurations.

II. Application Scenario

In our interdisciplinary research project ViERforES[1], logistic engineers face the problem of reliably tracking cargo at a logistic hub [7], [8]. A logistic hub is an area at which goods are temporarily stored and subsequently distributed to different locations. It contains different types of goods and some of them have to be treated in a special way. For example, hazardous cargo needs to be stored at special places and some goods have to be kept separated to prevent critical situations. Hence, logistic engineers require a reliable tracking and positioning system to identify and observe moving or stored containers.

We focus on a subsystem of the logistic hub, in which the tracking system is based on intelligent cameras (i.e., cameras have their own software). The cameras take pictures in a specified resolution and send them via Ethernet to a control center. In the control center, all pictures are processed by a video-analysis software to identify moving cargo.

There are three important non-functional properties in this scenario: picture quality, picture frequency (i.e., frequency at which pictures are taken), and WLAN bandwidth. First, the configured resolution of cameras in combination with the used compression algorithm defines the quality of a picture. If quality is low, the identification process may fail, leading to an unsafe state in the logistic hub (e.g., hazardous goods cannot be tracked anymore). Second, the frequency at which pictures are taken influences actuality of cargo tracking, which in turn affects reaction time of the control center in case of an incident. Finally, bandwidth of Ethernet is limited. Sending high resolution pictures or pictures at a too high frequency exhaust bandwidth, which causes unsafe states (e.g., only a subset of cameras is able to send pictures).

In this scenario, there is a trade-off between picture quality, actuality, and bandwidth with the global quality requirements: (a) sufficient picture quality to be able to identify cargo, (b) sufficient actuality to have an up to-date state, and (c) keeping traffic within the available bandwidth. Furthermore, these factors depend on environmental factors. For example, if there is fog or rain, the picture quality decreases, which requires increasing camera resolution. If all cameras increase their resolutions at the same time, we may end up with exhausted bandwidth. Moreover, speed of cargo influences the proper picture frequency. Tracking a moving object requires a higher frequency of pictures. Finally, there are interactions between these factors. For example, a moving container that increases distance to the camera needs a high picture frequency but also a steadily increasing resolution. Hence, there is a trade off between all configuration possibilities to ensure interoperability and safety.

III. Foundations of Non-functional Interoperability

To ensure non-functional interoperability, we have to configure programs of all participating systems such that they do not violate functional or non-functional requirements. A configuration may define a certain set of preprocessor flags that lead to the compilation of different program variants. For example, a stakeholder selects one out of several algorithms that implement the same functionality, but with different non-functional properties. Another possible realization is to use program options, such as command-line parameters, configuration files, and registry entries. For instance, selecting a certain picture-encoding algorithm influences a picture's quality, the size of a picture (which in turn influences required bandwidth), and encoding time. Current approaches aim at reconfiguring a system at runtime [9], [10], [11]. However, they can only cope with a limited variability. That is, the configuration space of our application scenario is so huge that we need an enormous computation power and time to find optimal solutions. This is not possible at runtime and not possible at local embedded devices. Moreover, our used hardware does not support runtime adaptation, because the hardware needs to be flashed.

The question is how to know which configuration option influences a non-functional property and to what extent. Answering this question is far from trivial. We have to quantify the influence of each configuration option on a non-functional property. In addition, we must consider interactions between configuration options that cause unexpected non-functional properties. In the following, we present a recent approach that allows us to measure the influence of a configuration option (referred to as feature) on a non-functional property [12]. As a necessary foundation, we

[1] http://vierfores.de

describe how configurations and variability of programs can be modeled using feature models and present a classification of non-functional properties that determines which properties are measurable (i.e., quantifiable properties) and which are not (i.e., we can provide only qualitative statements about a property). Finally, we explain how we can determine the influence of a configuration option on a non-functional property.

A. Modeling Configuration Options

Feature models are used in product-line engineering to model commonality, variability, and constraints in a domain [13], [14]. A configuration option is represented by a *feature*, – an end-user visible characteristic of a program [14]. Features are structured hierarchically (as shown in Figure 2) and relationships between them enforce that only valid feature combinations can be selected (i.e., only valid configurations can be created). For example, features *JPEG*, *Bitmap*, and *PNG* represent different encoding algorithms that are modeled as alternatives (cf. Fig. 2). That is, exactly one of these algorithms must be selected in a configuration, but not more than one. We can specify additional constraints in a feature model (e.g., feature A requires feature B) to enforce domain and non-functional constraints. To produce a variant, stakeholders select features that satisfy requirements. Based on a mapping from features to implementation, the corresponding program is generated. This model builds the base of configuring multiple systems to ensure non-functional interoperability.

B. Classification of Non-functional Properties

Derived from measurement theory, there are two kinds of non-functional properties: *qualitative properties* and *quantifiable properties*. Qualitative properties are not measurable and their values refer to an ordinal scale in measurement theory. For example, we cannot measure security of different database systems in terms of a metric. However, we can qualitatively rank them, such that we can rate which database is more secure than another. Usability, user-friendliness, and availability are additional examples of this class.[2]

Quantifiable properties can be measured according to some metric. For example, we can measure for different encoding algorithms the time needed to encode a certain picture. Once measured, we can attach the result of the measurement to the corresponding feature in the feature model. This allows us to compute in advance which configuration option influences which non-functional property and to what extent. In this paper, we focus on quantifiable properties only.

C. Measurement of Non-functional Properties

To evaluate whether a configuration meets certain requirements, we can generate the corresponding program and measure its properties. Unfortunately, the configuration space grows exponentially with the number of features. For example, having 33 optional and independent features, we can generate a configuration for each human on the planet, and 320 optional features result in more configurations than estimated atoms in the universe. Hence, a brute-force approach (i.e., measuring all configurations) is infeasible.

In previous work, we developed an approach to tackle this problem [12]. Instead of measuring each configuration, we measure the influence of each feature on a non-functional property. Using the feature model, we compute two configurations that differ only in a single feature and measure their corresponding programs. We use the delta of these measurements to approximate the influence of the differing feature on a non-functional property. This way, we need only $n + 1$ measurements, in which n is the number of features (or configuration options) defined in a feature model. We implemented this approach in the tool SPL Conqueror[3] [15].

IV. Configuration of Non-functional Properties

So far, we presented existing techniques that allows us to measure non-functional properties. Our aim is to use these approaches to configure non-functional properties, such that we ensure interoperability among multiple devices and that global quality requirements (i.e., requirements on the overall system) are met. We propose two steps to configure a complex system with interacting devices: *single-device configuration* and *multi-device configuration*. Single-device configuration ensures that a program's non-functional properties fit to local device-specific requirements. Multi-device configuration addresses non-functional interoperability of interacting devices and ensures that global quality requirements are met. Both configurations must match and contradicting requirements must be found to avoid flaws in a system design. In the following, we describe our approach to configure single as well as multi devices adequately.

A. Single-Device Configuration

To derive a configuration for a single device, a stakeholder defines requirements on a program (e.g., a maximum response time). In addition, the device itself can also state non-functional requirements driven by its resource capacities. For example, a device may constrain the size of a program or the maximum allowed main-memory consumption. Using our measurement approach (cf. Section III-C), we can quantify the influence of each feature on a non-functional property. This allows us to verify any given configuration against the stated non-functional requirements.

To clarify the relationship among features and their influence on non-functional properties, we show a feature model of a customizable camera software annotated with measured and parameterized non-functional properties in Figure 2a. For example, when we select feature *Encryption*,

[2]Note that the classification of a non-functional property depends on the application scenario. That is, a property classified as qualitative in one scenario may be classified as a quantitative property in another scenario, because we can provide a meaningful metric.

[3]http://fosd.de/SPLConqueror

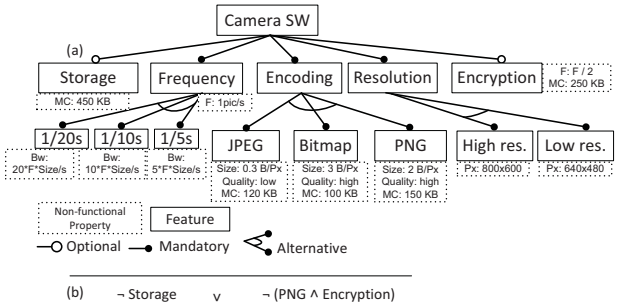

Figure 2. Feature model of a camera-software product line. The influence of a feature on a non-functional property is directly annotated to the corresponding feature. In part (b), additional boolean constraints are introduced that constrain variability according to non-functional requirements. Bw: bandwidth; Px: pixel; F: frequency at which pictures are taken; MC: memory consumption.

memory consumption (MC) increases by 250 KB, and the refresh rate halves. A configuration's property is the sum of the influences of all selected features. If we select features *Encryption*, *Storage*, *Bitmap*, and *High resolution*, the resulting configuration yields to an MC of 800 KB and a picture size of 1.4 MB, because of 800×600 pixel from feature *High resolution* and 3 Byte per pixel from feature *Bitmap*. Since feature *Frequency* is mandatory, we have to select a subfeature, e.g., *1/10s*. Now, we can compute the used bandwidth, because feature *1/10s* defines the bandwidth as: $10 * Frequency * size\ per\ second$; we calculate (10 / 2) * 1.4 MB per second, which is 7 MB/s. Note that we have to halve the frequency, because of feature *Encryption*.

The configuration is mapped to the according implementations. That is, if a configuration maps to a set of preprocessor flags, the result is a program variant compiled only with the given features. One problem remains: How can we identify a valid configuration without using a trial-and-error approach?

Current approaches transform a feature model into an alternative representation (e.g., boolean). The result is a constraint system for which we define an objective function that covers our non-functional requirements. Next, we use a *constraint satisfaction problem (CSP)* solver [16] to compute an optimal configuration. Unfortunately, finding an optimal configuration is NP-hard [17]. If we additionally consider non-functional requirements of other devices, the problem becomes too complex to solve for feature models beyond a tiny size. Hence, we propose a *staged configuration process*.

The key idea is to find feature combinations that always yield to invalid programs and remove them from the feature model to shrink the search space. The first step is to find the set of (all) *invalid* feature combinations C_{inv}. That is, we identify those configurations i that always fail to fulfill non-functional requirements:

$$C_{inv} \subseteq C_{all} \ \mid \ \forall\ i\ in\ C_{inv} :\ i\ is\ invalid$$

Note that this process can become too complex if the influence of features are not always positive or always negative. That is, if there are features with a positive influence on a property and other features with a negative influence, the search space to find invalid combinations is already exponential in the number of features. In this case, we find only a subset of all invalid feature combinations that cover a large proportion of all invalid configurations.

Once identified, we encode invalid feature combinations via additional boolean constraints. For example, a non-functional requirement that limits memory consumption to 400 KByte prohibits selecting feature $Storage$ as well as the combination of features $Encryption$ and PNG (cf. Figure 2a). We compute the according boolean constraints as shown in Figure 2b: $\neg Storage \vee \neg Encryption \wedge \neg PNG$. Using boolean constraints instead of numerical values decreases computation time to find a valid configuration substantially, because we can use *satisfiability (SAT)* solvers instead of *constraint satisfaction problem (CSP)* solvers. That is, a SAT solver maintains a set of boolean variables and finds one valid allocation. In contrast, a CSP solver assigns values to the boolean variables and maintains numeric constraints over these values. Hence, a CSP solver does not only satisfy the boolean constraints, but also the numeric constraints. Another benefit of SAT solvers is that the search space reduces to $C_{all} - C_{inv}$ configurations, which limits variability to find only those configurations that yield always to a valid program.

Finding Invalid Feature Combinations: To find invalid feature combinations, we use the hierarchical structure of feature models to reduce computation effort. That is, if we detect that a certain feature or feature combination does not meet a non-functional requirement, we mark also all child features of these features as invalid, because child features can only further increase the value of a non-functional property.

Our algorithm creates an ordered list of all features according to its contribution to a non-functional property. For instance, we compute the following order for memory consumption: *Storage*, *Encryption*, *PNG*, *JPEG*, and *Bitmap*. Next, we search for features that do not meet the given requirement when selected. For example, if a feature contributes with 50 KB to a program's memory, but a requirement limits the size to 30 KB, selecting this feature always yields to an invalid product (as well as selecting its child features). If we detect such a feature, we remove this feature and all child features from the following computations. Furthermore, we introduce a boolean constraint in the feature model: *NOT A*. Hence, selecting an invalid feature always yields to false.

In the next run, we verify for each pair of features (beginning with the two top features of the ordered list) whether they satisfy all non-functional requirements (e.g, whether *Storage* and *Encryption* result in a valid program). If we find an invalid combination, we introduce a boolean constraint: *NOT (B AND C)*. Then, we continue with combinations of three features and so on. We abort a run (e.g., when searching with three features in combination), when adding a new feature to a feature combination leads to a valid configuration. Remember, due to the ordering of

features, we know that all following feature combinations would also satisfy all requirements.

To validate a certain feature combination, we build a configuration that contains the according features and all further required features, such as parent features or features that have to be included due to a certain constraint. Since we can use the boolean constraints of preceding steps, the overall computation time is low. As a result, we obtain a feature model that can be configured within the boundaries given by the used system.

B. Multi-Device Configuration

The goal of this step is to ensure non-functional interoperability. That is, we aim at achieving compatibility of the configurations of interacting systems. This includes compatibility of requirements stated by the communication medium (e.g., bandwidth limitations), by the participating systems (e.g., maximum allowed requests per minute to save energy), and by global quality requirements (e.g., responsiveness and reliability). Global requirements (also known as *quality-of-service (QoS)*) emerge from the interplay of multiple devices. To address all issues, we need a model that describes which devices interact with each other and which devices contribute to a global quality property.

In Figure 3, we illustrate such a model for the logistic hub application scenario.[4] Although the cameras do not communicate with each other, they contribute to the global property bandwidth, because of sending images. This requires us to configure all cameras according to that overall requirement. Furthermore, each camera interacts with the control center. We have to find valid configurations for all cameras according to non-functional requirements of the control center (i.e., picture quality and picture frequency). Based on this model, the remaining task is to infer an *objective function* (e.g., minimize used bandwidth) that accounts for all non-functional requirements over the whole system.

To construct an objective function, we specify non-functional requirements as side conditions first. For example, we list all side constraints defined by the control center for each camera ($C1$, $C2$, and $C3$)

$$\begin{aligned} Pixel_{C1} >= \quad & 640 \times 480 \quad // \quad Control\ center \\ Pixel_{C2} >= \quad & 800 \times 600 \quad // \quad requirements \\ Pixel_{C3} >= \quad & 640 \times 480 \quad // \quad ... \end{aligned}$$

as well as all global quality constraints:

$$\begin{aligned} Bandwidth <= \quad & 10 MBit \quad // \quad Ethernet\ limit \\ Frequency >= \quad & 10\ Pictures/s \quad // \quad Global \\ Picture\ quality >= \quad & Medium \quad // \quad Requirement \end{aligned}$$

Furthermore, we have to express the relationships between different non-functional properties in terms of functions so that we can compute the outcome of global properties for a specific configuration. For example, we describe the influence of the selected camera resolutions and picture frequency on the bandwidth with the following formula:

$$\begin{aligned} Bandwidth = \quad & Frequency_{C1} * Size_{C1} \\ + \quad & Frequency_{C2} * Size_{C2} \\ + \quad & Frequency_{C3} * Size_{C3}, \end{aligned}$$

where $Frequency_{C1}$ denotes the selected frequency sub-feature for camera 1 and $Size_{C1}$ denotes the binary size of a picture taken by camera 1. Since we have already measured how each encoding algorithm affects size of an image given a certain resolution (cf. Figure 2a), we compute *in advance* whether a certain configuration violates any given requirements.

Finally, we specify an objective function as a global optimization goal. In our application scenario, we aim at maximizing the frequency for all cameras:

$$Max(\frac{Frequency_{C1} + Frequency_{C2} + Frequency_{C3}}{3})$$

Although we use a simple example, it is not trivial to find a configuration for each camera such that all requirements are met. Obviously, we do not want to manually find a valid configuration. Instead, we aim at finding an optimal configuration in an automated manner. To this end, feature models for all cameras are translated to a boolean representation. Features are represented by variables that can have only one of two states: true (selected) or false (not selected). Together with all equations and side conditions, we construct a constraint system, which can be solved by a CSP. Note that our first step minimizes the number of variables in the constraint system such that computing a configuration can be done in a feasible amount of time.

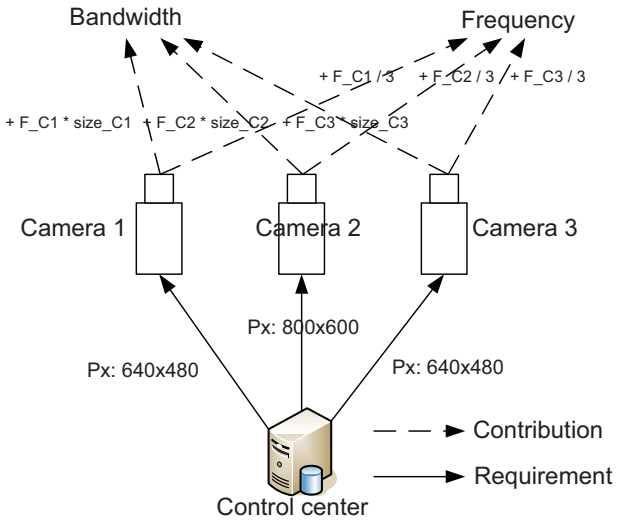

Figure 3. Interoperability model of the logistic hub describing which devices interact and how each device contributes to a property.

[4]We omit the specifications of this model, because they are out of scope of this paper.

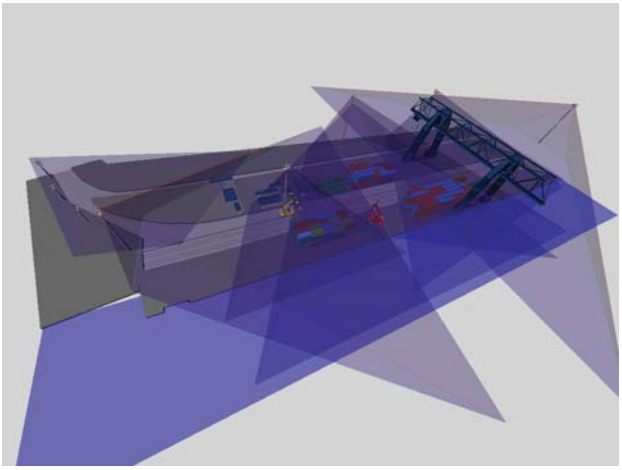

Figure 4. Virtual reality image of a logistic hub including different camera devices. Blue planes visualize line of sight of cameras.

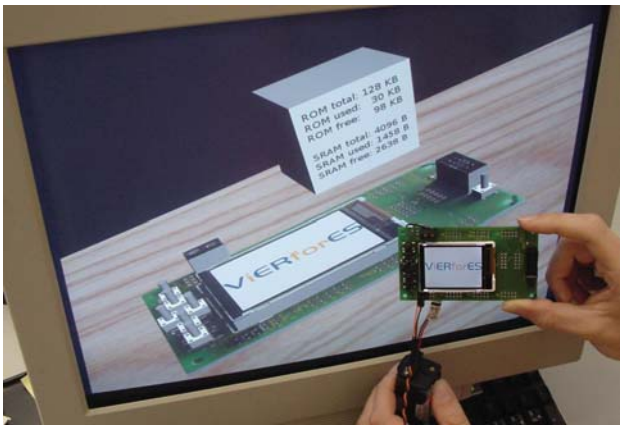

Figure 5. Connecting an embedded device with the virtual reality and transmitting current non-functional properties.

V. Virtual Reality

Today, *virtual-reality (VR)* environments are more used to design and test a system before it is actually deployed. In Figure 4, we demonstrate a VR model of our logistic hub scenario. The line of sight of cameras are visualized with blue planes. In a simulation, containers are moved and we can verify their tracking status. We have to enrich this model for other non-functional properties, such as bandwidth, and encode the picture size within the simulated cameras. This test infrastructure is a key element of the practicality of our approach. That is, we simulate multiple devices with different specifications in the VR resulting in varying non-functional requirements. With such an experimental setup, we can measure the influence of our configurations and record possible interactions and side effects. Based on test results, we can alter the network design and configurations to find an optimal setup of all participating systems in the end. Finally, we deploy this setup including all configurations in reality.

Real-world Data: In previous work, we developed a technique based on services to connect real embedded systems with their counterparts in the VR [18]. In Figure 5, we show such a connection of an embedded device including the visualization of its current non-functional properties. With this approach, we connect real systems within the VR environment to use actually measured data of non-functional properties. Furthermore, we can supply the VR with up-to-date data (e.g., current utilization of bandwidth and response time) making optimizations more accurate. Finally, we may test changes to the real system in the VR first, which increases reliability of the overall system.

Dynamically Changing Requirements: So far, we compute a single optimal configuration for a set of requirements. However, some requirements may change over time or non-functional properties of a program change depending on the environment. In our logistic-hub scenario, picture quality depends considerably on the environmental conditions. If there is rain or fog, the control center demands a higher resolution to be able to track an object. This, in turn, requires a reconfiguration of the camera software which can exhibit large problems. For example, JPEG compression produces larger files when the camera takes bright images. To avoid bandwidth problems, we may change the compression algorithm for some cameras.

We envision two possibilities to address dynamically-changing requirements and parameters: (a) simulating changes in VR and compute optimal configurations in advance and (b) parallel execution of VR with the real system. We can simulate changing environmental conditions in VR environment, such that we *precalculate* suitable configurations. In VR, processing power and time is not critical compared to the limited processing power of embedded systems or the limited time to respond on changes during running a system. This gives us the opportunity to test the system for, e.g., weather forecasts. We are able to create profiles for such situations that, once detected in reality, just lead to a reconfiguration of the global system according to this profile. This leads to an automated verification and simulation system that identifies hazardous system states for certain environmental situations and non-functional requirements.

VI. Related Work

There is some related work to find optimal configurations. Benavides and others use CSP solvers to find the best configuration for a given objective function [16]. White and others [17] allow users to specify hardware constraints on non-functional properties to generate a suitable variant. Both works use filtered Cartesian flattening to approximate a nearly optimal product for large scale feature models. In contrast to our approach, both do not consider the measurement and computation of a feature's non-functional properties nor account for interconnecting devices.

The project *COMQUAD* focuses on techniques for tracing and adapting non-functional properties in component-based systems [19]. The approach requires a dedicated component model, which is an extension of *Enterprise JavaBeans* and *CORBA Components*. Although component design may be used for interconnecting systems, they cannot express constraints to optimize an overall quality property. Furthermore, we consider a wide variety of properties and address the exponential number of products that occur during the configuration process.

Czarnecki and others proposed a staged configuration approach for product lines [20]. In each configuration step, a feature model is consecutively refined such that variability is reduced by selecting features. We use also a staged configuration, but introduce boolean constraints to a feature model to prohibit selecting features. These constraints are based on numeric requirements, which are not considered by Czarnecki and others.

VII. Conclusion

In this paper, we highlighted the problem of missing non-functional interoperability using a real-world scenario of a logistic hub. To ensure non-functional interoperability, software of multiple interacting devices must be configured in such a way that they meet all non-functional requirements. To enforce non-functional constraints for single devices and the global system, we proposed a staged configuration. First, a feature model describing the configuration options is refined such that only configurations can be generated that meet the local device-specific requirements. Second, configurations for all participating devices in a system are computed using a constraint satisfaction problem solver. The necessary input is given by an interoperability model of the global system that describes which devices interact and which requirements are present.

In future work, we implement the staged-configuration processing in our tool SPL Conqueror. Using our application scenario, we simulate different configurations and evaluate with existing requirements our configuration process within a virtual-reality environment.

Acknowledgments

The work of Siegmund, Mory, Feigenspan, Saake, Nykolaychuk, and Schumann is supported by the German ministry of education and science (BMBF), Nb. 01IM10002B. The work is part of the ViERforES-II project.

References

[1] P. Wegner, "Interoperability," *ACM Comput. Surv.*, vol. 28, pp. 285–287, 1996.

[2] S. Diallo, H. Herencia-Zapana, J. Padilla, and A. Tolk, "Understanding interoperability," in *Symposium of Emerging M&S Applications in Industry and Academia*. Society for Computer Simulation International, 2011, pp. 84–91.

[3] R. Shah and J. Kesan, "Interoperability challenges for open standards: ODF and OOXML as examples," in *International Conference on Digital Government Research*. Digital Government Society of North America, 2009, pp. 56–62.

[4] C. Jovellanos, "Semantic and syntactic interoperability: in transactional systems," in *International Conference on Electronic Commerce*. ACM, 2003, pp. 266–267.

[5] D. Smith, E. Morris, and D. Carney, "Interoperability issues affecting autonomic computing," in *Workshop on Design and Evolution of Autonomic Application Software*. ACM, 2005, pp. 1–3.

[6] S. Robertson and J. Robertson, *Mastering the requirements process*. ACM Press, 1999.

[7] N. Siegmund, M. Pukall, M. Soffner, V. Köppen, and G. Saake, "Using software product lines for runtime interoperability," in *Proceedings of Workshop on Reflection, AOP and Meta-Data for Software Evolution*. ACM Press, 2009, pp. 1–7.

[8] N. Siegmund, J. Feigenspan, M. Soffner, J. Fruth, and V. Köppen, "Challenges of secure and reliable data management in heterogeneous environments," in *International Workshop on Digital Engineering*. ACM, 2010, pp. 17–24.

[9] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Using feature models for developing self-configuring smart homes," in *Proceedings of the International Conference on Autonomic and Autonomous Systems*. IEEE, 2009, pp. 179–188.

[10] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44 –51, 2009.

[11] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel, "Tailoring dynamic software product lines," in *International Conference on Generative Programming and Component Engineering*. ACM, 2011, pp. 3–12.

[12] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines," in *Proceedings of International Software Product Line Conference*. IEEE, 2011, pp. 160–169.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," SEI, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[14] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[15] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, 2011.

[16] D. Benavides, A. Ruiz-Cortés, and P. Trinidad, "Automated reasoning on feature models," in *International Conference on Advanced Information Systems Engineering*. Springer, 2005, pp. 491–503.

[17] J. White, B. Dougherty, and D. C. Schmidt, "Selecting highly optimal architectural feature sets with filtered Cartesian flattening," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1268–1284, 2009.

[18] V. Köppen, N. Siegmund, M. Soffner, and G. Saake, "An architecture for interoperability of embedded systems and virtual reality," *IETE Technical Review*, vol. 26, no. 5, pp. 350–356, 2009.

[19] S. Göbel, C. Pohl, S. Röttger, and S. Zschaler, "The COMQUAD component model: Enabling dynamic selection of implementations by weaving non-functional aspects," in *International Conference on Aspect-oriented software development*. ACM, 2004, pp. 74–82.

[20] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged configuration using feature models," in *International Software Product Line Conference*. Springer, 2004, pp. 266–283.