JOT
JOURNAL OF
OBJECT TECHNOLOGY



# Methodical and Formally Verified Model-Driven Architecture Refactoring

**Lars Fischer, Hendrik Kausch, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Lucas Wollenhaupt**
Software Engineering, RWTH Aachen University, Germany

**ABSTRACT** Verification is necessary to ensure the correctness of safety-critical software systems. When developing such systems in an agile way, it is important to guarantee that the correctness still holds after refactoring, e.g., no new behavior is introduced. To support the iterative development of such systems, we have translated refactoring patterns based on syntactic transformations of pipeline architectures to Isabelle, an interactive theorem prover. Isabelle is employed to verify refactoring steps performed on component-and-connector architectures transformed from SysMLv2 models. In particular, we have translated several verified architecture refactoring patterns to Isabelle. The application of the development patterns is demonstrated by a case-study in which a secure communication channel is added to an architecture modeled in SysMLv2. We envision that by utilizing model-driven system engineering in conjunction with development patterns and tool-supported formal behavior verification, engineers can effectively improve and refactor existing systems without compromising previously certified correctness results.

**KEYWORDS** Architecture Refactoring, Development Pattern, Formal Verification, Theorem Proving, Model-Driven, SysML

## 1. Introduction

Testing is not always sufficient to safeguard the quality of a software system. While quality assurance using automated unit tests can detect many faults within a software system, it cannot ensure their absence (Dijkstra 1972). Thus, in cases where a system's failure to comply with prescribed requirements might lead to significant damage or costs, formal verification of the software's correctness and guarantees becomes crucial. Additionally, regulatory bodies may require thorough certification of safety-critical systems. Formal verification contributes to ensuring that regulatory requirements can be met.

Often, systems are refactored throughout their development and life cycle to reduce their internal complexity or otherwise enhance it in a manner that does not change its overall behavior, e.g., replacing a car's engine with a different model with roughly equivalent specifications. Such substantial reconstructions overhaul and refactor major parts of the system, but by splitting the refactoring process into several smaller steps the complexity is still manageable. Fowler (1999) defines refactoring as changing the internal structure of a system without changing the overall behavior. As such, it constitutes a special case of refinement in the mathematical foundation for specification and verification of distributed systems FOCUS (Broy & Stølen 2001). The behavioral equivalence between the original and refactored system can be formally verified by proving that a refinement relation holds in both directions. Note that in many cases the context of the system must be considered, as well. A gasoline-based engine and an electric engine serve the same purpose but behave very differently. However, these differences can be abstracted away in the context of a larger system. Instead, the behavioral equivalence under a system invariant is considered. For electrical motor refactoring it might state, that the energy transmitted via fuel to the gasoline engine is equivalent to the electrical energy transmitted to the electric engine.

Tool-support is desirable not only to reduce the manual effort required for formal verification of distributed systems but also to reduce manual errors. MontiBelle (Kausch, Pfeiffer, et al. 2021) is a tool enabling the verification of software architecture speci-

**JOT reference format:**
Lars Fischer, Hendrik Kausch, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Lucas Wollenhaupt. *Methodical and Formally Verified :Model-Driven Architecture Refactoring.* Journal of Object Technology. Vol. 24, No. 2, 2025. 
http://dx.doi.org/10.5381/jot.2025.24.2.a13

fications given as SysMLv2 (SST 2023) or MontiArc (Haber et al. 2012) models. This allows coupling continuous verification with agile development, thus erasing errors in the early development stages and simplifying integration and potential maintenance later on. Both SysML and MontiArc allow the modeling of hierarchical component-and-connector architectures, where components are specified regarding their input/output behavior and connected by channels that allow for asynchronous communication. Using FOCUS as a mathematical underpinning for these architecture models, allows formal analysis, e.g., verifying refinement between models.

To formally prove the architectural refinement of these models, the MontiBelle project employs the interactive theorem prover Isabelle (University of Cambridge and Technische Universität München 2023). MontiBelle includes a transformation tool (Kausch, Michael, et al. 2021; Kausch et al. 2020) that translates architectures into equivalent Isabelle theories and generates a corresponding lemma for each refinement step. These lemmata can then be formally proven with the help of an existing core library of already-proven lemmata and proof schemata.

**Contribution** We implemented and formally verified generalized versions of refactoring patterns for component-and-connector architecture models introduced by (Philipps & Rumpe 1997, 1999) in the theorem prover Isabelle. These refactoring patterns include:

1. adding or removing components,
2. adding or removing channels,
3. refactoring under an invariant,
4. (un-)folding of sub-architectures.

In contrast to (Philipps & Rumpe 1997, 1999), we do not require components to have input or output channels when adding or removing components. They discuss refinement under invariant, whereas we extend this to refactoring. All theorems stated apply to both the addition or the removal of components or channels. While (Philipps & Rumpe 1997, 1999) implicitly use the concept of effects, we formalize this notion and use it explicitly. Furthermore, we present a SysMLv2 architecture refactoring process that demonstrates integrated and agile development while supporting formal verification of correctness. Thereby we extend both MontiBelle (Kausch, Pfeiffer, et al. 2021) as wells as FOCUS (Broy & Stølen 2001).

**Structure** The remainder of this paper is structured as follows: In the following section, we discuss related work, in particular publications about the refinement of pipeline architectures, as well as other works concerning model refinement. In section 3, we describe the foundational concepts needed to understand our contributions' technical and formal aspects. Then, section 4 introduces the refactoring patterns with an example that we will refer back to in the later sections. In section 5, we outline the mode-driven development process and the functionality of the transformation tool, which translates SysML models into corresponding Isabelle theories and generates lemmata for each refinement step. The architecture refactoring patterns and their application are discussed in detail in section 6. Finally, section 7 concludes and gives an outlook on future work.

## 2. Related Work

According to Mens & Tourwé (2004) the term refactoring was first introduced by Opdyke (1992) to describe the restructuring of object-oriented programs. The concepts breakthrough came with its integration into the software development process *Extreme Programming* (Beck 1999). Fowler (1999) informally applies refactoring techniques to Java, presenting 72 refactorings. As discussed by Philipps & Rumpe (2003), the notion of restructuring the internals of a system without changing its observable behavior puts refactoring close to the concept of refinement, pioneered by Dijkstra (Dahl et al. 1972), Wirth (1971), and Bauer (Bauer & Woessner 1982).

Model refinement and refactoring require some formal notion of model semantics. Harel & Rumpe (2004) discuss this notion independent from any specific modeling language and conclude that formal semantics for any modeling language includes a semantic mapping that maps syntactically valid models to a set of legal instances within a well-defined and well-understood semantic domain. This formal notion of model semantics is also used for semantic differencing (Maoz et al. 2010, 2012), where two models are compared regarding their legal instances. Semantic differencing operators can also be utilized to check for model refinement and refactoring. For example, the operator `CDDiff`, introduced by Maoz et al. (2011b), takes two class diagrams as input and uses a bounded model-checking approach to find instances of the first diagram that are not part of the semantics of the second diagram. Should no such instances exist, then the first diagram is a refinement of the second. Note, however, that the operator only detects instances within a specified bound and, therefore, generally provides no guarantees on refinement. Since then, `CDDiff` has been expanded to allow for semantic differencing under an open-world assumption (Ringert et al. 2023). However, its completeness is still limited by the bounded model-checking approach. Similar semantic differencing operators include variants of `ADDiff` (Maoz et al. 2011a; Kautz & Rumpe 2018) for semantic differencing on activity diagrams, as well as operators for feature models (Drave, Kautz, et al. 2019), statecharts (Drave, Eikermann, et al. 2019; Butting et al. 2017), and sequence diagrams (Kautz 2021). While these operators are fully automatic, they rely on bounded model-checking approaches or finite search spaces. They are, therefore, significantly more limited in their expressiveness when compared to verification approaches using Isabelle.

In this paper we are concerned with connector-and-component architectures modeled in SysML and MontiArc. Both languages describe component-behavior either by a specification in relational logic relating the input and output of the component (Broy & Stølen 2001), by a state-chart (Harel 1987; Rumpe 2017), or by hierarchical decomposition (Kausch, Michael, et al. 2021). The components are then connected to send data from one component to other components via channels. An architecture can have multiple hierarchical levels, in case a sub-component is also defined as a composition of com-

ponents. These sub-architectures are abstracted (or folded) away on a higher hierarchical level. A component's behavior can be underspecified, i.e., allowing multiple correct implementations and non-determinism to support iterative development.

The architecture refactoring patterns that we implemented in Isabelle are generalized versions of patterns introduced by (Philipps & Rumpe 1997, 1999). A precise mathematical model of data-flow architectures and their refinement based on FOCUS was employed to manually prove the correctness of the aforementioned patterns. However, to our knowledge, these patterns have never before been implemented or proven with formal verification software. FOCUS itself was introduced by Broy et al. (1992) and designed to specify the input/output behavior of software components using relational logic and the concept of message streams for modeling asynchronous, timed communication between components via directed channels. The model allows for under-specification and has a precise notion of formal refinement, which also ensures that the refinement of an individual component implies the refinement of the whole system. Consequently, verifying the refinement of a complex system can be reduced to verifying the refinement of each subcomponent. As of last year, Broy (2023) continues to extend the FOCUS theory. Note, however, that the model of data-flow architectures in (Philipps & Rumpe 1997) and (Philipps & Rumpe 1999) uses behavioral semantics based on stream-processing functions instead of relations. In this paper, we also use a notion of effects of channels first introduced as an influence relation in (KRW20 2020) to conceptualize prerequisites for some of the refactorings discussed.

Alternatives to FOCUS include other formalisms such as Communicating Sequential Processes, as described in (Hoare 1985) and used in, e.g., (Murray & Lowe 2009), Calculus of Communicating Systems (Milner 1982), $\pi$-calculus (Milner 1999), Ptolemy (Lee 2016), Temporal Logic of Actions (Abadi & Lamport 1994), and Petri Nets (Reisig 1985).

They all constitute possible mathematical underpinnings for reasoning about system specifications since they support nondeterminism, underspecification, a notion of behavioral refinement, time-sensitive specifications, and hierarchical decomposition.

Petri Nets are, for example, used within the Palladio approach (Reussner et al. 2016) to provide a semantic foundation for modeling and analyzing distributed component-based systems to predict their performance, throughput, and response times (Kounev 2006; Koziolek & Reussner 2008). These analyses are simulation-based. The interactive theorem prover Isabelle (University of Cambridge and Technische Universität München 2023), on the other hand, enables machine-supported and automated proof searches and allows the generation and verification of machine-based and machine-checked formal proofs. Compared to model-checking, this approach does not have a significant state-explosion problem but, so far, lacks full automation.

Just as with FOCUS alternatives to Isabelle exist. E.g., while the theorem prover Dafny (Leino 2010) allows generating correct code from proofs, Isabelle does not require termination, thus permitting the definition of potentially non-terminating functions, e.g., fixpoints. The well-established theorem prover Coq is based on the calculus of inductive construction (Bertot & Castéran 2013), which allows defining dependent types (Barendregt et al. 2013), but misses, similar to the Proof Verification System (PVS) (Owre et al. 1992), a powerful general proof finder like Isabelle's sledgehammer (Böhme & Nipkow 2010). Lean (de Moura et al. 2015) is an alternative promising and upcoming theorem prover, so far mainly used for verifying mathematical theory, e.g., the theorem of liquid modules (Scholze 2022) or topology theorems for perfectoid spaces (Buzzard et al. 2020), but lacks the long history and intensive evaluation of Isabelle's archive of formal proofs (University of Edinburgh and Technische Universität München n.d.).

Finally, the Correctness-by-Construction (CbC) approach (Kourie & Watson 2012) should also be mentioned. It constitutes a formalism for creating a correct program via incremental refinement of pre-/postcondition specifications based on Hoare logic (Hoare 1969). Moreover, the approach has since been applied to component-based software architectures in the open-source tool ArchiCorC (Knüppel et al. 2020). However, to our knowledge CbC lacks any notion of time-sensitivity in its specifications.

## 3. Foundation

In this section, we present the mathematical foundations of FOCUS required for the following sections. A more detailed introduction to FOCUS is given in (Broy & Stølen 2001). The related implementation of FOCUS in the theorem-prover Isabelle used in this paper is presented in (Bürger et al. 2020) and extended and demonstrated in case studies in (Kriebel et al. 2019; Kausch, Michael, et al. 2021; Kausch, Pfeiffer, et al. 2021; Kausch et al. 2023).

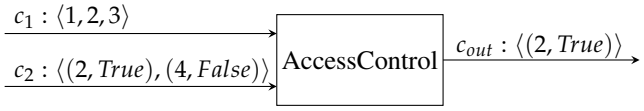



**Figure 1** Component with exemplary input/output

FOCUS is used to mathematically describe component&connector systems that send messages over directed communication channels. Figure 1 shows one component named `AccessControl`. The component has two input channels named $c_1$ and $c_2$ and produces output on the channel named $c_{out}$. The observation of the transmitted messages is called a *stream*. In this example, on channel $c_1$, the first transmitted message is 1, followed by 2 and 3. Other message types, such as tuples or booleans, are also possible. Streams are the primary datatype in FOCUS. They can be finite or infinite. Encoding observations with infinitely many messages allows FOCUS to reason about liveness properties.

**Definition 1** (Stream (Bürger et al. 2020))**.** *The set of all streams over message-type $M$ is denoted by*

$$M^\omega := \{\epsilon\} \cup \{\langle m_1, m_2, \ldots, m_i \rangle \mid i \in \mathbb{N}. \forall j \in [1, i].\, m_j \in M\} \cup \{\langle m_1, m_2, \ldots \rangle \mid \forall j \in \mathbb{N}. m_j \in M\}$$

*which is the union of the empty stream, the set of all finite streams and the set of all infinite streams.*

As seen in Figure 1, the input of a component can consist of multiple streams. This is encoded in the *bundle* datatype, which maps from channels to streams. Crucially, not all messages may be transmitted on a channel. For example, one channel may transmit only boolean messages, while another may transmit only natural numbers. Thus, each channel $c$ has a set of allowed messages denoted by $M_c$.

**Definition 2** (Stream Bundle (Bürger et al. 2020))**.** *Let $C$ be a set of channel names, $M_c$ the set of allowed messages for a channel $c \in C$ and $M = \bigcup_{c \in C} M_c$. The stream bundle type is then defined as:*

$$C^\Omega := \{sb \in (C \to M^\omega) \mid \forall c \in C.\, sb(c) \in M_c^\omega\}$$

The common variable name for stream bundles is *sb*. In this paper, we only require two operators over bundles. First, we define a method to combine two bundles whose channel sets do not overlap.

**Definition 3** (Bundle Union)**.** *Given two disjoint channel sets $C_1$ and $C_2$ and two bundles $sb_1 \in C_1^\Omega$ and $sb_2 \in C_2^\Omega$. The union of stream-bundles is defined as:*

$$sb_1 \uplus sb_2 \in (C_1 \cup C_2)^\Omega$$
$$sb_1 \uplus sb_2 := \lambda c. \begin{cases} sb_1(c) & c \in C_1 \\ sb_2(c) & c \in C_2 \end{cases}$$

The domain of bundles can be restricted to a smaller channel set.

**Definition 4** (Bundle Restriction)**.** *For $C_1 \subseteq C_2$ the bundle $sb \in C_2^\Omega$ can be restricted to only the channels in $C_1$ with the suffix-operator $\cdot|_{C_1}$:*

$$sb|_{C_1} \in {C_1}^\Omega$$
$$sb|_{C_1} := \lambda c.sb(c)$$

With the bundle-datatype, we can now define the datatypes for components. First, we define deterministic components where each input bundle is mapped to exactly one output bundle. Additionally, these components must be Scott-continuous (Broy & Stølen 2001; Abramsky & Jung 1995) to ensure realizability in practice.

**Definition 5** (stream processing function (SPF) based on total function (Bürger et al. 2020))**.** *Let $C$ be the set of all channels and let $I, O \subseteq C$. We define the SPF type $SPF_{I,O}$ that includes all continuous functions with input channels $I$ and output channels $O$ via*

$$SPF_{I,O} := \{f \in (I^\Omega \to O^\Omega) \mid \mathit{continuous}(f)\}$$

We define the composition operator $\otimes$ to connect components into systems of components. Identical channels are automatically connected.

**Definition 6** (Composition of SPF)**.** *Let $f \in SPF_{I_f,O_f}$ and $g \in SPF_{I_g,O_g}$ with disjoint output channels $O_f$ and $O_g$. For the composition of deterministic components $\otimes$, the following properties hold:*

$$(f \otimes g) \in SPF_{(I_f \cup I_g)\setminus(O_f \cup O_g),(O_f \cup O_g)}$$
$$(f \otimes g)|_{O_f} = \lambda sb.f((sb \uplus ((f \otimes g)sb))|_{I_f})$$
$$(f \otimes g) = (g \otimes f)$$

The composition operator is defined by a least-fixed point whose existence is guaranteed by Kleene's fixed-point theorem (Kleene 1952). Consequently, it can be recursively computed by a fixed-point iteration which is defined as follows.

$$fixiter(f,g,sb)_i \in (O_f \cup O_g)^\Omega$$
$$fixiter(f,g,sb)_i := \begin{cases} \lambda c.\epsilon & i = 0 \\ f(sb \uplus fixiter(f,g,sb)_{i-1}) \uplus \\ g(sb \uplus fixiter(f,g,sb)_{i-1}) & i > 0 \end{cases}$$

Similar to the bundle restriction we can restrict stream processing functions. This is achieved by essentially setting some of the input channels to contain no information. We use this in theorem 3.

**Definition 7** (Restriction of SPF)**.** *Let $C$ be the set of all channels and let $I, I', O \subseteq C$ such that $I \subseteq I'$. For $f \in SPF_{I',O}$ we define*

$$f|_I \colon I^\Omega \to O^\Omega \colon sb \mapsto f(sb \uplus \varepsilon)$$

*where $\varepsilon \in (I' \setminus I)^\Omega$ is the bundle of empty streams.*

A set of deterministic components is used to encode under-specified components. This can be used in early development steps to precisely describe the behavior that is known and leave unknown behavior under-specified.

**Definition 8** (stream processing specification (SPS) (Bürger et al. 2020))**.** *Let $C$ be the set of all possible channels and $I, O \subseteq C$. We define the SPS type $SPS_{I,O}$ with input channels $I$ and output channels $O$ as shown below:*

$$SPS_{I,O} := \mathbb{P}(SPF_{I,O})$$

We call an SPS *consistent* when there exists at least one SPF in the set.

**Definition 9** (Composition of SPS)**.** *Let $F \in SPS_{I_F,O_F}$ and $G \in SPS_{I_G,O_G}$ with disjoint output channels $O_F$ and $O_G$. The composition of non-deterministic components $\bigotimes$ is defined as:*

$$F \bigotimes G \in SPS_{(I_F \cup I_G)\setminus(O_F \cup O_G),(O_F \cup O_G)}$$
$$F \bigotimes G := \{f \otimes g \mid f \in F \wedge g \in G\}$$

Similar to the bundle restriction from definition 4, the channel sets of components can be restricted. This operation can be used to include unused input channels and to hide output channels of a component.

**Definition 10** (Restriction of SPS)**.** *Let channel sets $I \subseteq I'$, $O' \subseteq O$, and the component $F \in SPS_{I,O}$.*

$$[F]^{I'}_{O'} \in SPS_{I',O'}$$
$$[F]^{I'}_{O'} := \{(\lambda sb.(f(sb|_I))|_{O'}) \mid f \in F\}$$

We omit the restriction of an unchanged bundle. For example, in the case of an unchanged output bundle $O' = O$, we only write $[F]^{I'}$ instead of $[F]^{I'}_{O'}$, and in the case of an unchanged input bundle $I = I'$, we write $[F]_{O'}$ instead of $[F]^{I'}_{O'}$.

FOCUS uses *refinement* to relate abstract components with more concrete components. Since the non-deterministic components are represented as sets of deterministic components, we can use the subset relation as a refinement relation. When $F' \subseteq F$, we say that component $F'$ *refines* component $F$. As a consequence, properties of component $F$ directly hold for the refined component $F'$. Consider, for example, property $P$ that holds over all SPF in $F$. Then this property automatically holds over all SPF in $F'$:

$$F' \subseteq F \longrightarrow \forall SPF \in F.\, P(SPF) \longrightarrow \forall SPF' \in F'.\, P(SPF')$$

In the special case of set equivalence $F' = F$, we say that component $F'$ *refactors* component $F$. Consequently, a refactoring consists of two distinct, mutually inverse refinements. Refactoring a component can modify the internal specification of a component while preserving its behavior.

Sometimes, refining components is only possible if a specific condition for the input bundle is fulfilled. For example, a refinement may only be valid if a pair of input channels always contain the same stream. In such cases, refinement cannot be used. Therefore, we define a more general form of refinement, namely refinement under input-invariant that utilizes an invariant constraining the input bundles.

**Definition 11** (Refinement under Input-Invariant)**.** *For two components $F', F \in SPS_{I,O}$ refinement under invariant $J \subseteq I^\Omega$ is defined as:*

$$F' \subseteq_J F :\longleftrightarrow \forall SPF' \in F'.\exists SPF \in F.\forall sb \in J.\ SPF'\ sb = SPF\ sb$$

For the special case of refactoring under invariant, write $F' =_J F$ as an abbreviation for $F' \subseteq_J F \wedge F \subseteq_J F'$.

The exposition of the conditions we meet when discussing the refactoring pattern for adding or deleting input channels benefits from conceptualizing a specific property. Intuitively, for a channel to be deleted without changing the components' behavior, this channel must not be relevant to any of the components' outputs. This property can be formalized as follows.

**Definition 12** (Effect)**.** *Let $F \in SPS_{I,O}$ be a stream processing specification where $I, O \subseteq C$ for a set of possible channels $C$. We say that channel $i \in I$ has an effect on output channels $O' \subseteq O$ if there exists $f \in F$ and there exist two input bundles $s, t \in I^\Omega$ which are equal on all channels except i, i.e., $s \setminus i = t \setminus i$, and the output on ports $O'$ differs, that is, $f(s)|_{O'} \neq f(t)|_{O'}$.*

*We denote input channel $i$ affecting output channel $o$ by $i \rightsquigarrow o$.*

Consider for example $f \in SPS_{I,O}$ where $I = \{i_1, i_2\}$ and $O = \{o\}$. Let $f(sb) = sb(i_1) \wedge sb(i_2)$ calculate the element wise or of two binary streams. Then both inputs have an effect on the output channel. In fact, any two different streams on $i_1$ yield different output streams when the stream on $i_2$ is fixed and vice versa.

## 4. Case-Study

This section gives an overview of the considered system and the refactoring objective. The original system is presented in Figure 2a and consists of two strongly underspecified components `Admin` and `AccessControl`. For the architecture refactoring process, several requirements must be ensured throughout the development:

1. the system behavior shall not change
2. on this abstract level, the encryption should be configurable
3. the refactoring must be formally verified

The first requirement ensures the original system's functionality is preserved throughout the refactoring process. The second requirement facilitates freedom regarding final encryption scheme choices. The last requirement demands valid mathematical and formal evidence, e.g., for certification. From a mathematical view, this removes the need to certify the refactored system again. This could prove useful in the aviation sector, where adding secure communication to already existing airplanes could entail verifying correctness again.

In the demonstrating example, `AccessControl`'s inputs consist of `Control` messages and `Request` messages and outputs `Control` messages. An input `Control` message sets the access control rights for users. Furthermore, the `AccessControl` component can be requested to output the access control rights. `Admin` accepts `Control` messages and outputs them according to the rights towards the `AccessControl` component. This is represented by an underspecified filtering by the `Admin` component. Thus, users can manage user roles and rights through the `Admin` component, if they are eligible, and users can receive the roles and rights by sending requests to the `AccessControl` component. To prevent malicious users from faking admin control messages by hijacking the network connection, access should be secured by encrypting the communication channel between the components. At the end of the refactoring, the `EncryptedSystem` is composed of two subcompositions `EncryptedAdmin` and `EncryptedAccessControl` as depicted in Figure 2g. `EncryptedAdmin` sequentially composes `Admin` with an `Encrypt` component, which takes a stream of `Control` messages as input and outputs a stream of `EncryptedControl` messages that also acts as the output for the composition. `EncryptedAccessControl` sequentially composes a `Decrypt` component and `AccessControl`. The former takes a stream of `EncryptedControl` messages as input for the composition. It outputs a stream of `Control` messages that act as input for `AccessControl`.

The `Encrypt` applies the encryption function to map the `Control` input to encrypted messages that are transmitted to

`Decrypt`. Similarly, `Decrypt` uses its decryption function to map the `EncryptedControl` messages to `Control` messages for the access control. To allow usage of any encryption scheme, we must also assume that the decryption function is a left inverse of the encryption function. This is enough to allow formal verification of behavior equivalence over the systems. There are no concrete specification message datatypes to allow the use of this refactored architecture as a reference architecture that can be implemented in different scenarios with different encryption algorithms and systems. Formally abstracting away from the `Admin` and `AccessControl` behavior allows a formally verified secure channel implementation into an arbitrary system. As a result, this demonstration not only verifies this specific case but also results in a verified reference model.

The proof of equality between the refactored and original system is divided into six smaller proofs to showcase the application of the different development patterns. Five intermediate systems are modeled for this, and each proof step represents a transformation operation.

The final architecture in Figure 2g is iteratively developed. Each development step is presented in section 6. As a result, an encryption component and a decryption component are added to ensure secure communication while maintaining behavioral equivalence throughout the process.

## 5. Model-driven Development Process

Combining the model-driven development and refactoring of the system with a transformer to generate the case study in Isabelle allows for using simple and abstract models in an architecture description language. Here, we use a SysMLv2 (SST 2023) profile that perfectly fits the architecture descriptions and already supports generating Isabelle code from model artifacts (Kausch, Michael, et al. 2021). The refactoring of a system is achieved by first changing the SysML models of the system (1), creating Isabelle theories for the original and the changed system (2), and finally verifying the refactoring (3). After changing the SysML models manually (1), a developer can use a generator to create system encodings and proof obligations in the theorem prover Isabelle automatically (2). Proving the refactoring in Isabelle (3) is currently done manually, but automation possibilities exist and are discussed in section 7. Because of the mathematical foundation FOCUS, the refinement of a component leads directly to the refinement of an architecture. Each component of a system can be iteratively developed further and refined through the cycles of a typical agile development process without compromising already achieved goals. The model-driven refactoring process facilitates agility further, by allowing for changing system architectures throughout the whole development process or even in production, without the need for additional (integration) testing.

The running example consists of models for part definitions, constraints, and compositions. These correspond to components and systems interfaces, behavior, and internal architecture. An example of the original system model is given in Listing 1. The original system is defined as a part/component (Line 1). Furthermore, the input and output interface is defined using ports (Lines 2-4). In this example, a '∼' prefix denotes an input port definition. Afterward, subcomponents are instantiated as parts (Lines 6, 7). The referenced subcomponents `Admin` and `AccessControl` have their own interfaces and are modularly defined in other models. At the end of the model, the internal system architecture is modeled by connecting the ports of the subcomponents and the system (Lines 9-12).

```
1  part def OriginalSystem {
2    port input: ~Requests;
3    port control: ~Controls;
4    port output: Controls;
5
6    part admin: Admin;
7    part access_control: AccessControl;
8
9    connect control to admin.control;
10   connect input to access_control.input;
11   connect admin.config to access_control.config;
12   connect access_control.output to output;
13 }
```

**Listing 1** SysMLv2 model of original system, graphical representation in Figure 2a.

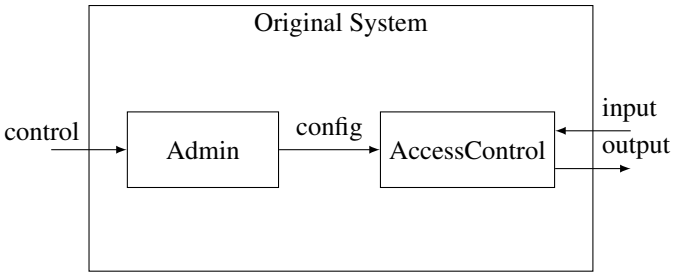
The communication between components specifies the behavior of `OriginalSystem`. The behavior of each sub-component is defined by constraints in a requirement block as shown in listing 2. The `AccessControl` constraint (Lines 7-8) enforces that only received control messages are sent over the output channel.

```
1 part def AccessControl {
2  port input: ~Requests;
3  port config: ~Controls;
4  port output: Controls;
5
6  satisfy requirement OutputRequirements {
7   require constraint OutputSubsetControl {
8    output.data.values() ⊆ config.data.values()
9 }}}
```

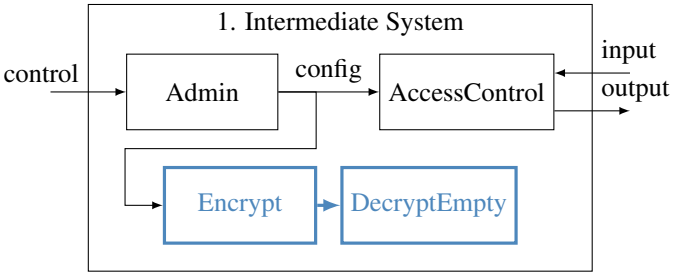
**Listing 2** SysMLv2 model of the access control component

The models of all components and intermediate representations are then transformed to Isabelle by mapping the model artifacts to the corresponding FOCUS concepts using the semantics for behavior descriptions in SysMLv2 defined in (Kausch, Pfeiffer, et al. 2021). Then, it is possible to specify and reason about the preservation of behavior within the theorem prover.

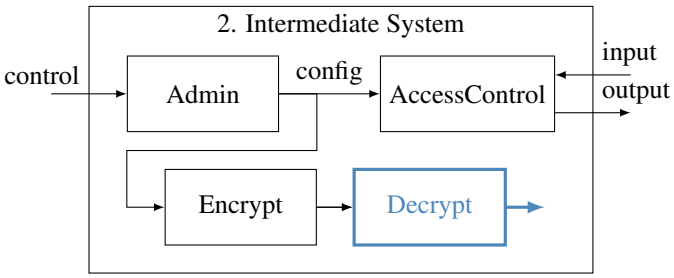
Scalability for larger cases is achieved by specifying meaningful hierarchical levels. In this case study, listing 1 defines one hierarchical level that is later further analyzed. It is irrelevant if sub-components are later decomposed over additional hierarchical levels since each decomposition itself can be analyzed. Thus, even though the original system has only two components in our abstract architecture, the `Admin` component can be specified by 1000 additional components over 3 hierarchical levels, where each (sub)-architecture has only 10 components and each decomposition is analyzable. Building upon the transformed Isabelle models, the architecture refactoring process, including formal development patterns, their application to the running example, and the formal verification thereof, are described in the following section.



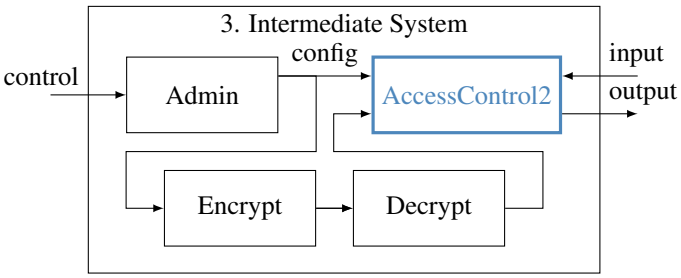

**(a)** Original system architecture.



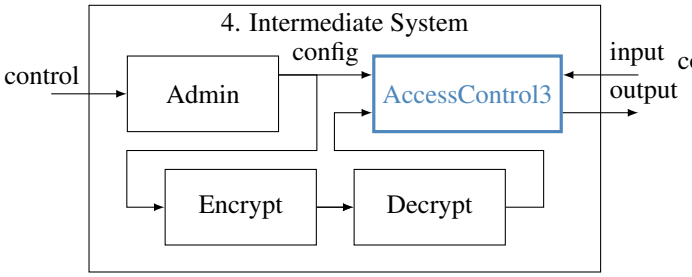

**(b)** Added `Encrypt` and `DecryptEmpty` component.



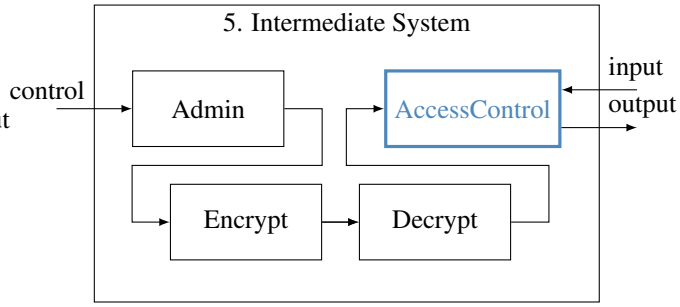

**(c)** Added output channel to `DecryptEmpty` component.



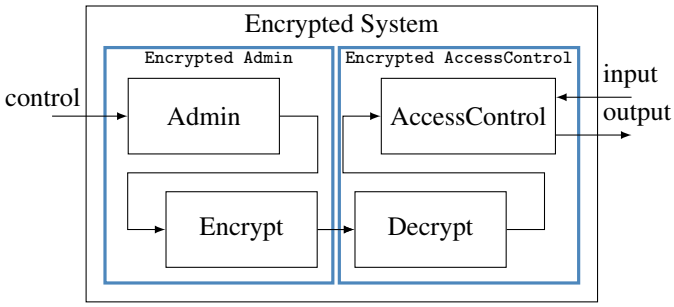

**(d)** Replaced `AccessControl` with `AccessControl2` which additionally accepts `Decrypt`'s output as input.




**(e)** Replaced `AccessControl2` (specified using `Admin`'s output) with `AccessControl3` (specified using `Decrypt`'s output).




**(f)** Replaced `AccessControl3` with `AccessControl` which only accepts `Decrypt`'s output as input.




**(g)** Fold components into two separate subarchitectures.

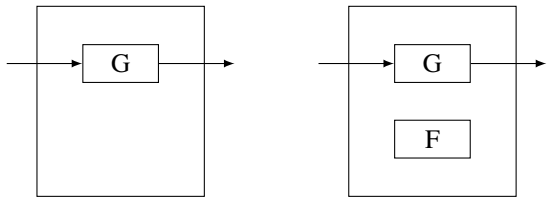
**Figure 2** Overview of the original system architecture (2a), the encrypted system architecture (2g) and all intermediate system architectures in between that are used to formally verify their equivalence using refactoring patterns. The bold and blue border indicates a modification compared to the previous system architecture.

## 6. Architecture Refactoring Patterns

There are many ways to refactor an architecture. For instance, a component's interface is modified, or its internal channel routing is changed. In the following sections, we will introduce five distinct patterns of systematic architecture refactoring. We refer to each refactoring pattern by its corresponding pair of mutually inverse refinements. The theorems presented have been proven using the theorem prover Isabelle in the general case as stated below. Thus, it suffices to prove that the respective assumptions hold for for any specific case to conclude that the respective refinement is valid. We discuss how we achieved this for our case-study for each of the theorems.

### 6.1. Adding/Removing Components

The first refactoring pattern allows components to be added/removed from an architecture. In both cases, it is crucial that the behavior of the component in question does not influence the behavior of the architecture. Assuming that each output channel of the component is not used either as a direct output of the architecture or as an input to another component ensures this. This assumption holds immediately for the special cases considered in (Philipps & Rumpe 1999), where the added components have no output channels. Note, however, that the components' output channels are allowed to contain feedback channels to the component itself. Assumption (2) in Theorem 1 constrains the component to be consistent. This is needed to ensure that removing the component is a valid refinement. Otherwise, removing an inconsistent component from an architecture may result in a consistent architecture, which would be an invalid refinement.




**(a)** Architecture without the additional component.

**(b)** Architecture with the added component.

**Figure 3** Illustration of the adding/removing component refactoring pattern.

**Theorem 1** (Adding/Removing Components)**.** *Let $C$ be the set of all channels. Let $[G_1 \bigotimes \cdots \bigotimes G_n]^I_O \in SPS_{I,O}$ be an architecture for an $n \in \mathbb{N}$, where $I \subseteq C$ is the input channel set and $O \subseteq C$ is the output channel set of the architecture. Furthermore, let $I_G, O_G \subseteq C$ be the input, respectively output channel sets such that $G_1 \bigotimes \cdots \bigotimes G_n \in SPS_{I_G,O_G}$. Additionally, let $F \in SPS_{I_F,O_F}$ be a component with input channels $I_F \subseteq C$ and output channels $O_F \subseteq C$. If*

$$O_F \cap (I_G \cup O) = \emptyset, \quad (1)$$

$$\text{and } F \neq \emptyset \quad (2)$$

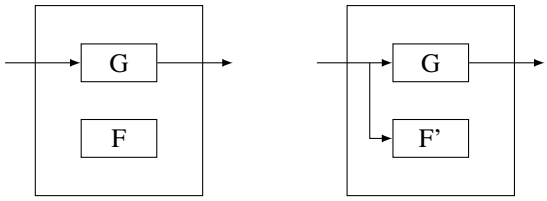
*then*

$$[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O = [G_1 \bigotimes \cdots \bigotimes G_n]^I_O,$$

*that is, the architecture $[G_1 \bigotimes \cdots \bigotimes G_n]^I_O$ is invariant under addition (or removal) of component $F$.*

**Case-Study** In the case-study, this refactoring pattern is applied twice to the original system in Figure 2a to add the `Encrypt` and `DecryptEmpty` components, resulting in the first intermediate system shown in Figure 2b. The correctness of the refactorings is verified by proving the corresponding assumptions. First of all, `Encrypt` and `DecryptEmpty` are shown to be consistent with their specifications. Secondly, it is shown that the output of `Encrypt` is not used anywhere else in the architecture, so `Encrypt` is added. Finally, `DecryptEmpty` is added after proving that `DecryptEmpty`'s output is not used anywhere else in the architecture. Note that `Encrypt` must be added first, as `DecryptEmpty` depends on the output of `Encrypt`. Otherwise, when adding `DecryptEmpty` before `Encrypt`, assumption (2) would be violated as the output of `Encrypt` would be used as input by `DecryptEmpty`. Although `DecryptEmpty` uses `Encrypt`'s output channel, `DecryptEmpty` itself does not affect the architecture behavior. Thus, adding `Encrypt` after adding `DecryptEmpty` is also valid. However, a more thorough effect analysis is required to formally verify this.

### 6.2. Adding/Removing Input Channels




**(a)** Architecture without the additional input channel.

**(b)** Architecture with the added input channel.

**Figure 4** Illustration of the adding/removing input channel refactoring pattern.

The second refactoring pattern concerns the addition or removal of input channels of components within an architecture. Three assumptions are needed to verify this refactoring. Adding or removing input channels of a component replaces the component in the architecture with another component that has the same output signature but an input signature that contains more or fewer channels, respectively. This requirement is captured in assumption (1) of Theorem 2. Assumption (2) requires the added input channel to be either an output of another component or an input of the architecture. Similarly to the first pattern, it is important to ensure that adding or removing the channel in question does not change the behavior of the entire architecture. Therefore, assumption (3) is essential to ensure that the behavior of the replaced component remains unchanged. On one hand, it requires that the component with additional input channels is a refinement of the component with fewer inputs when restricted to the common inputs which means that the former does not introduce any new behavior that depends on the additional inputs. On the other hand, it ensures that the component with fewer input channels is a refinement of the component with additional

inputs which means that the former is capable of all behaviors of the latter when restricted to the common input channels. The assumptions of the theorem below apply both for addition or removal of input channels. Note, that $F$ is always the component with less input channels and $F'$ is always the component with more input channels.

**Theorem 2** (Adding/Removing Input Channels)**.** *Let $C$ be the set of all channels. Let $[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O \in SPS_{I,O}$ be an architecture for an $n \in \mathbb{N}$, where $I \subseteq C$ is the input channel set and $O \subseteq C$ is the output channel set of the architecture. Furthermore, let $I_G, O_G \subseteq C$ be the input, respectively output channel sets such that $G_1 \bigotimes \cdots \bigotimes G_n \in SPS_{I_G,O_G}$. Additionally, let $F \in SPS_{I_F,O_F}$ and $F' \in SPS_{I_{F'},O_F}$ be components with input channels $I_F, I_{F'} \subseteq C$ and output channels $O_F \subseteq C$. If*

$$I_F \subseteq I_{F'} \tag{1}$$
$$I_{F'} \setminus I_F \subseteq I \cup O_F \cup O_G \tag{2}$$
$$[F]^{I_{F'}} = F' \tag{3}$$

*then*

$$[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O = [F' \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O,$$

*that is, the architecture $[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O$ is invariant under addition (or removal) of input channels to the component $F$.*

**Case-Study** This refactoring pattern is applied twice in the case-study. First, it is used to add the output of `Decrypt` as an input to the `AccessControl` component in Figure 2c. The resulting component is named `AccessControl2`, shown in Figure 2d, whose behavior is specified equally to that of `AccessControl`. The second application of this refactoring pattern removes the output channel `Control` of the `Admin` component from the input of `AccessControl3` in Figure 2e. The resulting component is again `AccessControl`, as shown in Figure 2f.

In both cases, the most challenging part of verifying this refactoring is proving the assumption (3), ensuring equal behavior between the replaced components. More precisely, proving

$$F' \subseteq [F]^{I_{F'}}, \tag{4}$$

i.e., showing that no new behavior is introduced by adding input channels, turns out to be challenging. This is because constructing for every $f' \in F'$ the corresponding $f \in F$ that has equal behavior on the common input channels is not straightforward since the behavior of $f'$ may depend on the added input channel. Our approach to this problem in the case-study exploits the fact that the stream on the added channel is always identical to the stream of one of the existing input channels. This invariant holds nicely in our particular example and enables concise proof that property (4) holds. Finding a suitable invariant might not always be possible, however. In subsection 6.4, a more detailed introduction to invariants is given.

In cases where no suitable invariant can be found additional information about $F'$ is required.

To ensure the added channel does not change the component's behavior, the predicate specifying the behavior of $F'$ has to provide information about the inputs on the added channel. A helpful way to conceptualize this is via effects. It suffices if the component's behavior, depending on the previously existing input channels, stays the same, and the added channel has no effect on any output channel. Therefore, if the predicates specifying $F'$ follow from those specifying $F$ as well as the absence of any effects property (4) can be proven more easily. If only the addition of a channel is desired, one may reuse the predicates specifying $F$ with appropriately adapted signatures; that is, they accept a bundle extended by the additional channel but assert nothing about input on this channel. In addition, one can then add a predicate specifying that the added channel has no effect on any output channel, that is, $i \not\rightsquigarrow o$ for all added channels $i$ and all output channels $o$. Note that this is necessary since if the predicate provides no assertion about the added channel $F'$ may include functions for which the added channel exhibits an effect on an output channel. Hence, we have to restrict the component to only those functions that do not exhibit such effects.

This yields the following theorem, which we encoded and proved in Isabelle as well.

**Theorem 3.** *Let $C$ be the set of all channels. Let $[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O \in SPS_{I,O}$ be an architecture for an $n \in \mathbb{N}$, where $I \subseteq C$ is the input channel set and $O \subseteq C$ is the output channel set of the architecture. Furthermore, let $I_G, O_G \subseteq C$ be the input, respectively output channel sets such that $G_1 \bigotimes \cdots \bigotimes G_n \in SPS_{I_G,O_G}$. Additionally, let $F \in SPS_{I_F,O_F}$ and $F' \in SPS_{I_{F'},O_F}$ be components with input channels $I_F, I_{F'} \subseteq C$ and output channels $O_F \subseteq C$. If*

$$I_F \subseteq I_{F'} \tag{1}$$
$$I_{F'} \setminus I_F \subseteq I \cup O_F \cup O_G \tag{2}$$
$$\forall f' \in F' \exists f \in F \colon f'|_{I_F} = f \tag{3}$$
$$\forall i \in I_{F'} \setminus I_F \forall o \in O_F \colon i \not\rightsquigarrow o \tag{4}$$

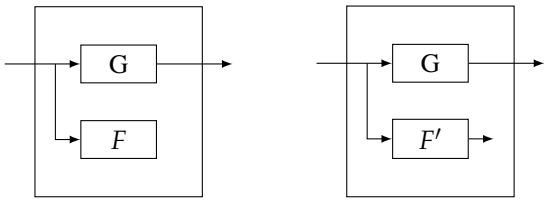
*then*

$$F' \subseteq [F]^{I_{F'}}$$

Here, assumption (3) encodes that $F'$ behaves on common channels equivalent to $F$. It asserts that as long as no information flows on the added channels, $F'$ behaves like $F$. Crucially, this does not mean that $F'$ can not exhibit different behavior when non-trivial input is provided on the added channels. Only assumption (4) guarantees that this is not the case.

### 6.3. Adding/Removing Output Channels

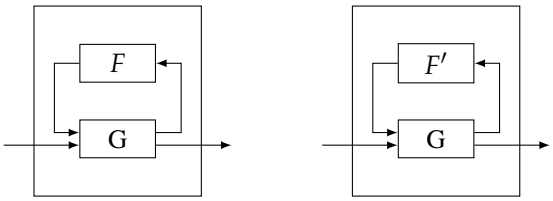
Similar to adding or removing input channels, the next refactoring pattern allows for adding or removing unused output channels of components within an architecture. This is done by replacing the component with another component whose input signature is the same, but whose output signature is modified. To verify this refactoring, three assumptions must be shown. Assumption (1) requires the output channels of the component with fewer output channels to be a subset of the output channels of the other component. Furthermore, assumption (2) ensures that any added output channels do not affect the behavior of the architecture, meaning that they are not used as input for any other component (including the refactored component itself),




**(a)** Architecture without the additional output channel.

**(b)** Architecture with the added output channel.

**Figure 5** Illustration of the adding/removing output channel refactoring pattern.




**(a)** Architecture with component $F$.

**(b)** Architecture with component $F'$.

**Figure 6** Illustration of the refactoring under an invariant pattern.

nor are they directly used as output for the architecture. Assumption (3) constrains the behavior of the components. It ensures that every possible component behavior with fewer outputs is also possible with more output channels when restricting the component to the common output channels, an vice versa. Further, it also ensures that the component with additional output does not exhibit behavior that is not possible in the component with fewer output channels.

**Theorem 4** (Adding/Removing Output Channels)**.** *Let $C$ be the set of all channels. Let $[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O \in SPS_{I,O}$ be an architecture for an $n \in \mathbb{N}$, where $I \subseteq C$ is the input channel set and $O \subseteq C$ is the output channel set of the architecture. Furthermore, let $I_G, O_G \subseteq C$ be the input, respectively output channel sets such that $G_1 \bigotimes \cdots \bigotimes G_n \in SPS_{I_G,O_G}$. Additionally, let $F \in SPS_{I_F,O_F}$ and $F' \in SPS_{I_F,O_{F'}}$ be components with input channels $I_F \subseteq C$ and output channels $O_F, O_{F'} \subseteq C$. If*

$$O_F \subseteq O_{F'} \tag{1}$$

$$(O_{F'} \setminus O_F) \cap (O \cup I_F \cup I_G) = \emptyset \tag{2}$$

$$F = [F']_{O_F} \tag{3}$$

*then*

$$[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O = [F' \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O.$$

*that is, the architecture $[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O$ is invariant under addition (or removal) of output channels to the component $F$.*

**Case-Study** This refactoring pattern was applied to add an output channel to `DecryptEmpty` within the first intermediate system (Figure 2b). Thereby `DecryptEmpty` is replaced with `Decrypt` that uses the same input signature and a different output signature that contains an additional channel as shown in Figure 2c. Since `DecryptEmpty` has no output channels, assumption (1) holds directly. Assumption (2) is also satisfied since the added output channel is not used elsewhere in the architecture as an input for another component or as output for the architecture itself. Since there are no common output channels between `DecryptEmpty` and `Decrypt`, as the former has no output channels, assumption (3) follows directly from the consistency of both components.

### 6.4. Refactoring under Invariant

The pattern of refactoring under invariant is based on refinement under invariant, which is a generalized form of behavioral refinement. It enables replacing a component within an architecture with another component of the same input and output signature. To ensure the replacement is valid, an invariant that encodes a condition on the input of the components has to be satisfied in the context of the architecture for all possible inputs. For example, an invariant might require two distinct input channels always to contain the same stream.

To verify this refactoring pattern, two assumptions have to be shown. Assumption (1) requires that the replaced components exhibit equal behavior under the given invariant. In contrast, assumption (2) ensures that the invariant holds in the context of every possible input of the architecture. More precisely, for every possible input of the architecture, the invariant must hold at every step of the corresponding fixed-point iteration of the composition.

**Theorem 5** (Refactoring under Invariant)**.** *Let $C$ be the set of all channels. Let $[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O \in SPS_{I,O}$ be an architecture for an $n \in \mathbb{N}$, where $I \subseteq C$ is the input channel set and $O \subseteq C$ is the output channel set of the architecture. Furthermore, let $I_G, O_G \subseteq C$ be the input, respectively output channel sets such that $G_1 \bigotimes \cdots \bigotimes G_n \in SPS_{I_G,O_G}$. Additionally, let $F \in SPS_{I_F,O_F}$ and $F' \in SPS_{I_F,O_F}$ be components with input channels $I_F \subseteq C$ and output channels $O_F \subseteq C$. Further, let $J \subseteq {I_F}^\Omega$ be an invariant. If*

$$F =_J F' \tag{1}$$

$$\begin{aligned} &\forall f \in F \cup F', \\ &\quad g \in G,\ i \in \mathbb{N}, \\ &\quad sb \in ((I_F \cup I_G) \setminus (O_F \cup O_G))^\Omega. \\ &\quad (sb \uplus fixiter(f, g, sb)_i)|_{I_F} \in J \end{aligned} \tag{2}$$

*then*

$$[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O = [F' \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O.$$

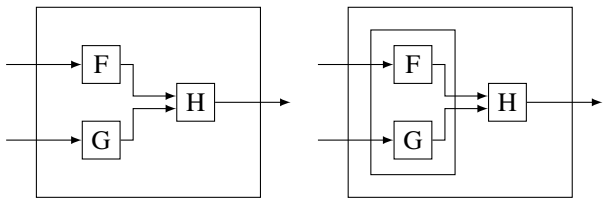
*that is, the architecture $[F \bigotimes G_1 \bigotimes \cdots \bigotimes G_n]^I_O$ is invariant under refactoring under invariant $J$ on the component $F$.*

**Case-Study** In the case-study, refactoring under invariant was used to verify the replacement of `AccessControl2` in the third intermediate system (Figure 2d) by `AccessControl3`, resulting in the fourth intermediate system shown in Figure 2e. While `AccessControl2` is specified using the output of the `Admin`

component, `AccessControl3` is specified by an equal specification, except that it uses the output of `Decrypt` instead of `Admin`. The chosen invariant requires that the streams on the output channel `Control` of `Admin` and `Decrypt` are equal.

The correctness of this refactoring was verified by proving both assumptions. Assumption (1) follows from the chosen invariant in combination with `AccessControl2`'s and `AccessControl3`'s specifications. The proof of assumption (2) one uses the assumption that `Decrypt` is left inverse to `Encrypt`, resulting in `Decrypt`'s output stream being equal to `Admin`'s output stream. Consequently, the chosen invariant is satisfied in this context.

### 6.5. Un-/Folding of Sub-Architectures




**(a)** Architecture with components $F$, $G$ and $H$.

**(b)** Architecture with components $F$ and $G$ folded into a sub-architecture.

**Figure 7** Illustration of the folding refactoring pattern.

The last refactoring pattern enables folding and unfolding of sub-architectures within an architecture. A sub-architecture is a component that is specified by another architecture. For example, sub-architectures are used to separate groups of closely related components. Unfolding a sub-architecture means replacing the component specified by the sub-architecture with the components contained in the sub-architecture. Conversely, folding a component means replacing a set of components in the architecture with a single component specified by a sub-architecture consisting of the replaced components.

Two assumptions, which only constrain the input and output signature of the sub-architecture are necessary to verify this refactoring. Assumption (1) ensures the input channels of the sub-architecture at least contain all non-feedback input channels of the components of the sub-architecture. Additionally, it limits it to contain at most all channels which are input channels of the architecture or output channels of components from the remaining architecture. Consequently, the input of the sub-architecture may contain channels that are not used by any of its components. Similarly, assumption (2) constrains the output channels of the sub-architecture to contain at least all output channels of components from the sub-architecture that are used as either output for the architecture or input for another component in the remaining architecture. As a result, it allows the sub-architecture to conceal internal output channels that are not used outside of the sub-architecture. Assumption (2) also ensures that the sub-architecture outputs no more than the output channels of its contained components.

**Theorem 6** (Un-/Folding Sub-Architectures)**.** *Let $C$ be the set of all channels. Let $[F_1 \bigotimes \cdots \bigotimes F_n \bigotimes G_1 \bigotimes \cdots \bigotimes G_m]^I_O \in SPS_{I,O}$ be an architecture for $n, m \in \mathbb{N}$, where $I \subseteq C$ is the input channel set and $O \subseteq C$ is the output channel set of the architecture. Furthermore, let $I_F, O_F \subseteq C$ be the input, respectively output channel sets such that $F_1 \bigotimes \cdots \bigotimes F_n \in SPS_{I_F,O_F}$. Similarly, let $I_G, O_G \subseteq C$ be the input, respectively output channel sets such that $G_1 \bigotimes \cdots \bigotimes G_m \in SPS_{I_G,O_G}$. Additionally, let $I', O' \subseteq C$ be the input, respectively output channel sets of the sub-architecture. If*

$$I_F \setminus O_F \subseteq I' \subseteq I \cup O_G \tag{1}$$

$$(O \cup I_G) \cap O_F \subseteq O' \subseteq O_F \tag{2}$$

*then*

$$[F_1 \bigotimes \cdots \bigotimes F_n \bigotimes G_1 \bigotimes \cdots \bigotimes G_m]^I_O = [[F_1 \bigotimes \cdots \bigotimes F_n]^{I'}_{O'} \bigotimes G_1 \bigotimes \cdots \bigotimes G_m]^I_O$$

*that is, the architecture $[F_1 \bigotimes \cdots \bigotimes F_n \bigotimes G_1 \bigotimes \cdots \bigotimes G_m]^I_O$ is invariant under folding (or unfolding) on the sub-architecture $[F_1 \bigotimes \cdots \bigotimes F_n]^{I'}_{O'}$.*

**Case-Study** This refactoring pattern is applied twice in the case-study to fold `Admin` and `Encrypt` as well as `Decrypt` and `AccessControl` in the fifth intermediate system architecture (Figure 2f), into dedicated sub-architectures named `EncryptedAdmin` and `EncryptedACcessControl` (Figure 2g), respectively. Both sub-architectures group closely related components while hiding internal channels, resulting in only the `EncryptedControl` channel being exposed between them. Again, the correctness of these refactorings was verified by proving the corresponding assumptions. In the case of `EncryptedAdmin`, for instance, it was proven that the input of `EncryptedAdmin` includes at least the input channel of the `Admin` component. It is worth noting that the input of the `Encrypt` component is not included since it is already internally connected to the output channel of the `Admin` component. Additionally, it was shown that the output of `EncryptedAdmin` only contains output channels from either `Admin` or `Encrypt`. Finally, it was proven that the output channel of `Admin`, which is the only hidden channel, is not used in any other part of the outer architecture. The proof for unfolding the `EncryptedAccessControl` sub-architecture works similarly.

## 7. Conclusion

We discussed the five architecture refactoring patterns presented in (Philipps & Rumpe 1999) and demonstrated their application in an example where a secure communication channel is added to an architecture by inserting two components that encrypt and decrypt the communication. Then these added components are folded. For each of these patterns, we demonstrated which conditions suffice to conclude that the refactored architecture is equivalent to the original one by proving the presented theorems in Isabelle. Our work's main contribution is an encoding of these refactoring patterns in the theorem prover Isabelle as well as exemplifying its applicability. Crucially, each pattern has

been formally verified through our work. Our encoding consists of both proof schemes and various lemmata about the pattern's properties. These properties are generalized and are independent of the particular example for which they were initially conceived, enabling them to be used for other examples as well. Currently, the specific proofs required to verify the particular example discussed here are, for the most part, hand-written.

However, our analysis indicates that automating the verification process required for adding and removing components, as well as for folding and unfolding sub-architectures, should be feasible. Full automation might not be feasible for the other refactoring patterns. For instance, when refactoring under an invariant, it is necessary to have a valid invariant.

**Lessons-learned and Future Work** Finding invariants such as those required for the refactoring pattern in subsection 6.4 automatically is difficult. By using advanced proving tools like sledgehammer (Böhme & Nipkow 2010) or PSL (Nagashima & Kumar 2017) the burden of manual proving each step can be reduced. But automatically finding the correct invariant for a proof is only possible in special cases. The same holds for adding and removing input channels for which either an invariant needs to be found, or the effects of the added or removed channel have to be known. Encoding effects in Isabelle and deducing effect properties from given predicates specifying the behavior of a component is actively investigated. Similarly, using effects as additional prescriptive properties to specify component behavior is future work.

An effect specification language, which is under active development, could be helpful to simplify such a specification. For composition, we restricted the composite to only allow behavior already exhibited by its components. Instead, we could have used a composition operator without this restriction, adding all possible behaviors adhering to the components' specifications, including those with different signatures. This is helpful for refinement proofs. However, it renders reasoning about effects much more difficult as this operator adds functions with differing signatures and effects across channels that do not appear with the other. Examination of different composition operators regarding effect analyses is ongoing work.

In Figure 2, we provide a complete model for each refactoring step. This approach increases the modeling effort significantly. Instead, we could use a delta model that only models the difference from an original model to reduce this complexity or streamline the refactoring process by not explicitly modeling all intermediate systems.

The refactoring patterns discussed in this paper can be considered atomic because they cannot be partitioned further. A larger redesign of a system can be achieved by applying a sequence of these atomic patterns, e.g. fig. 2. Such a sequence of atomic refactoring patterns can be understood as a refactoring pattern for a specific purpose, such as inserting encrypted communication as in our case-study. In practice, the starting point of the refactoring might be the final refactored model only. Dividing the refactoring into the atomic patterns is akin to decomposing a high-level function to assembler code. Note, that this decomposition is not necessarily unique. However, it suffices to show behavior preservation for one sequence of atomic refactoring steps to formally verify the overall refactoring. Instead of doing this manually, it could be beneficial to automatically decompose the final refactoring into atomic steps that can be verified automatically if possible, such that manual intervention is only required for those steps for which an automated proof is not feasible.


### Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 499241390 & 250902306 and German Federal Ministry for Economic Affairs and Climate Action, AMoBaCoD-Project (Grant No. 20X2201C).


## References


Abadi, M., & Lamport, L. (1994). Open Systems in TLA. In J. Anderson, D. Peleg, & E. Borowsky (Eds.), *Proceedings of the thirteenth annual acm symposium on principles of distributed computing - podc '94* (pp. 81–90). New York, New York, USA: ACM Press.

Abramsky, S., & Jung, A. (1995). Domain theory. In *Handbook of logic in computer science (vol. 3): Semantic structures* (p. 1–168). USA: Oxford University Press, Inc.

Barendregt, H. P., Dekkers, W., & Statman, R. (2013). *Lambda calculus with types*. Cambridge University Press.

Bauer, F. L., & Woessner, H. (1982). *Algorithmic language and program development.* Springer-Verlag.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, *32*(10), 70–77.

Bertot, Y., & Castéran, P. (2013). *Interactive theorem proving and program development: Coq'art: the calculus of inductive constructions*. Springer Science & Business Media.

Böhme, S., & Nipkow, T. (2010). Sledgehammer: Judgement Day. In *Automated reasoning: 5th international joint conference, ijcar 2010, edinburgh, uk, july 16-19, 2010. proceedings 5* (pp. 107–121).

Broy, M. (2023). Specification and verification of concurrent systems by causality and realizability. *Theoretical Computer Science*, *974*, 114106.

Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T. F., & Weber, R. (1992). *The Design of Distributed Systems – An Introduction to* FOCUS. Munich, Germany: Institut für Informatik, Technische Universität München.

Broy, M., & Stølen, K. (2001). *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg.

Bürger, J. C., Kausch, H., Raco, D., Ringert, J. O., Rumpe, B., Stüber, S., & Wiartalla, M. (2020). *Towards an Isabelle Theory for distributed, interactive systems - the untimed case*. Shaker Verlag.

Butting, A., Kautz, O., Rumpe, B., & Wortmann, A. (2017, April). Semantic Differencing for Message-Driven Component & Connector Architectures. In *International conference on software architecture (icsa'17)* (p. 145-154). IEEE.

Buzzard, K., Commelin, J., & Massot, P. (2020). Formalising perfectoid spaces. In (p. 299–312). New York, NY,

USA: Association for Computing Machinery. doi: 10.1145/3372885.3373830

Dahl, O.-J., Dijkstra, E. W., & Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

de Moura, L., Kong, S., Avigad, J., Van Doorn, F., & von Raumer, J. (2015). The Lean theorem prover (system description). In *Automated deduction-cade-25: 25th international conference on automated deduction* (Vol. 25, pp. 378–388).

Dijkstra, E. W. (1972, oct). The humble programmer. *Commun. ACM*, *15*(10), 859–866.

Drave, I., Eikermann, R., Kautz, O., & Rumpe, B. (2019, February). Semantic Differencing of Statecharts for Object-oriented Systems. In S. Hammoudi, L. F. Pires, & B. Selić (Eds.), *Proceedings of the 7th international conference on model-driven engineering and software development (modelsward'19)* (p. 274-282). SciTePress.

Drave, I., Kautz, O., Michael, J., & Rumpe, B. (2019, September). Semantic Evolution Analysis of Feature Models. In T. Berger et al. (Eds.), *International systems and software product line conference (splc'19)* (p. 245-255). ACM.

Fowler, M. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Haber, A., Ringert, J. O., & Rumpe, B. (2012, February). *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems* (Technical Report No. AIB-2012-03). Aachen, Germany: RWTH Aachen University.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231-274. doi: 10.1016/0167-6423(87)90035-9

Harel, D., & Rumpe, B. (2004, October). Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, *37*(10), 64-72.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576–580.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Englewood Cliffs, N.J.: Prentice Hall International.

Kausch, H., Michael, J., Pfeiffer, M., Raco, D., Rumpe, B., & Schweiger, A. (2021, November). Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace europe conference 2021 (aec 2021)*. Council of European Aerospace Societies (CEAS).

Kausch, H., Pfeiffer, M., Raco, D., Rath, A., Rumpe, B., & Schweiger, A. (2023, February). A Theory for Event-Driven Specifications Using Focus and MontiArc on the Example of a Data Link Uplink Feed System. In I. Groher & T. Vogel (Eds.), *Software engineering 2023 workshops* (p. 169-188). Gesellschaft für Informatik e.V.

Kausch, H., Pfeiffer, M., Raco, D., & Rumpe, B. (2020, February). An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems. In R. Hebig & R. Heinrich (Eds.), *Combined proceedings of the workshops at software engineering 2020* (Vol. 2581). CEUR Workshop Proceedings.

Kausch, H., Pfeiffer, M., Raco, D., & Rumpe, B. (2021, February). Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams. In S. Götz, L. Linsbauer, I. Schaefer, & A. Wortmann (Eds.), *Proceedings of the software engineering 2021 satellite events* (Vol. 2814). CEUR.

Kautz, O. (2021). *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Shaker Verlag.

Kautz, O., & Rumpe, B. (2018, October). Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In *Proceedings of models 2018. workshop me.*

Kautz, O., Rumpe, B., & Wortmann, A. (2020, April). Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering Journal*, *27*, 119-151.

Kleene, S. C. (1952). *Introduction to metamathematics*. Groningen: P. Noordhoff N.V.

Knüppel, A., Runge, T., & Schaefer, I. (2020). Scaling correctness-by-construction. In *Leveraging applications of formal methods, verification and validation: Verification principles: 9th international symposium on leveraging applications of formal methods, isola 2020, rhodes, greece, october 20–30, 2020, proceedings, part i 9* (pp. 187–207).

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

Kourie, D. G., & Watson, B. W. (2012). *The correctness-by-construction approach to programming* (Vol. 264). Springer.

Koziolek, H., & Reussner, R. (2008). A model transformation from the palladio component model to layered queueing networks. In *Spec international performance evaluation workshop* (pp. 58–78).

Kriebel, S., Raco, D., Rumpe, B., & Stüber, S. (2019, February). Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In S. Krusche et al. (Eds.), *Proceedings of the workshops of the software engineering conference. workshop on avionics systems and software engineering (aviose'19)* (Vol. 2308, p. 87-94). CEUR Workshop Proceedings.

Lee, E. (2016, 11). Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems*, *1*, 1-26.

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning* (pp. 348–370).

Maoz, S., Ringert, J., & Rumpe, B. (2012, November). An Interim Summary on Semantic Model Differencing. *Softwaretechnik-Trends*, *32*. doi: 10.1007/BF03323524

Maoz, S., Ringert, J. O., & Rumpe, B. (2010). A Manifesto for Semantic Model Differencing. In *Proceedings int. workshop on models and evolution (me'10)* (p. 194-203). Springer.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011a). ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on foundations of software engineering (esec/fse '11)* (p. 179-189). ACM.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011b). CDDiff:

Semantic Differencing for Class Diagrams. In M. Mezini (Ed.), *Ecoop 2011 - object-oriented programming* (p. 230-254). Springer Berlin Heidelberg.
Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, *30*(2), 126–139.
Milner, R. (1982). *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer-Verlag.
Milner, R. (1999). *Communicating and mobile systems: the pi calculus*. Cambridge university press.
Murray, T., & Lowe, G. (2009, 09). On Refinement-Closed Security Properties and Nondeterministic Compositions. *Electr. Notes Theor. Comput. Sci.*, *250*, 49-68.
Nagashima, Y., & Kumar, R. (2017). A proof strategy language and proof script generation for isabelle/hol. In *Automated deduction–cade 26: 26th international conference on automated deduction, gothenburg, sweden, august 6–11, 2017, proceedings* (pp. 528–545).
Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign.
Owre, S., Rushby, J. M., & Shankar, N. (1992). Pvs: A prototype verification system. In *International conference on automated deduction* (pp. 748–752).
Philipps, J., & Rumpe, B. (1997). Refinement of Information Flow Architectures. In M. Hinchey (Ed.), *Icfem'97 proceedings*. Hiroshima, Japan: IEEE CS Press.
Philipps, J., & Rumpe, B. (1999). Refinement of Pipe-and-Filter Architectures. In *Congress on formal methods in the development of computing system (fm'99)* (p. 96-115). Toulouse, France: Springer.
Philipps, J., & Rumpe, B. (2003). Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K. (Ed.), *Practical Foundations of Business and System Specifications* (p. 281-297). Kluwer Academic Publishers.
Reisig, W. (1985). *Petri Nets: An Introduction*. Berlin, Heidelberg: Springer.
Reussner, R. H., Becker, S., Happe, J., Heinrich, R., & Koziolek, A. (2016). *Modeling and simulating software architectures: The palladio approach*. MIT Press.
Ringert, J. O., Rumpe, B., & Stachon, M. (2023, July). On implementing open world semantic differencing for class diagrams. *Journal of Object Technology*, *22*(2), 2:1-14. doi: 10.5381/jot.2023.22.2.a11
Rumpe, B. (2017). *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International.
Scholze, P. (2022). Liquid tensor experiment. *Experimental Mathematics*, *31*(2), 349–354.
SST. (2023). *Sysml v2*. Retrieved 07.08.2023, from https://github.com/Systems-Modeling
University of Cambridge and Technische Universität München. (2023). *Isabelle*. Retrieved 26.07.2023, from https://isabelle.in.tum.de/
University of Edinburgh and Technische Universität München. (n.d.). *Archive of Formal Proofs*. Retrieved from https://www.isa-afp.org/
Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, *14*(4), 221–227.

## About the authors

**Lars Fischer** is a former research assistant at the Chair of Software Engineering at RWTH Aachen University. His research interests covered formal verification in the theorem prover Isabelle.

**Hendrik Kausch** is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research interests cover model semantics, refinement, and formal analyses and model-driven development of distributed systems.You can contact the author at kausch@se-rwth.de or visit https://se-rwth.de/~kausch.

**Bernhard Rumpe** is professor at the RWTH Aachen University leading the Software Engineering department and the Deputy Editor-in-Chief of the International Journal on Software and Systems Modeling. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods, as well as model engineering based on UML/SysML-like notation and domain-specific languages.You can contact the author at rumpe@se-rwth.de or visit https://se-rwth.de/~rumpe.

**Max Stachon** is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research focuses on model semantics, refinement, and difference analysis. You can contact the author at stachon@se-rwth.de or visit https://se-rwth.de/~stachon.

**Sebastian Stüber** is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research interests cover model semantics, refinement, formal verification, and compositional analysis of systems.You can contact the author at stueber@se-rwth.de or visit https://se-rwth.de/~stueber.

**Lucas Wollenhaupt** is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research interests cover analyses of effects and a formal foundation of effects, effect chains, and effect networks in distributed systems.You can contact the author at wollenhaupt@se-rwth.de or visit https://se-rwth.de/~wollenhaupt.