Verifying Patterns of Dynamic Architectures Using Model Checking

Diego Marmsoler Silvio Degenhardt
Technical University of Munich
Germany
{diego.marmsoler | silvio.degenhardt}@tum.de

Architecture patterns capture architectural design experience and provide abstract solutions to recurring architectural design problems. They consist of a description of component types and restrict component connection and activation. Therefore, they guarantee some desired properties for architectures employing the pattern. Unfortunately, most documented patterns do not provide a formal guarantee of whether their specification indeed leads to the desired guarantee. Failure in doing so, however, might lead to wrong architectures, i.e., architectures wrongly supposed to show certain desired properties. Since architectures, in general, have a high impact on the quality of the resulting system and architectural flaws are only difficult, if not to say impossible, to repair, this may lead to badly reparable quality issues in the resulting system. To address this problem, we propose an approach based on model checking to verify pattern specifications w.r.t. their guarantees. In the following we apply the approach to three well-known patterns for dynamic architectures: the Singleton, the Model-View-Controller, and the Broker pattern. Thereby, we discovered ambiguities and missing constraints for all three specifications. Thus, we conclude that verifying patterns of dynamic architectures using model checking is feasible and useful to discover ambiguities and flaws in pattern specifications.

1 Introduction

Architecture patterns capture architectural design experience and are regarded as the "Grand Tool" for designing a software system's architecture [34]. Patterns for dynamic architectures are patterns for architectures in which components may appear and disappear and connections may change over time [36, 15, 8].

Usually, a pattern provides *abstract* solutions to recurring architectural design problems. The solution is usually a specification of component types, connection, and activation constraints. Moreover, it guarantees an overall property for architectures employing them [33, 9]. This property then leads to certain, desired quality aspects of the resulting software system. Consider, for example, the Singleton pattern. It consists of a component type singleton which is supposed to behave as follows: if a new component of this type is required to be activated, this is only done if there is not yet any active component of this type available. The guarantee for the overall architecture is then, that at every point in time at most one component of this type is activated. This guarantee, in turn, leads to reduced resource utilization, since memory usage is minimized.

Unfortunately, for most patterns, there is no formal guarantee that their specification indeed leads to the desired guarantee. For example, there is no guarantee that the specification of the Singleton pattern [16] leads indeed to an architecture in which there is only one active component of that type during the whole execution. Indeed, as shown later on, there are many hidden assumptions about the environment which are left implicit and which are required in order to satisfy the guarantee. Thus, there is actually no guarantee that architectures employing the specification indeed fulfill the desired quality attribute of reduced resource utilization.

To appear in EPTCS.

However, since architects rely on patterns when designing an architecture, this may lead to wrong architecture decisions. Wrong architecture decisions, however, may strongly influence a software systems quality [6, 17] and are only difficult, if not impossible, to repair [17, 23]. In security-related applications, for example, a good architecture may eliminate or mitigate up to 92% of the most dangerous weaknesses [11]. Similarly, two conclusions of the Ariane 5 explosion were that it could have been avoided at design time using correct architecture specifications and a "software architect" position was requested for future projects [12]. To put it in the words of Garlan [17]: "a poor architecture can lead to a disaster for the whole project."

Thus, we propose a 5 step approach based on model checking to verify patterns w.r.t. their claimed guarantees:

1. Review current literature about a pattern.
2. Identify and formalize the interfaces of the involved component types.
3. Model the behavior of component types in terms of abstract state machines.
4. Specify the pattern's guarantee in terms of temporal logic formulae.
5. Verify the guarantee by applying model checking.

To evaluate the approach, we applied it to verify three contemporary patterns for dynamic architectures: The Singleton pattern, the Model-View-Controller pattern, and the Broker pattern.

In the following paper we report on our experience of applying model checking to verify architecture patterns. Thus, the major contributions of this paper are as follows:

1. We describe an approach to formally verify patterns of dynamic architectures.
2. We show feasibility and usability of applying model checking to verify patterns for dynamic architectures.
3. We provide (verified) formalizations of the Model-View-Controller pattern.
4. We describe characteristic (verified) properties for Singleton, MVC and Broker pattern.

The paper is structured as follows: In Sect. 2 we first provide some background information about the techniques used to formalize the pattern specifications as well as of model checking in general. Then, in Sect. 3 we describe the details of the approach, demonstrated by means of a running example. In Sect. 4 we summarize our results obtained for the Singleton, the Model-View-Controller, and the Broker pattern. In Sect. 5 we then report on our experience of applying model checking to pattern verification and critically discuss our approach. Finally, we provide an overview of related work in Sect. 6 and conclude the paper with a summary of major findings, a discussion of possible implications, and points to future work in Sect. 7.

2 Background

In the following we briefly describe the background of our work. Therefore, we first introduce the formalism used to specify patterns for dynamic architectures. Then, we briefly discuss the basic idea behind model checking in general and the NuSMV symbolic model checker specifically.

2.1 Specifying Constraints of Dynamic Architectures

Over the last decades, a series of so-called architecture description languages appeared to support in the formal specification of dynamic software architectures. Examples include the Chemical Abstract Machine [19], Rapide [24], Darwin [25], Wright [3] and its dynamic extension [2], Π-ADL [31], xADL [13], and ACME [18]. Around the same time, some approaches emerged to formalize architectural styles

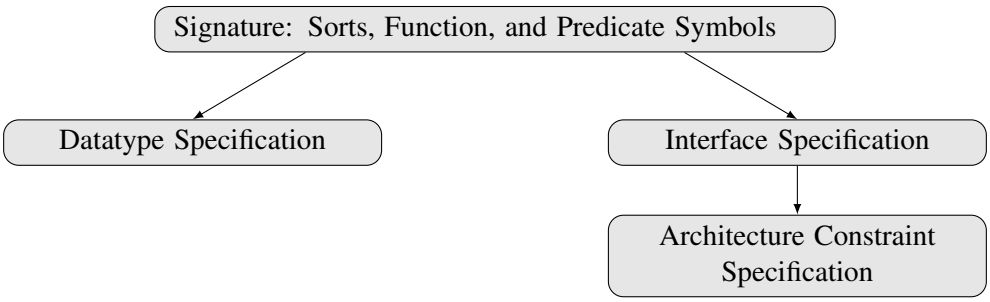

Figure 1: Approach to specify properties of dynamic architectures.

and patterns [1, 30, 32, 22, 5, 29]. Only recently, however, formal models of dynamic architectures emerged [35, 15, 8] and specification techniques for properties of such architectures were developed [27, 28].

In the following, we briefly summarize the major concepts and notations of dynamic architectures found in these works. Moreover, we introduce some techniques used in their specification.

2.1.1 A Model of Dynamic Architectures

In this work, a dynamic architecture is modeled by a set of so-called Configuration Traces (CTs) [28]. A CT, in turn, is a sequence of Architecture Configurations (CNFs) which consist of a set of active components, valuations of their ports with messages, and connections between their ports.

2.1.2 Specification Techniques

For the specification of CTs, we employ algebraic specification techniques, interface specifications, and linear temporal logic [26]. The overall approach is summarized in Fig. 1.

As a first step, a suitable signature is specified to introduce symbols for sets, functions, and predicates. These symbols form the primitive entities of the whole specification process. Datatype specifications and interface specifications as well as architecture constraint specifications are based on these symbols.

Then, datatypes are algebraically specified over the signature [37, 7]. A Datatype Specification (DTS) consists of a set of so-called datatype assertions, built over datatype terms, to assert characteristic properties of the datatype and provide meaning for the symbols introduced in the signature.

Interfaces are also directly specified over the signature. Therefore, a set of ports is typed by sorts of the corresponding signature by means of so-called port specifications. Then, an interface is specified by assigning an interface identifier with three sets of ports: local, input, and output ports. Finally, a set of interface assertions is associated with each interface identifier to specify component types, i.e., interfaces with associated global invariants.

Finally, architecture constraints can be specified by means of configuration trace assertions over the interfaces. Configuration Trace Assertions (CTAs) are a temporal specification technique based on linear temporal logic [26] to specify sets of CTs. They allow the specification of temporal properties over component interfaces. Thereby, port names denote the valuation of a component port in a configuration and can be used as variables in algebraic terms as well. For example, $c.p$ denotes the current valuation of port p of component c. Moreover, CTAs allow for the specification of activation and connection predicates:

- *Activation predicates* can be used to specify activation and deactivation of components. An activation of component c, for example, is denoted with $\|c\|$.
- *Connection predicates* can be used to specify connection between component ports. A connection between port p of component c and port p' of component c', for example, is denoted with $c.p \rightarrow c'.p'$.

2.1.3 Configuration Diagrams

We use Configuration Diagrams (CDs) as introduced in [27] as a graphical notation to support the specification of interfaces. A CD is a graph whose nodes resemble interfaces (group of ports) and whose edges denote connections between component ports. CDs can be annotated by certain common activation and connection constraints:

- *Activation annotations* can be used to introduce common activation constraints, such as min./max. number of components of a certain type.
- *Connection annotations* can be used to denote connection constraints, such as required connections between components of a certain type.

2.2 Model Checking

Model checking [4] (MC) is a technique for automatically verifying the correctness of certain properties against a model of a finite-state system. The model of the system is thereby usually given in terms of a finite state machine and the properties are specified in terms of temporal logic formulae.

In this work we will use the NuSMV symbolic model checker [10]. A NuSMV model always consists of several modules, each of which represents a distinct state machine. Each module consists of 2 main parts: a VAR part for declaring module variables and an ASSIGN part for defining the logic of the module, i.e., the initial value of the variables as well as state changes. Moreover, an optional part DEFINE can be used to define variable abbreviations. Modules can interact with each other through parameters (input) and (output) variables. Every module has to specify the parameter it gets, which acts like input variables and can itself define variables in the DEFINE section that can be read by other modules in their transfer parameters. Properties can then be expressed in LTL as well as CTL and checked against the model.

3 Approach

Figure 2 provides an overview of our approach to verify patterns for dynamic architectures and the corresponding artifacts: After consulting related literature describing a pattern, we identify and specify the interfaces of involved component types. Then, the logic of each component type is modeled by a state machine and the guarantee of the pattern is given in terms of configuration trace assertions. Then, the model of the pattern as well as the formalized guarantee is translated into a corresponding NuSMV specification to verify whether the guarantees indeed hold.

Example 1 (Running Example: Model-View-Controller)**.** *In the following, we demonstrate each step of the approach by means of a running example. Therefore, we choose the Model-View-Controller (MVC) patter which is commonly used for the design of human-computer interfaces [9, 33, 34]. The pattern is well-suited for demonstration since it is not to complex, yet it provides all important aspects of pattern specifications.*

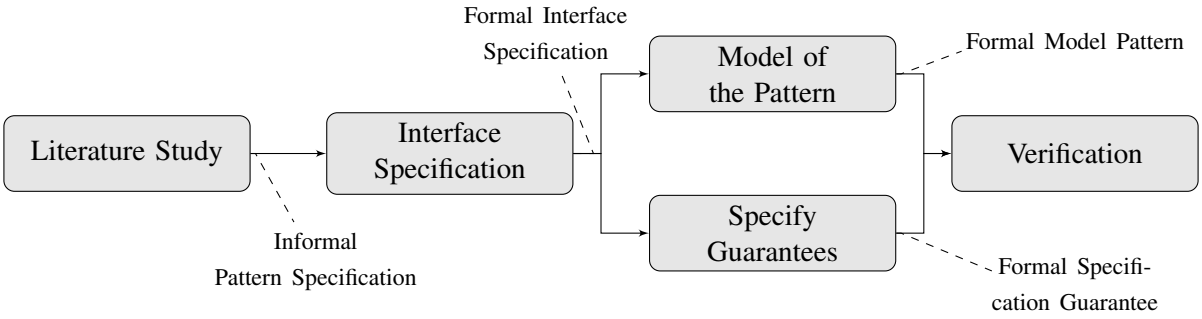

Figure 2: General approach

3.1 Literature Study

The study of a pattern starts by consulting current literature to identify the major component types and constraints about their interrelationships and activation. Moreover, described claims about the guarantees of a pattern should be collected and documented.

Example 2 (MVC Literature Study)**.** *It describes three types of components:*

Model: *a unique component responsible for consistently storing the necessary data.*

View: *displays the data of the model to the user.*

Controller: *handles user inputs.*

The pattern requires that there exists a designated controller component for each view component. A controller component recognizes user input (usually through events) and invokes a suitable service in the (unique) model component to update the model data accordingly. After this process, all views (and corresponding controllers) have to be updated with the new data from the model which is usually done through notifications.

The claimed benefits of the MVC-pattern are described as follows:

B1: Single point of data storage.
B2: Data consistency throughout the model and the views.
B3: Easy to extend with new views.
B4: High cohesion and low coupling of the components.

3.2 Interface Specification

Having an informal description of the pattern, we apply the techniques described in Sect. 2.1 to formalize the specification. As a first step, the syntactic interfaces of the involved component types have to be formalized. As described in Sect. 2.1, this can be done either graphically by means of configuration diagrams, or it can be done using interface specification templates. For each interface we specify local, input and output ports as well as their types.

Example 3 (MVC Interface Specification)**.** *As already described in the informal description of the pattern, the pattern consists of three types of components: Model, View, and Controller components.*

Figure 3 provides a formal specification of the interface for Model components. Each Model has two local ports: data *to store all the data and* service_ele *to store the name of the currently running service. It receives a service call through the* service *input port and a request to deliver the current data state through the* getData *input port. Moreover, it provides its data through the* output *output port and notifies its environment about data changes through its* notificate *output port.*

ISpec Model	**uses** Array
loc: service_ele, data **in**: service, getData **out**: output, notificate	
service, service_ele:	$-1\ldots2$
data, output:	array $0\ldots2$ of $0\ldots10$
getData, notificate:	bool

Figure 3: Model Interface Specification.

In Fig. 4 we provide a formal specification of the interface for view components. A view component stores its data in a corresponding local port named view_data*. Since we have more components of type view, each view has an associated identifier which is stored in local port* view_ele *and which is set at the beginning through its* element *input port. Moreover, it gets notified about data updates and receives them through the* notification *and* model_data *input ports. Finally, it requests new data from the model through its* getData *output port.*

ISpec View	**uses** Array
loc: view_data, view_ele **in**: notification, model_data, element **out**: getData	
view_data:	$0\ldots10$
view_ele, element:	$0\ldots2$
model_data:	array $0\ldots2$ of $0\ldots10$
notification, getData:	bool

Figure 4: View Interface Specification.

Figure 5 provides a specification of interfaces for controller components. The event a controller is currently working on is stored in the event_actual *local port. Again, each controller has a unique identifier which it receives through the corresponding* contr_id *input port. Through its* random *input port, a controller receives a notification about the occurrence of an event. The corresponding service invocation is then communicated through its* service *output port.*

ISpec Controller	
loc: event_actual **in**: contr_id, random **out**: service	
event_actual, service:	$-1\ldots2$
contr_id, random:	$0\ldots3$

Figure 5: Controller Interface Specification.

Having a formal specification of the components interfaces allows now to specify the behavior of a component type as well as the claimed guarantees over these interfaces.

3.3 Model of the Pattern

The behavior of the different component types is modeled using traditional mealy state machines. Thereby, each component type consists of a set of control states of which one is active at each point in time. State changes are triggered by valuations of a component's input ports and may result in valuations of component output ports as well as in a change of the currently active control state.

Example 4 (MVC Behavior Specification)**.** *In the following we specify the abstract behavior of model, view, and controller components, respectively.*

Figure 6 provides a specification of the behavior of model components. It is initialized (start) by setting service_ele *to* -1 *and initializing its data store* data*. If it gets a* service$!=-1$ *it outputs a* notificate=*false event, sets* service_ele=service*, and changes to control state* notification*. Thereby, depending on the valuation of its* getData *input port, it returns either an empty output (1) or a copy of* data *on its* output *port (2). In control state* notification*, a model immediately changes back to* update*. Thereby it changes its data store* data[service_ele] *depending on the effects of the service call and notifies its environment about this changes by setting* notificate=*true. Again, depending on the valuation of its* getData *port it returns either an empty output (3) or a copy of* data *on its* output *port (4). Finally, in control state* update *again, it waits for incoming data requests and depending on the valuation of its* getData *port it returns either an empty output (5) or a copy of* data *on its* output *port (6).*

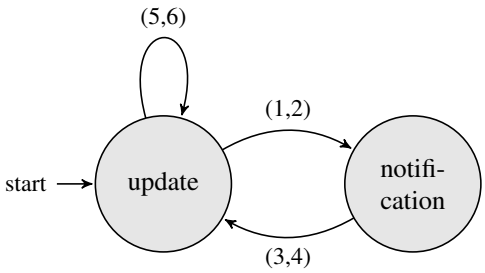

Figure 6: Model behavior.

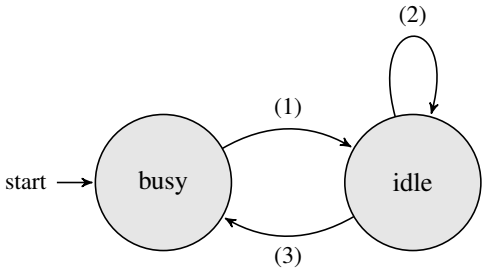

Figure 7: View behavior.

The behavior of components of type view is described by the state machine depicted in Fig. 7. A view component is initialized (start) with a unique identifier, received through its element *input port and stored in the* view_ele *local port. It starts in control state* busy *where it immediately changes to the* idle *state, outputs a request to receive data by setting* getData=*true, and updates its local data store accordingly by setting* view_data=model_data[view_ele] *(1). It remains idle until it gets notified about a data-change from the model (2). After getting notified through its input port* notification*, a view changes again its control state back to its initial state* busy *(3).*

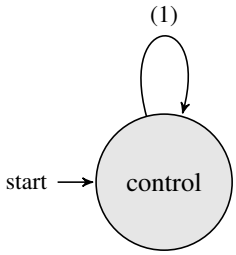

Figure 8: Controller behavior.

Finally, Fig. 8 models the behavior of a controller component. Similar as a view, a controller gets initialized (start) by an identifier. It then continuously generates events which it forwards to the model by sending a corresponding service identifier through its output port service *(1).*

3.4 Specify Constraints

As a next step, one can start to formalize the claims identified in Sect. 3.1 over the formal interface specification developed in Sect. 3.2. Therefore, one can again leverage the methods and techniques presented in Sect. 2.

Example 5 (MVC Guarantee Specification)**.** *In Ex. 2 we identified several claims made about a system build in a MVC pattern. In the following we use a so-called configuration trace assertion template to formalize some of these guarantees of the MVC pattern.*

Figure 9 provides a formal specification of two properties for MVC Architectures. Eq. (1) *requires that whenever a controller executes a service (in response to a user event), the views will at some point in the future request an update of their data. Eq.* (2)*, on the other hand, requires data consistency of the model and the views, i.e., that the data in the views is always updated according to their corresponding data in the model.*

Spec MVC		**uses** *Array*
var	m:	Model
	v:	View
	c:	Controller

$$\Box\Big(c.service \neq -1 \implies \forall v\colon \big(\Diamond(v.getData)\big)\Big) \quad (1)$$

$$\Box\Big(x = m.data[v.view\_ele] \implies \Diamond\big(v.view\_Data = x\big)\Big) \quad (2)$$

Figure 9: Specification of MVC guarantees.

3.5 Verification

In the last step we apply model checking to verify the guarantees against the model of the pattern. First, the interface specification developed in Sect. 3.2 is used to build a corresponding NuSMV structure. Then, the model of the pattern developed in Sect. 3.3 is systematically translated to a corresponding NuSMV model. Finally, the specification of the pattern guarantees developed in Sect. 3.4 are transferred to a corresponding LTL specification in NuSMV and verified against the model.

3.5.1 Translate Interfaces

The specification of a pattern's interfaces is used to develop a raw structure of the NuSMV model. The general structure of a NuSMV module consists of a VAR, ASSIGN and DEFINE part. Algorithm 1 describes the systematic translation of a pattern's interface specification to a NuSMV raw structure.

3.5.2 Translate Model

In the next step, the NuSMV template is enhanced using the specification of component type behavior. Algorithm 2 describes the systematic translation of a behavioral specification to a corresponding NuSMV pattern template.

Algorithm 1 Translate Interfaces to NuSMV

Require: interface specification of component types
1: create a new NuSMV-File with one main module
2: **for all** interface specifications *is* **do**
3: create a new module for *is*
4: **for all** input ports *ip* of *is* **do**
5: create module parameter for *ip*
6: **end for**
7: **for all** local ports *lp* of *is* **do**
8: create module variable for *lp*
9: **end for**
10: **for all** output ports *op* of *is* **do**
11: create variable in DEFINE part for *op*
12: **end for**
13: **end for**
14: **return** NuSMV template for the pattern

Algorithm 2 Translate Model to NuSMV

Require: NuSMV template + model (state machines)
1: **for all** component types *ct* **do**
2: create variable *controlState* with entries for each control state in *ct*'s VAR part
3: define initial states of all local variables in *ct*'s ASSIGN part
4: encode state machine using *next* statements in *ct*s ASSIGN part
5: define the valuation of output variables in the *ct*'s DEFINE part
6: **end for**
7: **return** enhanced NuSMV model with logic for all component type modules

Finally, the concrete architecture is configured in module main. First, components are instantiated and then, the configuration is encoded by connecting output ports (variables in DEFINE part) to corresponding input ports (module parameters) of the corresponding components.

3.5.3 Translate Pattern Guarantees

Now, as the NuSMV model is complete, we translate the guarantees formalized in Sect. 3.4 to corresponding NuSMV LTL specifications.

Example 6 (Verifying the MVC pattern)**.** *Listing 1 shows the NuSMV code resulting by applying Alg. 1 and Alg. 2 to the specifications developed in Ex. 3 and Ex. 4.*

Listing 1: NuSMV code for module *model*.

```
MODULE model (service, getData)
VAR
  controlState: {Update, Notification};
  service_ele : -1..2;
  data : array 0..2 of 0..10;
ASSIGN
  init(controlState) := Update;
  init (service_ele) := -1;
  init (data[0]) := 7;
  init (data[1]) := 5;
  init (data[2]) := 3;
  next (controlState) := case
    controlState=Update & service != -1 : Notification;
    TRUE : Update;
  esac;
  next(service_ele):= service;
    next(data[0]):= case
    controlState = Notification & service_ele = 0 : (data[0] + 7) mod 10;
    TRUE : data[0];
  esac;
  next(data[1]):= case
    controlState = Notification & service_ele = 1 : (data[1] + 5) mod 10;
    TRUE : data[1];
  esac;
  next(data[2]):= case
    controlState = Notification & service_ele = 2 : (data[2] + 3) mod 10;
    TRUE : data[2];
  esac;
DEFINE
  output :=   case
    getData : data;
    TRUE : output_empty;
  esac;
  output_empty:= [-1,-1,-1];
  notifacte:= case
    controlState = Notification : TRUE;
    TRUE : FALSE;
  esac;
```

Similarly, the models for view and controller components are translated to NuSMV.

As a next step, the main module is build to coordinate the different modules. Listing 2 depicts the corresponding NuSMV code.

Listing 2: NuSMV code for module *main*.

```
MODULE main
VAR
  view1: view(model.notifacte,model.output,0);
  controller1: controller(1, random);
  view2: view(model.notifacte,model.output,1);
  controller2: controller(2, random);
  view3: view(model.notifacte,model.output,2);
  controller3: controller(3, random);
  model: model(service, getData);
  random: 0..3;
ASSIGN
  init(random) := 0;
  next(random) := {1,2,3};
DEFINE
  service := controller1.service + controller2.service + controller3.service + 2;
  getData := view1.getData | view2.getData | view3.getData;
```

The main module is instantiating the components in the VAR part. It passes to every component the needed parameter. Most of them are output variables from the other components, but for example the IDs are passed as a static numeric. In the VAR part we also find a variable random*. The random variable is necessary to decide which of the controllers is allowed to invoke a service in the model. Without this restriction multiple controllers could invoke a service even though the model can only handle one call. In the DEFINE section we see a* service *and a* getData *variable.* service *indicates which service is called and* getData *whether a view is requesting an update of the data.*

Finally, the specification of the guarantees developed in Ex.5 is translated to a corresponding LTL specification in NuSMV (Listing 3).

Listing 3: LTL Specification

```
LTLSPEC G ((controller1.service + controller2.service + controller3.service)> -3)
  -> (F view1.getData) & (F view2.getData) & (F view3.getData)

LTLSPEC G model.data[view1.view_ele] = 0 -> F view1.view_data = 0
LTLSPEC G model.data[view2.view_ele] = 0 -> F view2.view_data = 0
...
```

4 Results

We applied the approach to formalize and verify three contemporary patterns for dynamic architectures: The Singleton, the MVC, and the Broker pattern. Table 1 provides an overview of our results[1]. In the following we discuss them in more detail.

[1] All the corresponding NuSMVscripts can be downloaded at `http://www.marmsoler.com/mc/`.

Table 1: Analysis Results.

Pattern	Prop.	Type	Description
Singleton	S1	liveness	whenever an instance is required it is returned eventually
	S2	safety	at every time, there is only one component of type singleton active
MVC	M1	liveness	views are eventually updated with user events observed by controllers
	M2	liveness	the data of model and view are eventually in a consistent state
Broker	B1	liveness	every service requires from a client is eventually executed by some server
	B2	liveness	a server requests to registered for a broker eventually results in a registration

4.1 Singleton

As shown in Tab. 1, we verified two common properties for the Singleton pattern. Finding S1 ensures that architectures applying the pattern are guaranteed to eventually activate a singleton if required. One observation we found during analysis is that the time required to access a singleton component may vary depending on whether it was the first access to the component or a subsequent access. This is because the first access usually requires to newly instantiate the singleton which requires some time.

The second finding S2 states another characteristic property of the Singleton pattern: Each architecture applying the pattern is guaranteed to have at most one active component of type singleton. During the analysis of this property, we found that there might be different interpretations of the pattern. First, the condition that there can only be one instance, can be interpreted as one instance which is activated every time or as maximal one instance (i. e. none or one). Another question that arises is whether the changing instance needs to be always the same or whether it is allowed to change over time.

4.2 MVC

Table 1 shows the properties verified for the MVC pattern. The first property M1 ensures that whenever an event is registered by the controller, all the views request a new copy of their data from the model. It is important that the views should be initialized in the idle state instead of the busy state since they need to request the model data at the beginning.

Finding M2 ensures consistency of the data from the views with the data from the model. It states that whenever data changes in the model, the corresponding data in the view is eventually updated.

4.3 Broker

As shown in Tab. 1, we investigated two properties for the Broker pattern. Finding B1 states that each service requested by a client is eventually executed by some of the servers. Thereby, the client communicates only with the broker and does not have to know where the service is actually executed.

Finally, finding B2 ensures that a server which wants to be registered to the broker, is guaranteed to be eventually registered. This is important to guarantee that the services of a registered server can be accessed by the clients.

In order to not complicate the pattern, we did not incorporate a bridge component and stick to one unique broker component. Our investigation suggests that there are different possibilities to implement the pattern. With our implementation every request is handled for sure, but only because the requesting itself is restricted. In practice, the client, the broker, and the server usually execute in different locations, which is why the information also has to be marshalled and un-marshalled.

Table 2: Computational effort.

Pattern	Lines of Code	LTL Specs	LTL average	LTL total	#Variables	#Components
Singleton	68	2	25 ms	50 ms	10	1
MVC	89	30	30 ms	1500 ms	15	4
Broker	309	6	1000 ms	6000 ms	42	6

5 Discussion

In the following section we summarize our experience with using model checking techniques to investigate patterns for dynamic architectures.

In general, we found that pattern specifications are usually underspecified in literature and that many different interpretations are possible where only some of them lead to the desired properties. Consider, for example, the Singleton pattern where there is ambiguity in whether the instance has to be active all time or can get deactivated and another instance can get active. We see our approach as helpful for architects to find and tackle these questions.

Scalability When it comes to formal methods, scalability is sometimes considered a critical issue. We tried to overcome this problem by raising the level of abstraction of our analyses. By investigating patterns, rather than concrete architectures, we concentrate on the architecturally important aspects and thus reduce complexity of the analyses.

Table 2 shows the computational effort to perform our analyses on a standard notebook. The table shows that the computational effort for single as well as for medium complicated patterns (Singleton and MVC) is very low (25 ms and 30 ms, respectively). Note that the Broker pattern is actually one of the more complex patterns documented in literature. However, we can see that also for this pattern the computational effort to analysis is rather low (1000 ms).

Effort Another problem usually mentioned in formal methods is the required effort. Figure 10 provides an overview of the effort needed to perform the analyses described in the paper[a]. We can see that the *relative* effort strongly decreased with growing experience. Indeed, after some experience, also relatively complex patterns (e.g. the Broker pattern) can be analyzed with a reasonable amount of effort (1 PM).
Again we would like to point out the impact of the effort. Since the results are at pattern level, the impact is high since it influences every concrete architecture implemented in a certain pattern.

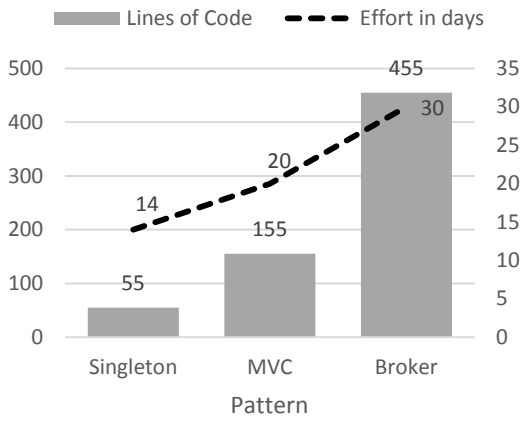

Figure 10: Effort applying the approach.

[a]All analyses were conducted by the same person. The person had a computer science background with no experience of using Model checking techniques so far.

Quality attributes. A last point which needs to be discussed in more detail regards an important aspect of software architectures in general. Our approach does actually not provide means to directly specify quality attributes such as performance, availability, etc. However, as our example shows, it allows us to specify the technical realization of such aspects. For example, finding S2 of the Singleton pattern can be used to ensure an upper bound of active components at each point in time. This could be actually seen as a possible realization of one aspect of efficiency.

6 Related Work

Recently, some approaches emerged which focus on the verification of software architectures and architecture patterns. They can roughly be classified into two main groups: automatic and interactive verification.

6.1 Interactive Pattern Verification

Work in this area applies interactive theorem proving (ITP) to pattern verification. One example comes from Marmsoler and Gleirscher [28] who apply the Isabelle/HOL ITP to investigate patterns for dynamic architectures. While interactive approaches are very expressive, they usually require manual interaction for verification. Thus, with our approach we actually complement work in this area by providing an alternative to verify certain properties automatically.

6.2 Automatic Pattern Verification

Work in this area applies automatic techniques to pattern verification. One example of work in this area comes from Kim and Garlan [21] who apply the Alloy [20] analyzer to automatically verify architectural styles specified in ACME [18]. A similar approach comes from Wong et al. [38] which also applies Alloy to the verification of architecture models. In contrary to this work, however, both approaches are using the Alloy Analyzer, which is based on SAT-solving whereas we are using model checking with focus on LTL formulae.

Another related approach in this area comes from Wirsing et al. [14] where the authors apply rewriting logic to specify and verify cloud-based architectures. Again, the authors apply rewriting logic whereas the focus of this work was to investigate the feasibility of model checking technologies.

Finally, Zhang et al. [39] applied model checking techniques to verify architectural styles formulated in Wright#, an extension of Wright [3]. However, whereas their work focuses on strictly static patterns, in this work we aim to support dynamic architectures, as well.

Indeed, to the best of our knowledge this is the first attempt to apply model checking to the verification of patterns for dynamic architectures.

7 Conclusion

With this paper we report on our experience of applying model checking for the *verification* of patterns for dynamic architectures. To this end, we first describe a 5-step approach to systematically formalize a pattern and its corresponding guarantees and employ model checking to verify the guarantees against the model of the pattern. Then, we apply the approach to investigate 3 commonly used patterns: the Singleton, the Model-View-Controller, and the Broker pattern. For each pattern, we formalized and verified two characteristic properties.

We found that patterns (also well-known once) are usually underspecified in literature and that many different interpretations are possible where only some of them lead to the promised guarantees. Thus, we conclude that the proposed approach is useful since it helps to discover such ambiguities. Moreover, our work shows that the approach is feasible also for relatively complex patterns (such as the Broker pattern). The additional effort is justified by the general nature of the results: each result at pattern level applies to every architecture applying the pattern.

We envisage three possible implications of our results: (i) *Analysis of existing patterns*: our approach can be used to specify and analyze existing patterns and uncover ambiguities and flaws in their specification. (ii) *Design of new patterns*: the approach can also be applied to help in the design of new patterns. A pattern can be formally specified and then be verified by the designer without even requiring an implementation thereof. (iii) *Pattern conformance analysis*: the formal pattern specifications may be used to help verifying that an architecture indeed implements a certain pattern. For example, the abstract model of the Singleton pattern provided in Ex. 4 may be used to check whether a concrete architecture indeed implements this pattern. In turn it is guaranteed that the architecture fulfills the desired guarantee provided in Ex. 5. To support these implications, future work is needed in two major areas: First, the approach should be applied to formalize and analyze further patterns (existing ones as well as new ones). Then, the results should be incorporated into tools to support in the (possible automatic) verification of pattern conformance.

Acknowledgments. We would like to thank Manfred Broy, Vasileios Koutsoumpas and all the anonymous reviewers for their comments and helpful suggestions on earlier versions of this paper.

References

[1] Gregory D Abowd, Robert Allen & David Garlan (1995): *Formalizing Style to Understand Descriptions of Software Architecture*. *TOSEM*, doi:10.1145/226241.226244.

[2] Robert Allen, Remi Douence & David Garlan (1998): *Specifying and Analyzing Dynamic Software Architectures*. In: *FASE*, doi:10.1007/bfb0053581.

[3] Robert J Allen (1997): *A Formal Approach to Software Architecture.* Technical Report, DTIC Document.

[4] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

[5] Marco Bernardo, Paolo Ciancarini & Lorenzo Donatiello (2000): *On the Formalization of Architectural Types with Process Algebras*. In: *ACM SIGSOFT SEN*, doi:10.1145/357474.355064.

[6] Lars Bratthall, Enrico Johansson & Björn Regnell (2000): *Is a Design Rationale Vital when Predicting Change Impact? –A Controlled Experiment on Software Architecture Evolution*. In: *PROFES*, doi:10.1007/978-3-540-45051-1_14.

[7] Manfred Broy (1996): *Algebraic Specification of Reactive Systems*. In: *AMAST*, doi:10.1007/bfb0014335.

[8] Manfred Broy (2014): *A Model of Dynamic Systems*. In: *FPS*, doi:10.1007/978-3-642-54848-2_3.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad & Michael Stal (1996): *PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A System of Patterns*.

[10] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri & Andrei Tchaltsev: *NuSMV 2.5 User Manual, 2010*. `http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf`.

[11] Corporation MITRE (2011): *CWE/SANS Top 25 Most Dangerous Software Errors*. Available at `https://cwe.mitre.org/top25/index.html`.

[12] Juan de Dalmau & Jacques Gigou (1997): *Ariane-5: Learning from Flight 501 and Preparing for 502*. Available at `http://www.esa.int/esapub/bulletin/bullet89/dalma89.htm`. ESA Bulletin Nr. 89.

[13] Eric M Dashofy, André Van der Hoek & Richard N Taylor (2001): *A Highly-Extensible, XML-Based Architecture Description Language*. In: *WICSA*, doi:10.1109/wicsa.2001.948416.

[14] Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer & Martin Wirsing (2012): *Stable availability under denial of service attacks through formal patterns*. In: *FASE*, doi:10.1007/978-3-642-28872-2_6.

[15] José Luiz Fiadeiro & Antónia Lopes (2013): *A Model for Dynamic Reconfiguration in Service-oriented Architectures*. *SoSyM*, doi:10.1007/s10270-012-0236-1.
[16] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (1994): *Design Patterns: Elements of Reusable Object-Oriented Software*.
[17] David Garlan (2000): *Software Architecture: a Roadmap*. In: *FOSE*, doi:10.1145/336512.336537.
[18] David Garlan (2003): *Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events*. In: *Sfm*, doi:10.1007/978-3-540-39800-4_1.
[19] Paola Inverardi & Alexander L Wolf (1995): *Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model*. *IEEE TSE*, doi:10.1109/32.385973.
[20] Daniel Jackson (2002): *Alloy: A Lightweight Object Modelling Notation*. *TOSEM*, doi:10.1145/505145.505149.
[21] Jung Soo Kim & David Garlan (2006): *Analyzing Architectural Styles with Alloy*. In: *ROSATEA*, doi:10.1145/1147249.1147259.
[22] Daniel Le Métayer (1998): *Describing Software Architecture Styles Using Graph Grammars*. *IEEE TSE*, doi:10.1109/32.708567.
[23] Zengyang Li, Peng Liang & Paris Avgeriou (2015): *Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios*. In: *WICSA*, doi:10.1109/WICSA.2015.19.
[24] David C Luckham, John J Kenney, Larry M Augustin, James Vera, Doug Bryan & Walter Mann (1995): *Specification and Analysis of System Architecture Using Rapide*. *IEEE TSE* 21(4), pp. 336–354, doi:10.1109/32.385971.
[25] Jeff Magee & Jeff Kramer (1996): *Dynamic Structure in Software Architectures*. *ACM SIGSOFT SEN*, doi:10.1145/239098.239104.
[26] Zohar Manna & Amir Pnueli (1992): *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Science & Business Media, doi:10.1007/978-1-4612-0931-7.
[27] Diego Marmsoler: *DACL - Dynamic Architecture Constraint Language*. `http://www.marmsoler.com/docs/DACL.pdf`. Available at `http://www.marmsoler.com/docs/psl.pdf`.
[28] Diego Marmsoler & Mario Gleirscher (2016): *Specifying Properties of Dynamic Architectures using Configuration Traces*. In: *ICTAC*, Springer, doi:10.1007/978-3-319-46750-4_14.
[29] Nikunj R Mehta & Nenad Medvidovic (2003): *Composing Architectural Styles From Architectural Primitives*. In: *ACM SIGSOFT SEN*, ACM, doi:10.1145/949952.940118.
[30] Mark Moriconi, Xiaolei Qian & Robert A Riemenschneider (1995): *Correct Architecture Refinement*. *IEEE TSE*, doi:10.1109/32.385972.
[31] Flavio Oquendo (2004): *π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures*. *ACM SIGSOFT SEN*, doi:10.1145/986710.986728.
[32] John Penix, Perry Alexander & Klaus Havelund (1997): *Declarative Specification of Software Architectures*. In: *ASE*, doi:10.1109/ase.1997.632840.
[33] Mary Shaw & David Garlan (1996): *Software Architecture: Perspectives on an Emerging Discipline*. 1, Prentice Hall Englewood Cliffs.
[34] Richard N Taylor, Nenad Medvidovic & Eric M Dashofy (2009): *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, doi:10.1145/1595696.1595754.
[35] Michel Wermelinger & José Luiz Fiadeiro (2002): *A graph transformation approach to software architecture reconfiguration*. *SCP*, doi:10.1016/s0167-6423(02)00036-9.
[36] Michel Wermelinger, Antónia Lopes & José Luiz Fiadeiro (2001): *A Graph Based Architectural (Re)configuration Language*. In: *ACM SIGSOFT SEN*, doi:10.1145/503271.503213.
[37] Martin Wirsing (1990): *Algebraic Specification*. In: *Handbook of Theoretical Computer Science (Vol. B)*, doi:10.1016/b978-0-444-88074-1.50018-4.
[38] Stephen Wong, Jing Sun, Ian Warren & Jun Sun (2008): *A Scalable Approach to Multi-Style Architectural Modeling and Verification*. In: *ICECCS 2008*, doi:10.1109/iceccs.2008.16.
[39] Jiexin Zhang, Yang Liu, Jing Sun, Jin Song Dong & Jun Sun (2012): *Model Checking Software Architecture Design*. In: *HASE*, doi:10.1109/hase.2012.12.