

Dynamic Software Architecture Development: Towards an Automated Process

Maurice H. ter Beek
ISTI-CNR, Pisa, Italy
terbeek@isti.cnr.it

Antonio Bucchiarone
FBK-Irst, Trento, Italy
bucchiarone@fbk.eu

Stefania Gnesi
ISTI-CNR, Pisa, Italy
gnesi@isti.cnr.it

Abstract—We propose a software engineering process to aid the development of Dynamic Software Architectures (DSAs). This process is based on the sequential application of a number of formal methods and tools, and it can support software architects throughout the design, analysis and code generation of software systems. To illustrate the process, we apply it to an industrial case study from the Service-Oriented Computing (SOC) domain.

I. INTRODUCTION

We propose an integrated software engineering process consisting of three approaches that are largely based on some of our independently obtained research results [2], [4]–[8], [12]. The current paper should be seen as an introduction to these papers, summarizing their contributions without considering all the technical details, augmented with a vision on the way these contributions can be used to form an engineering process to support the development of DSAs. This process consists of the appropriate sequential application of several formal methods and tools, notably graph grammars [4], [5], ALLOY [8], [10], UMC [12], UCTL [2] and ARCHJAVA [1].

It is composed of three main development phases:

- 1) Formally design structural and behavioural aspects;
- 2) Formally verify structural and behavioural properties;
- 3) Automatically generate code from architectural model.

The graph-based design produced in the first phase allows the use of model checking in the second phase to automatically verify correctness properties of the design. This allows the evaluation of design alternatives before implementation and testing. Since a large percentage of software errors is due to design errors, and since these are among the most expensive ones to resolve if discovered only after implementation, our process may contribute to considerable reductions in cost and to improved quality. Finally, the JAVA code that is automatically generated in the third phase has the advantage of originating from a design whose structure and behaviour have been proved to satisfy certain desired functional properties.

II. THE TRAFFIC LIGHT PROCESS

We propose a process consisting of three phases (hence the name traffic light). As sketched in Fig. 1, each phase is represented by a colour of the traffic light: **Red** for structural design and analysis, **Yellow** for behavioural design and analysis, and **Green** for automatic code generation. These phases are bound to particular languages and models, briefly described next.

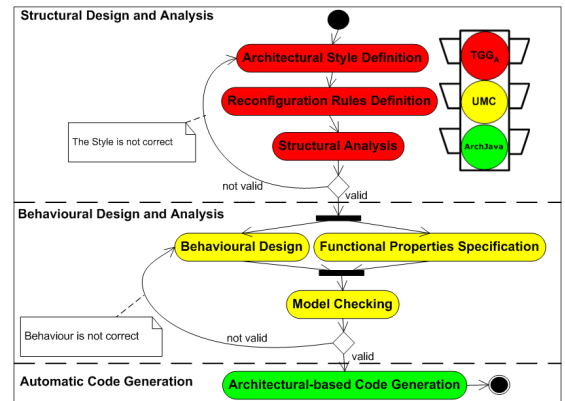


Fig. 1. The traffic light process.

A. Red Phase: Structural Design and Analysis

For the **red** phase we have chosen graph grammars to model the different notions of dynamicity because they provide (i) both a formal basis and a graphical representation that is in line with the usual way of representing architectures; (ii) a natural way to describe styles, configurations and reconfigurations. We define architectural styles by *type hypergraphs* and a set of constraints. Type hypergraphs describe the types of components, connectors, ports, rules and their connections. A configuration compliant to a type hypergraph T is described by a T -typed hypergraph. Each DSA is represented by a typed hypergraph; its components (connectors) modelled by hyperedges, its ports (roles) by outgoing tentacles. Components and connectors are attached by connecting their respective tentacles to the same node. We represent DSA reconfigurations by rewriting rules for typed hypergraphs. For details and examples of this formalization, see [4]–[6], [8].

The tool we have chosen to support the formal specification of DSAs is ALLOY [10], a description language for software models, based on signatures and relations, which we found very suitable to model hypergraphs associated to DSAs. Also, ALLOY is based on a rather simple notation, making it easy to use. We have used ALLOY to implement formal aspects of typed graph grammars (i.e., the TGG_A framework [6]).

ALLOY also provides an extension of first-order logic with relational operators to represent properties/constraints of the models. The ALLOY Analyzer can translate the model and the logical predicates into a (usually large) boolean formula, using

efficient SAT solvers to decide satisfiability, and provide a counterexample if the predicate does not hold over the model. It searches for counterexamples up to a bound k in the model’s number of elements. We have used it to show how to ensure style-consistency, to perform model finding and to validate architectural structural properties (like invariant analysis).

B. Yellow Phase: Behavioural Design and Analysis

Starting from a DSA designed in the **red** phase, in the **yellow** phase we validate the DSA’s conformance to certain functional properties using model checking. We have chosen to use our in-house model checker UMC [12]. It allows efficient verification of functional correctness properties formalized in the action-based and state-based branching-time temporal logic UCTL [2] over a set of communicating UML state machines (describing the components’ dynamic behaviour). UCTL thus allows specifying the properties a state should satisfy and combining these basic predicates with advanced temporal operators dealing with the actions performed. UCTL’s semantic domain is a doubly-labelled transition system (L^2TS), a transition system with nodes labelled by atomic propositions and transitions labelled by sets of actions.

UMC is an on-the-fly model checker allowing efficient verification of UCTL formulae over a set of communicating UML state machines. The possible system evolutions are represented as an L^2TS , whose nodes represent the various system configurations and whose transitions represent the possible evolutions of a system configuration. UMC uses an on-the-fly model-checking algorithm, which has as advantage that, depending on the formula, only a fragment of the overall state space might need to be generated and analyzed to produce the correct result. This algorithm has a linear complexity in the size of the formula and the dimension of the L^2TS . The current UMC prototype can be experimented via a web interface [12].

For each DSA configuration in the set of architectural configurations generated during the **red** phase, we associate to each component (i.e. hyperedge) a communicating UML state machine describing its behaviour. The complete behavioural specification is then composed of a set of UML state machines. We then use UMC to verify behavioural properties specified in UCTL. UCTL allows the specification of many basic liveness and safety properties a runtime DSA configuration should satisfy. If the DSA design is not properly specified, it needs to be revised, after which UMC is used again. When the DSA is validated, the **green** phase is entered.

C. Green Phase: Automatic Code Generation

After the DSA has been validated w.r.t. the desired properties (both structural and behavioural), JAVA code is automatically generated in the **green** phase. This phase has already been outlined in [7], but for our purposes it has been adapted to use UMC as model checker. It is performed in two steps: Starting from a validated UMC specification, ARCHJAVA code is automatically obtained by means of a JET-based code generator. Then, exploiting the existing ARCHJAVA Compiler, executable JAVA code is generated.

This phase thus uses ARCHJAVA, extending JAVA with component classes (describing objects that are part of the architecture), connections (enabling components’ communication) and ports (connection endpoints). Communication integrity is a key property enforced by ARCHJAVA: It ensures components can only communicate using connections and ownership domains that are explicitly declared in the DSA. ARCHJAVA guarantees communication integrity between an architecture and its implementation even in the presence of advanced architectural features like runtime component creation and connection. A prototype ARCHJAVA compiler is publicly available [1].

Each component (i.e. hyperedge) defined in the first phase is turned into an ARCHJAVA component, each of which has a set of ports (provided and required) and only connects a pair of components. This means that if a component must communicate with more than one component, it needs additional ports. The suitable number of required and provided ports is declared in the components’ ARCHJAVA specifications.

Moreover, an ARCHJAVA specification is generated for each UML state machine associated to a component in the UMC specification. ARCHJAVA does not offer direct support for this, but the second author et al. have extended the ARCHJAVA specifications so that a state machine associated to a software component is implemented as an adjacency list [7]. Finally, a main ARCHJAVA file is generated to define the binding among components’ ports and the instantiation of the involved state machines. These directives ensure the aforementioned communication integrity.

III. AUTOMOTIVE CASE STUDY

Advances in mobile technology have enabled telephone/Internet access in vehicles, which has resulted in many new services for the automotive domain. Their scenarios are used to validate the engineering approach developed in the EU project Sensoria [9], e.g. a Road Assistance Service (RAS): During a drive, the in-vehicle diagnostic system reports a too low oil level, indicating the car is no longer drivable, and sends a message with the diagnostic data and the GPS data to the RAS to find a solution (towing service, garage, rental car, etc.).

A. Structural Design and Analysis

We design and verify our case study with TGG_A , an ALLOY implementation of typed graph grammars. Each configuration of the DSA of the RAS can be composed from the components in Fig. 2, which we describe briefly next, while the following table spells out the abbreviations used for ports.

VV	Vehicle - Vehicle	OV	Orchestrator - Vehicle
VS	Vehicle - Service	BV	BANK - Vehicle
VO	Vehicle - Orchestrator	RAV	RAS - Vehicle
VGPS	Vehicle - GPS	LDO	LD - Orchestrator
OLD	Orchestrator - LD	GPSV	GPS - Vehicle

BANK: Offers bank operations relevant for RAS application;

GPS: Provides GPS data (like vehicle’s current position);

LD (Local Discovery): Looks for appropriate services in the local repository;

RAS: Provides services needed for car repair (like garage);

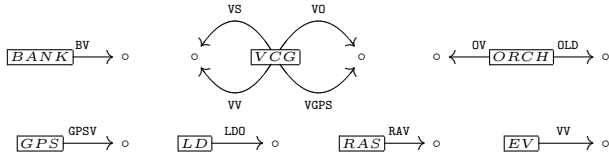


Fig. 2. Basic components of the RAS.

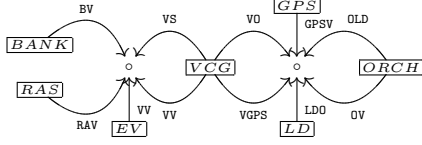


Fig. 3. TypeGraph of the RAS.

EV: External Vehicle that can be connected during a trip;
VCG (*Vehicle Communication Gateway*): Forwards messages to external components (*BANK*, *RAS*, *EV*);

ORCH (*Orchestrator*): Tries to achieve a goal by composing services, so upon each driver's request it performs a dynamic binding with in/external components *BANK*, *GPS*, *RAS*, etc.

After identifying each component of the RAS, we have defined the RAS style for each DSA configuration to conform with. To do so, we defined an ALLOY module implementing the *TypeGraph* of Fig. 3. Cf. [11] for the full ALLOY code.

Our process offers two analysis techniques (via ALLOY) to validate a DSA's structure. *Model finding* is the main technique. The ALLOY analyzer basically explores a bounded fragment of the state space of all possible models. We can thus easily use it to construct an initial configuration of the RAS: We need to ask for a typed graph instance satisfying the RAS style constraints and with a certain number of components. To do so, we implemented a module called *Model Finding* that, starting from the definition of the elements of our initial configuration (components, ports, etc.), generates the set of all possible DSA configurations composed of a precise number of components, ports, etc. When the ALLOY run command is executed, the ALLOY analyzer generates all possible RAS-style-conformant configurations. Two are shown in Fig. 4 (ia_i nodes represent communications between components inside the car, ea_j nodes between in- and external components).

The second technique is *invariant analysis*: Given a property P , is it invariant under sequences of operations? An operation is a rewriting step that from an initial configuration G and a production P generates a new configuration G' . A useful way to state the invariance of a property P is to specify that P holds in the initial configuration, and that for every non-initial configuration and every rewriting step, both $P(G)$ and $rwStep(G, G') \rightarrow P(G')$ hold. We thus defined a set of reconfiguration rules, a set of properties to verify and an ALLOY predicate *Invariant Analysis* used to verify the invariants of a set of properties over our system. Fig. 5 shows two reconfiguration rules. Fig. 6 presents a possible runtime

evolution of the RAS as a set of transitions, each composed of a *startState*, an *arrivalState* and a *trigger* (i.e. reconfiguration rules). Exemplary properties for each DSA configuration of the RAS to satisfy after each reconfiguration step are:

- 1) No *VCG* is connected directly to *LD*;
- 2) If *LD* exists, then also *ORCH* and they are connected;
- 3) *ORCH* has exactly two connections;
- 4) *VCG* has at most one component attached to each port.

Checking the *Invariant Analysis* predicate for these properties results in each of these properties being valid for each reachable configuration of our running example. Considering the DSA formalization presented in [5] we can conclude that the structural specification of the RAS is correct and complete.

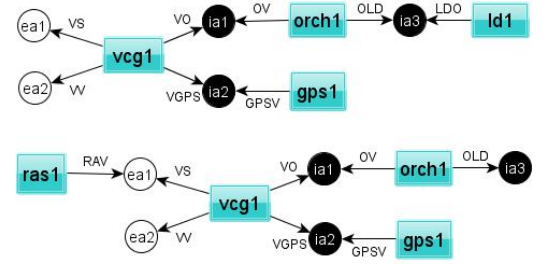


Fig. 4. Two RAS-style-conformant configurations.

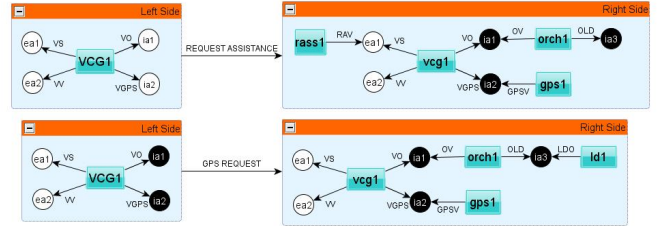


Fig. 5. Reconfiguration rules of the RAS.

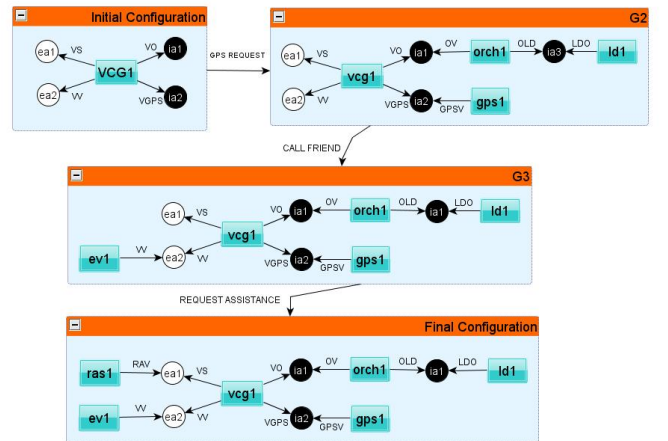


Fig. 6. Runtime evolution example of the RAS.

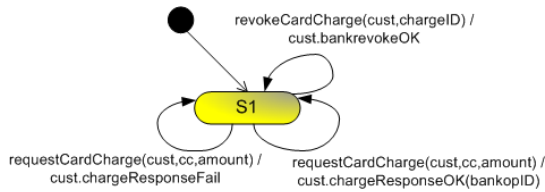


Fig. 7. The BANK state machine.

B. Behavioural Design and Analysis

We first specify the behaviour of all RAS components. Due to space limitations, we present the approach only for *BANK*. Its operations relevant for the RAS application are charging and revoking a credit card. Fig. 7 shows the *BANK* state machine describing its behaviour, while the corresponding UMC code is as follows (cf. [12] for the full UMC code).

```

Class Bank is
Signals :
  requestCardCharge(cust:Car, cc:Token, amount:Token);
  -- replies: cust.chargeResponseOK(chargeID)
  --          cust.chargeResponseFail
  revokeCardCharge(cust:Car, chargeID:Token);
  -- replies: revokeOK
State Top = s1
Transitions :
  s1 -> s1 {requestCardCharge(cust,cc,amount) /
            cust.chargeResponseOK(bankopID)}
  s1 -> s1 {requestCardCharge(cust,cc,amount) /
            cust.chargeResponseFail}
  s1 -> s1 {revokeCardCharge(cust,chargeID) / cust.revokeOK}
end Bank;

```

We used UMC to verify behavioural properties as “A service is *responsive* if it guarantees each request a response”. In UCTL:

$$AG [requestCardCharge] \\ A [true \ U_{chargeResponseOK \vee chargeResponseFail} \ true],$$

i.e. after *requestCardCharge* always eventually follows *chargeResponseOK* or *chargeResponseFail*. It takes UMC just seconds to verify it to be true. Such verifications can be used to show that the RAS requirements model is well designed [3].

C. Automatic Code Generation

Applying the JET-based code generator to the RAS’ UMC specification produces a set of ARCHJAVA files (one for each component). The state machine specifications are also synthesized, with a main file enabling execution of the obtained system w.r.t. the modelled DSA. In this file all components, state machines and port connections are instantiated, resulting in an encoding of the DSA’s properties. We list only the *BANK* state machine in ARCHJAVA code generated by the JET-based code generator (cf. [11] for the full ARCHJAVA code).

```

public class SM_BANK {
public final int S_S1= 0;
public final int T_revokeCardCharge=0;
public final int T_cust.bankrevokeOK=1;
public final int T_requestCardCharge=2;
public final int T_cust.chargeResponseOK=3;
public final int T_cust.chargeResponseFail=4;
private int currentState=S_S1;
private LinkedList states = new LinkedList(); ... }

```

```

public SM_BANK() { System.out.println("SM_BANK.constr");
LinkedList S1 = new LinkedList();
S1.add(new transition(T_revokeCardCharge, S_S1, 1));
S1.add(new transition(T_cust.bankrevokeOK, S_S1, 0));
S1.add(new transition(T_requestCardCharge, S_S1, 1));
S1.add(new transition(T_cust.chargeResponseOK, S_S1, 0));
S1.add(new transition(T_cust.chargeResponseFail, S_S1, 0));
states.add(S1); } ... }

```

IV. CONCLUSIONS AND FUTURE WORK

We outlined an architectural engineering process to support software architects through design, analysis and code generation of systems that evolve structurally during execution. We used formal methods and tools for architectural modelling, graph rewriting and model checking. Starting from a set of valid DSA configurations, we may automatically generate corresponding JAVA code. We successfully applied this to a case study from the automotive domain. The architectural dynamism we considered is limited to programmed dynamicity, in which all changes (like adding/removing components) are defined before runtime execution and triggered by the system.

In the future we aim to study how to extend the **red** phase to introduce repairing and ad-hoc dynamism, and to prove associated properties. Regarding code generation we aim to introduce aspects like reconfiguration rules in the JAVA code, modelled by graph grammar rules, to strongly relate the code to programmed dynamic reconfiguration. Finally, to automate the process we need an easy-to-use framework to automatically execute each step. ALLOY and ARCHJAVA are implemented in JAVA, while UMC’s core is a stand-alone ADA program. Wrapped into appropriate JAVA classes, this core can be transformed into a plug-in for the ECLIPSE environment. We thus aim to implement a plug-in-based framework to fully integrate the three tools into an automated process.

ACKNOWLEDGMENT

Work funded by EU projects FP6-IP Sensoria and FP7-NoE S-Cube, and by the Italian project MIUR-PRIN 2007 D-ASAP.

REFERENCES

- [1] ARCHJAVA, <http://archjava.org/>.
- [2] M.H. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti, *An action/state-based model-checking approach for the analysis of communication protocols for SOC*, in: *FMICS*, LNCS **4916**, 2008, 133–148.
- [3] M.H. ter Beek, S. Gnesi, N. Koch, F. Mazzanti, *Formal verification of an automotive scenario in SOC*, in: *ICSE*, 2008, 613–622.
- [4] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, A. Lluch Lafuente, *Graph-based design and analysis of dynamic software architectures*, in: *Concurrency, Graphs and Models*, LNCS **5065**, 2008, 37–56.
- [5] R. Bruni, A. Bucchiarone, S. Gnesi, H. Melgratti, *Modelling dynamic software architectures using typed graph grammars*, in: *GT-VC*, ENTCS **213**, 2008, 39–53.
- [6] A. Bucchiarone, “Dynamic software architectures for global computing systems,” Ph.D. thesis, IMT Institute for Advanced Studies, Lucca, 2008.
- [7] A. Bucchiarone, D. Di Ruscio, H. Muccini, P. Pelliccione, *From requirements to Java code: An architecture-centric approach for producing quality systems*, in: *Model-driven software development*, IGI global, 2008.
- [8] A. Bucchiarone, J.P. Galeotti, *Dynamic software architectures verification using DynAlloy*, in: *GT-VMT*, ECEASST **10**, 2008.
- [9] EU project Sensoria, <http://www.sensoria-ist.eu/>.
- [10] D. Jackson, *Alloy: A lightweight object modelling notation*, TOSEM **11** (2002), 256–290.
- [11] RPS Code, <http://www.antonioBucchiarone.it/code/RPSCode.zip>.
- [12] UMC model checker, <http://fmt.isti.cnr.it/umc/>.