Chapter 13
Using Afra in Different Domains by Tool Orchestration

Ehsan Khamespanah, Pavle Mrvaljevic, Anas Fattouh, and Marjan Sirjani

Abstract The formal modelling and verification of distributed systems represents a complex process in which multiple tools are involved. Rebeca is a language which is developed to make modelling and verification of distributed systems with asynchronous message passing easier. This chapter shows how different tool orchestration methods are used for developing different verification engines for Rebeca models. As the first step, the way of enabling performance evaluation for Rebeca models is shown. To this end, state spaces which are generated for Rebeca models are transformed to the input of a third party tool and the result of the verification is given to the modeller. The second one is developing a search-based optimisation for wireless sensors and actuators applications. Running the model checker in a loop with different input parameters helps in finding the optimum values for parameters with respect to a given optimisation goal. The third one is for safety verification and performance evaluation of collaborative autonomous machines of Volvo car. The verification is done through developing and evaluating models by the model checking tool and Volvo car simulator (VCE Simulator).

This case-study chapter illustrates concepts introduced in Chap. 5 and addresses Challenge 2 in Chap. 3 of this book.

13.1 Introduction

Rebeca is a modelling language which is developed based on Hewitt and Agha's actors [AH87]. The actor model is a well-known model for the development of highly available and high-performance concurrent applications. It benefits from the universal primitives of concurrent computation, called actors. Hewitt introduced the

E. Khamespanah (✉)
University of Tehran, Tehran, Iran
e-mail: e.khamespanah@ut.ac.ir

P. Mrvaljevic · A. Fattouh · M. Sirjani
Mälardalen University, Västerås, Sweden
e-mail: pmc19001@student.mdh.se; anas.fattouh@mdh.se; marjan.sirjani@mdh.se

R. Heinrich et al. (eds.), *Composing Model-Based Analysis Tools*,
https://doi.org/10.1007/978-3-030-81915-6_13

actor model as an agent-based language [Hew72] and is later developed by Agha as a mathematical model of concurrent computation [Agh]. Actors in Rebeca are independent units of concurrently running programs that communicate with each other through message passing. The message passing is an asynchronous non-blocking call to the actor's corresponding message server. Message servers are methods of the actor that specify the reaction of the actor to its corresponding received message. In the Java-like syntax of Rebeca, actors are instantiated from reactive class definitions that are similar to the concept of classes in Java. Actors in this sense can be assumed as objects in Java. Each reactive class declares a set of state variables and the messages to which it can respond.

Rebeca is usable for software engineers and programmers. They are familiar with the Java-like syntax of Rebeca, and with the object-oriented style of programming. For concurrent programming, programmers are mostly using thread-based programming, and the event-based model of computation may not be as widely used by all the programmers. Usually it would be enough to tell them that each actor is one thread of execution, and message servers run atomically with no preemption. Different extensions for Rebeca are proposed to make it more usable for different domains and types of analysis. Timed Rebeca [Rey+14] is an extension on Rebeca with time features which supports modelling and verification of time-critical systems [KKS18, Kha+15, SK16]. Probabilistic Rebeca is another extension of Rebeca which is developed to consider the probabilistic behaviour of actor systems [VK12]. *Probabilistic Timed Rebeca* (PTRebeca) is an extension of Rebeca which benefits from modelling features of Timed Rebeca and Probabilistic Rebeca, combining the syntax of both languages [Jaf+14]. Inheritance for Rebeca is introduced in [You+20] to make modelling easier and enable modellers to define custom communication mechanisms.

Afra is a toolset which is developed for the purpose of providing modelling and verification facilities for Rebeca models and its extensions. Similar to many other Eclipse plugins, Afra contains a set of Eclipse views and editors together with a set of Java components for implementing models and analysing them. Considering the tool orchestration strategies which are presented in Chap. 5 of this book [Hei+21], this chapter shows how Afra is used together with other tools and libraries for the analysis of Rebeca models. We explain how orchestration of Afra with other tools is used in various domains for different purposes like model checking, performance evaluation, or search-based optimisation. Chapter 5 of this book [Hei+21] proposed a reference architecture along with important concepts that can be used to orchestrate analysis tools. Among six different strategies, single analysis orchestration (strategy A), cooperating analysis orchestration (strategy D), and sequential analysis orchestration (strategy E) are used in analysis tools of Rebeca. Single analysis orchestration uses a tool driver to translate the model into a valid input for an external black-box analysis tool. Then the modelling environment translates back the result of the analysis tool by using the tool driver again. Using sequential analysis orchestration the modelling environment invokes one tool, then translates the result into an input to another tool, and then translates the results of the second tool back to the domain-specific model to provide it to the domain expert.

In cooperating analysis orchestration the modelling environment invokes one tool, then translates the result into an input to another tool, and then translates the results of the second tool back into an input of the first tool to run another analysis.

In the rest of this chapter, first, Sect. 13.2 introduces Rebeca modelling language and how correctness properties are defined for Rebeca models using a running example. The main features of Afra are presented in Sect. 13.3. The next four sections show how orchestration of Afra with other tools is used to develop new analysis tools, i.e., Sect. 13.4 for performance evaluation, Sect. 13.5 for schedulability analysis, and Sects. 13.6 and 13.7 for flow management. Finally, Sect. 13.8 concludes the chapter.

13.2 Reactive Object Language (Rebeca)

We illustrate the Rebeca language with the example of a simple ticket service system. The actor model of this system is presented in Fig. 13.1. The model consists of three actors: *Customer*, *Agent*, and *Ticket Service System*. *Customer* asks *Agent* for issuing a ticket. The *Agent* actor forwards the request to *Ticket Service System* and it replies to *Agent* by sending a *ticket is issued* response. *Agent* responds to *Customer* by sending the issued ticket information. A Rebeca model has reactive objects with no shared variables, asynchronous message passing with no blocking send and no explicit receive, and unbounded buffers for messages. Objects in Rebeca are reactive and self-contained. Communication takes place by message passing among actors. The unbounded buffer of actors, called message *queue*, is used to store its arriving messages. Actor takes a message—that can be considered as an event—from the top of its message queue and executes its corresponding message server (also called a method). The execution of a message server is atomic which means that there is no way to preempt the execution of a message server of an actor and start executing another message server of that actor.

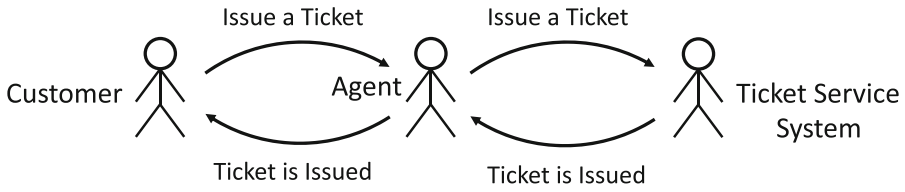

Fig. 13.1 The actor model of Ticket Service System

```
1  reactiveclass TicketService (3) {
2    knownrebecs {Agent a;}
3    statevars {
4      int nextId;
5    }
6    TicketService() {
7      nextId = 0;
8    }
9    msgsrv requestTicket() {
10     delay(?(0.4:2, 0.6:3));
11     a.ticketIssued(nextId);
12     nextId = nextId + 1;
13   }
14 }
15 reactiveclass Agent (2) {
16   knownrebecs {
17     TicketService ts;
18     Customer c;
19   }
20   msgsrv requestTicket() {
21     delay(1);
22     ts.requestTicket() deadline(5);
23   }
24   msgsrv ticketIssued(byte id) {
25     c.ticketIssued(id);
26   }
27 }
28 reactiveclass Customer (2) {
29   knownrebecs {Agent a;}
30   statevars {
31     boolean waiting;
32   }
33   Customer() {
34     self.try();
35     waiting = false;
36   }
37   msgsrv try() {
38     waiting = true;
39     a.requestTicket();
40   }
41   msgsrv ticketIssued(byte id) {
42     waiting = false;
43     self.try() after(30);
44   }
45 }
46 main {
47   Agent a(ts, c):();
48   TicketService ts(a):();
49   Customer c(a):();
50 }
```

Listing 13.1 The Rebeca model of ticket service system

Listing 13.1 shows the Rebeca model of the ticket service system of Fig. 13.1. A Rebeca model consists of a set of reactive classes (i.e., actor types) and the main block. In the main block, actors which are instances of the reactive classes are declared (lines 47–49). The body of the reactive class includes the declaration of its known actors, state variables, and message servers. For the case of `Customer` reactive class, its only known actor is an `Agent` which is accessible by variable `a` (line 29). As declared in line 31, `Customer` has one state variable which shows that and actor is sent a request and waits for the response. It also has two message servers `try` and `ticketIssued` and one constructor (line 33). Message servers consist of the declaration of formal parameters (e.g., `id` in line 41) and the body of the message server. The statements in the body can be assignments (line 38), conditional statements, enumerated loops, nondeterministic assignment, and method calls (line 39). Method calls are sending asynchronous messages to other actors (or to itself).

A reactive class has an argument of type integer denoting the maximum size of its message queue (e.g., 2 for `Customer` as depicted in line 28). Although message queues are unbounded in the semantics of Rebeca, to ensure that the state space is finite, we need a user-specified upper bound for the queue size. The operational semantics of Rebeca has been introduced in [Sir+04] in more detail. In comparison

```
1 property {
2   define {
3     waiting = c.waiting == true;
4   }
5   LTL {
6     NoStarvation : G(waiting -> F(!waiting));
7   }
8 }
```

Listing 13.2 The property file for the Rebeca code in Listing 13.1 stating the safety property as an LTL formula

with the standard actor model, dynamic creation and dynamic topology are not supported by Rebeca. Also, actors in Rebeca are single-threaded.

A Rebeca code can be model checked against a given set of *linear temporal logic* (LTL) properties. These properties specify the correct behaviours/states of the model. For example, in the case of Ticket Service System, one correctness property is that there is no starvation in issuing tickets for customers. This property can be specified in LTL using $\Box(waiting \rightarrow \Diamond(\neg waiting))$ formula which means that now and forever in the future, waiting for a ticket results in not waiting for a ticket (having ticket) eventually in the future.

Listing 13.2 shows how the mentioned LTL property is specified in the Rebeca property file. At the first step, the atomic propositions of the formula are defined in the `define` section of a Rebeca property file, considering the state variables of the actors (line 3). The name of the atomic propositions is set to `waiting` and its corresponding formula is put after the equal sign. In the `LTL` section, the correctness property is specified (line 5). In this example, only one property with the name `NoStarvation` is defined. Textual presentation of LTL modality $\Box$ (now and forever in the future) is `G` and $\Diamond$ (eventually in the future) is `F` in Rebeca property files.

Timed Rebeca [Rey+14] is an extension on Rebeca with time features for modelling and verification of time-critical systems. To this end, three primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events*. In a Timed Rebeca model, each actor has its own local clock and the local clocks evolve uniformly. Methods are still executed atomically, however passing time while executing a method can be modelled. In addition, instead of a queue for messages, there is a bag of messages for each actor.

The timing primitives that are added to the syntax of Rebeca are *delay*, *deadline*, and *after*. The *delay* statement models the passing of time for an actor during execution of a message server. The keywords *after* and *deadline* can only be used in conjunction with a method call. The value of the argument of *after* shows how long it takes for the message to be delivered to its receiver. The *deadline* shows the timeout for the message, i.e., how long it will stay valid.

As shown in line 21 of the model of Listing 13.1, processing time of a request in the agent is one time unit. At line 22 the actor instantiated from *Agent* sends a message *requestTicket* to actor *ts* instantiated from *TicketService*, and gives a deadline of five to the receiver to take this message and start serving it. The periodic task of retrying for a new ticket is modelled in line 39 by the customer sending a *try* message to itself and letting the receiver to take it from its bag only after 30 units of time (by stating *after(30))*. Model checker of Timed Rebeca models considers schedulability of message servers. It means schedulability is preserved if none of the specified deadlines of messages is missed.

PTRebeca language supports modelling and verification of real-time systems with probabilistic behaviours [Jaf+16]. PTRebeca introduced probabilistic assignment which is similar to nondeterministic assignment but associate a probability with each value option. In the probabilistic assignment, probabilities are real values between 0 and 1, and sum up to 1. Notably, by using probabilistic assignments, the values of the timing constructs (delay, after, and deadline) can also become probabilistic.

Different probabilistic behaviours can be modelled using the PTRebeca language, depending on the system under study. In the Rebeca code of Listing 13.1, issue time of a ticket in the ticket service system is set to two with the probability of 0.4 and three with the probability of 0.6 (line 10). Finding the expected value of the waiting time for issuing a ticket or computing the probability of deadline misses are two examples of probabilistic analysis which can be done using PTRebeca.

13.3 Afra

Afra is the *integrated development environment* (IDE) for model checking Rebeca and Timed Rebeca models.[1] It is developed as an Eclipse plugin and released as a standalone Eclipse product. It contains a set of Eclipse views and editors together with four Java components for implementing models and analysing them. Afra plugin contains a compiler component for compiling its given models and the *Rebeca model checker* (RMC) component for generating model checking codes for models. Using Afra, syntactically and semantically correct Rebeca models are transformed into a set of C++ source codes which generate the transition system of the model and perform property checking. In other words, running the generated C++ codes provides the model checking result. The working environment of Afra is shown in Fig. 13.2 which contains project explorer, Rebeca code editor, analysis result viewer, and counterexample viewer.

In addition to using Afra for the analysis of Rebeca models, its internal components (i.e., shown in Fig. 13.3) can be orchestrated in collaboration with other components to provide more comprehensive analysis solutions. Having an explicit

[1] Afra can be downloaded from http://rebeca-lang.org/alltools/Afra.

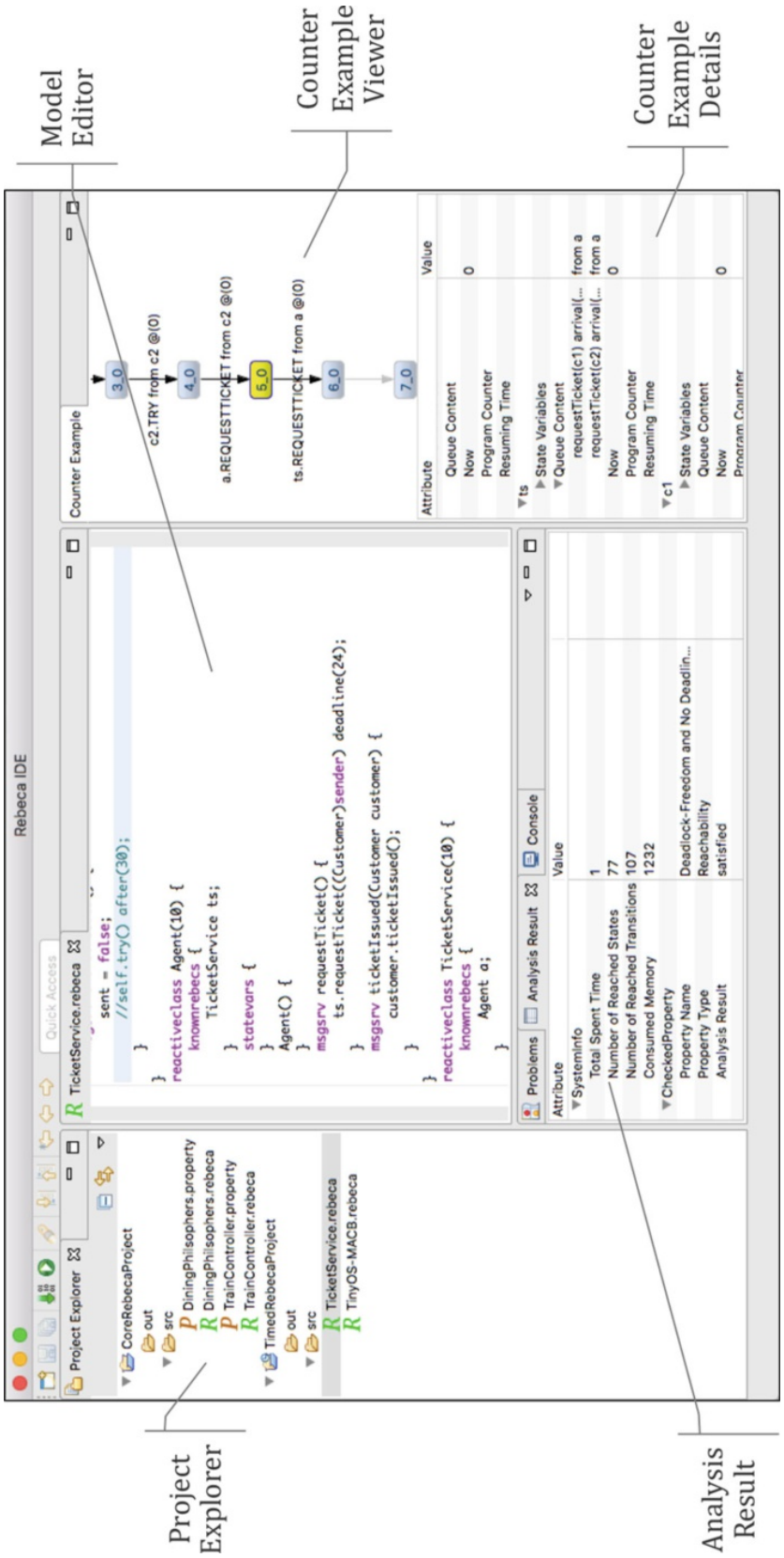

Fig. 13.2 Afra development environment

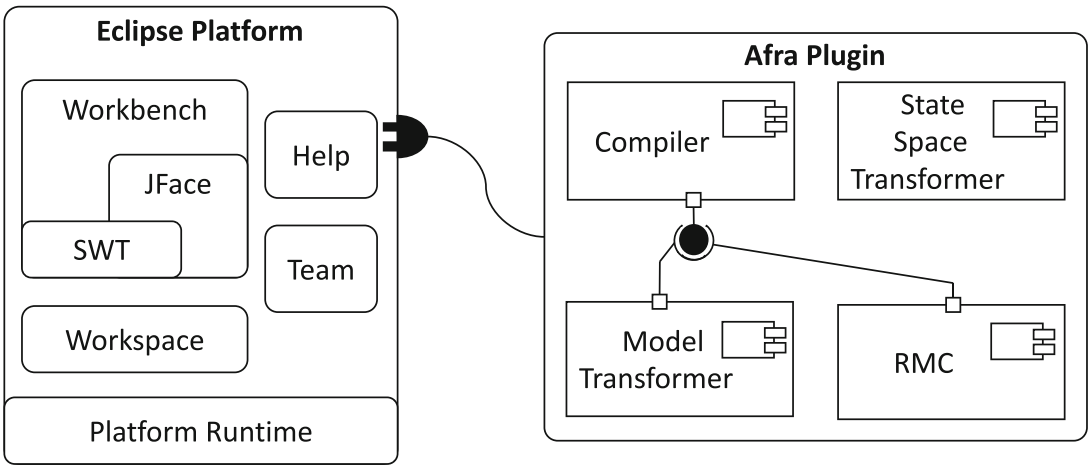

Fig. 13.3 Afra tool architecture

output language in the form of a standard *abstract syntax tree* (AST), the result of the *Compiler* component can be consumed by other components and tools. Using sequential orchestration of the components, the *RMC* component transforms this AST to C++ codes and the *Model Transformer* component transforms it to Real-Time Maude [ÖM07] (for bounded model checking), ROS [Qui+09] (for deploying in autonomous robots), and Akka [Akk09] (for running on Java Virtual Machine). Real-Time Maude is a rewriting-logic-based language which supports the formal specification and analysis of real-time systems. *Robotic Operating System* (ROS) is a robot middleware which has been widely used as an open source framework for the development of robotic applications and has become a standard in academic and industrial environments. Akka is a toolkit for building distributed, highly concurrent and event driven implementation of Hewitt's Actor Model on JVM.

As we will show in the following sections, the majority of analysis orchestrations for Rebeca models are realised by the analysis of the state space of models. By running the C++ codes which are generated by *RMC*, the state space of the model is generated together with applying model checking algorithms. This state space is presented in the XML format and can be used for further analysis, including third party applications or the Rebeca *State Space Transformer* and *Model Transformer* components. How the State Space Analysis tool is used is explained further in this section.

13.4 PTRebeca Model Checking: Sequential Analysis Orchestration

In the model checking of PTRebeca we have **Sequential Analysis Orchestration** strategy of RMC and IMCA (strategy E), shown in Fig. 13.4. *interactive Markov chain analyzer* (IMCA) is a tool for the quantitative analysis of interactive Markov

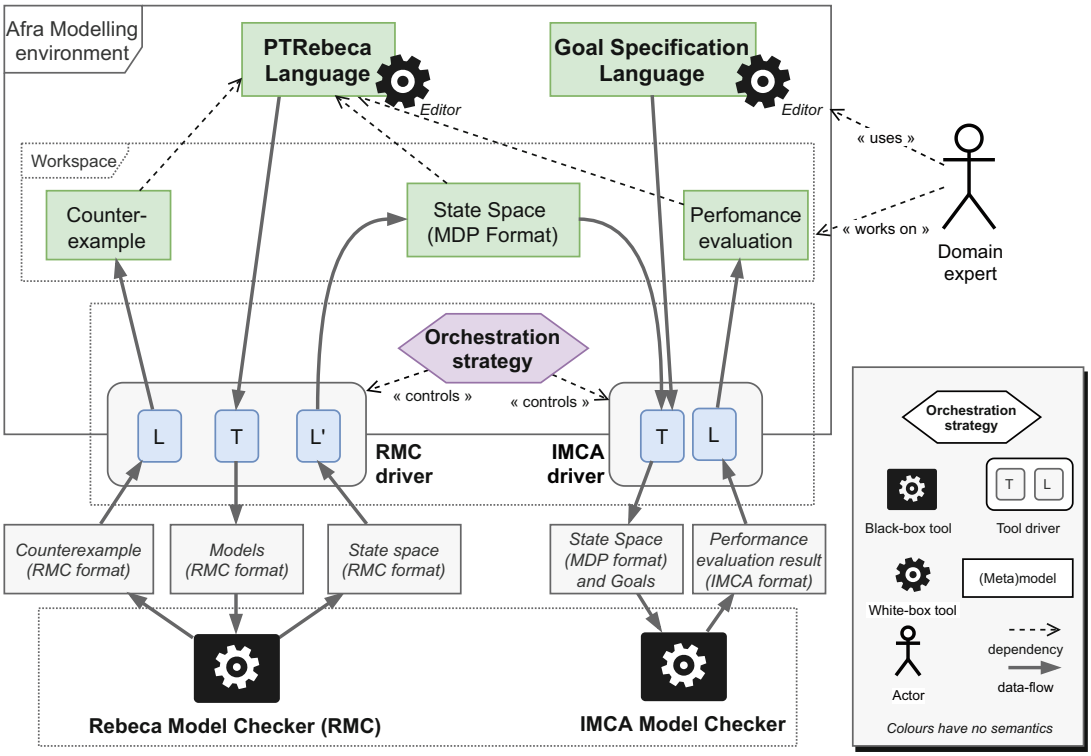

Fig. 13.4 Orchestration of tools and components for the analysis of Probabilistic Timed Rebeca models using IMCA tool

chains. In particular, it supports the verification of interactive Markov chains against reachability objectives, timed reachability objectives, expected time objectives, expected step objectives, and long-run average objectives. Figure 13.4 is developed based on the reference architecture for the integration of analysis tools in Chap. 5 of this book [Hei+21]. In this figure, Afra modelling environment is responsible for both, interacting with analysis tools and interacting with the domain expert. The modelling environment comprises four components: (a) the *domain-specific modelling languages* (DSMLs), (b) a set of tools to create, manipulate, or verify models conforming to these DSMLs, (c) a set of orchestration strategies to manage the interaction with and combination of analysis tools, and (d) the tool drivers that are responsible for actually interacting with the specific analysis tools.

As shown in Fig. 13.4, the modelling environment invokes RMC analysis tool then translates the result into the IMCA [Guc+12] input for performing additional performance analysis. In this case, the modelling DSML is the PTRebeca language and the input PTRebeca model is directly fed to RMC, and running the resulting C++ file, generates the state space of the given model in *time-dependent Markov decision process* (TMDP) format. The *IMCA driver* tool is developed using *State Space Transformer* component to convert the XML file of the TMDP of the model to the input language of IMCA model checker. It also uses the specification of the *goal states* of the model to generate one Markov automaton as the input of the IMCA model checking tool.

Note that the output of this tool as the malfunctioning which is detected in the model checking phase (i.e., RMC counterexample) and property violation in performance evaluation are lifted to Afra IDE viewer format to be usable for the domain expert.

13.5 Schedulability Analysis and Optimisation: Cooperating Analysis Orchestration

Orchestrating Afra components with some searching scripts for performing search-based optimisation is used in the analysis of *wireless sensor and actuator networks* (WSANs) applications. WSANs provide low-cost continuous monitoring but require dealing with the complexity of concurrent and distributed programming, networking, real-time requirements, and power constraints. So, it is hard to find a configuration that satisfies these constraints while optimising resource use. In [Kha+18] we build a script for search-based optimisation using schedulability analysis of Afra. This script computes the maximum sampling rate that nodes of WSAN can collect data from the environment without saturating the communication network and missing deadlines of their internal tasks.

The characteristics of real-time variants of the actor model make them appropriate for using as the DSML of WSAN applications: many concurrent processes with interdependent real-time deadlines. Considering the specification of the WSAN applications, there are many nodes which have the role of data acquisition and data transmission. For data acquisition, nodes have different sensors which periodically acquire data from the environment and send the data to the processing unit of the node. The processing unit validates the data and sends it to a central node using a wireless communication device, which is another actor of the model. As shown in [Kha+18], the node-level Timed Rebeca [Rey+14, KKS18] model of a WSAN application is developed to check for the possibility of deadline violations. Specifically, by changing the timing parameters of the model, the maximum safe sampling rate in the presence of other (miscellaneous) tasks in the node is found. Composing the models of standalone nodes to have a multi-node model requires that the wireless communication protocol is implemented for radio communication devices. Changing the configuration of the network and timing parameters of the model, the new maximum safe sampling rate is found. This optimisation of the sampling rate is implemented by the search-based optimisation technique.

Assigning different values for the parameters of the model, different maximum sampling rates are achieved as the result of the optimisation problem, shown by 3D surfaces in [Kha+18]. This requires running the model checker of Rebeca multiple times and integrating the result. To this end, we developed a script which runs the given model using different configurations to solve the optimisation problem. The script assigns different values for the maximum transmission time of the network,

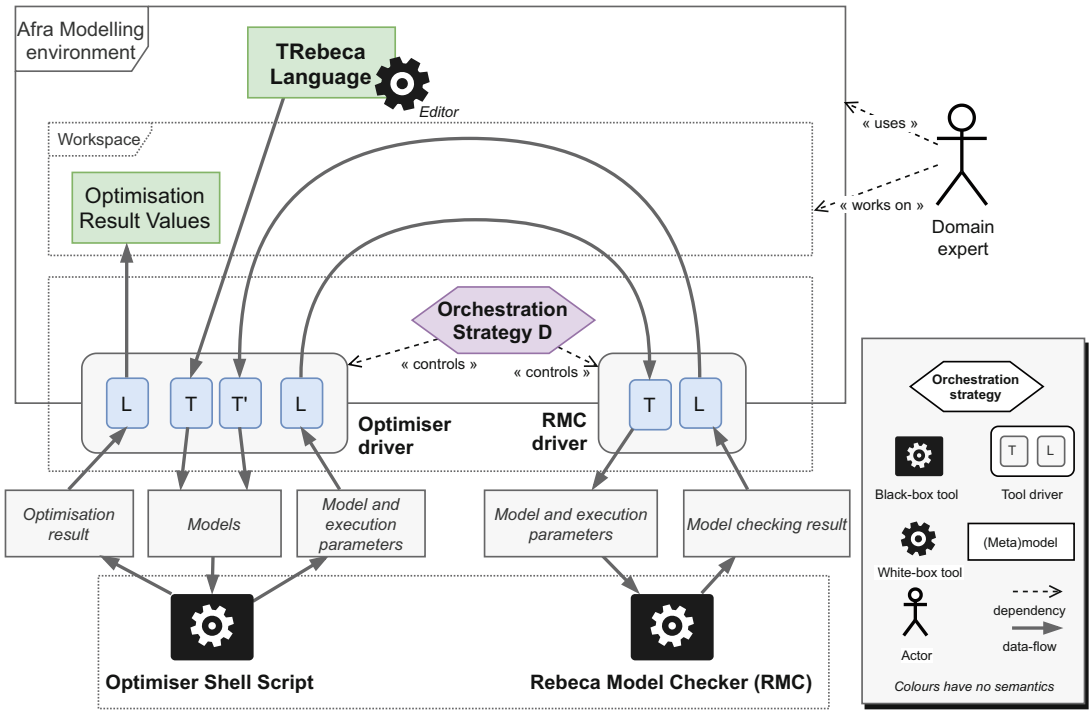

Fig. 13.5 Orchestration of tools and components for the schedulability analysis and optimisation of WSAN models

delay of sensors, the number of nodes in the system, the network packet size, etc. The orchestration of the tools for this problem is shown in Fig. 13.5.

The strategy of orchestration between the optimiser script and *RMC* component is **Cooperating Analysis Orchestration** (strategy D) as the result of the model checking part (RMC) is lifted to be given as the feedback to the optimiser. The result of model checking has to be lifted and transformed to the input of the *Optimiser* and the values which are generated by the *Optimiser* have to be transformed to the input of RMC, which are done by the *Optimiser* and *RMC* drivers, using simple text processing shell scripts. Note that in this tool, Timed Rebeca is the DSML for specifying input models.

13.6 Flow Management: Nested Analysis Orchestrations

AdaptiveFlow [Sir+19, For+20] is an actor-based framework which is used for track-based flow management. There are different track-based flow management systems such as warehouse management systems and transportation systems which play a crucial role in our daily life. All of these systems include a set of moving objects which travel on predefined tracks, e.g., trains on rails, cars on roads, automated vehicles in aisles of a warehouse, and airplanes in predefined airspace-tracks. In this view, the flowing entities move around some environments to transport some assets between some points of interest. AdaptiveFlow as a formal framework provides a

common abstraction for movement scenarios in these systems, and utilises model checking for safety checking and performance evaluation of models. Additionally, AdaptiveFlow allows the designer to specify policies for adapting the system behaviour with respect to possible changes in the environment. Sudden changes like blocking of a track, or change of a point of interest like a charging station being out of order can be modelled, too [For+20].

In the AdaptiveFlow framework, the DSML that is used for the specification of the model is in XML format and is given in three different files. The `environment.xml` file defines the base layer of the environment of the system, as a matrix. The layer is split into segments, each is surrounded by neighbouring segments and each neighbouring segment is labelled based on the location relative to the current one (i.e., NE-northeast, SW-southwest, E-east, etc.). The location of *point of interests* (PoIs) in the environment is defined in the `topology.xml` file. The PoIs can be perceived as key spots on which tasks are executed and are specified by unique identifiers, x and y coordinates, type of the point, and operating time. As the third input of AdaptiveFlow, the system configuration is specified in the `configuration.xml` file which includes information such as the number of moving objects, re-sending periods for requests, safe distance between moving objects, etc. The specification of moving objects and their properties are given to AdaptiveFlow using `configuration.xml`. Each moving object has a list of tasks IDs that are assigned to it, together with its attributes: unique identifiers, machine type, leaving time from parking station, fuel capacity, CO2 emission, etc.

One round of AdaptiveFlow workflow is split into three phases, shown in Fig. 13.6. The initial phase is the pre-processing phase in which different Timed Rebeca models are generated based on the XML input files (using a Python script) by running the *model generator* script. The second phase consists of formal verification of the generated model by generating the state space [Sir+19]. This verification is performed with model checking tools such as Afra or RMC [Sir+19]. These tools convert Rebeca models to C++ files which are afterwards compiled to an executable file [Sir+19]. Aside from checking regular properties such as deadlock-freedom and safety, AdaptiveFlow also verifies properties like fuel consumption of

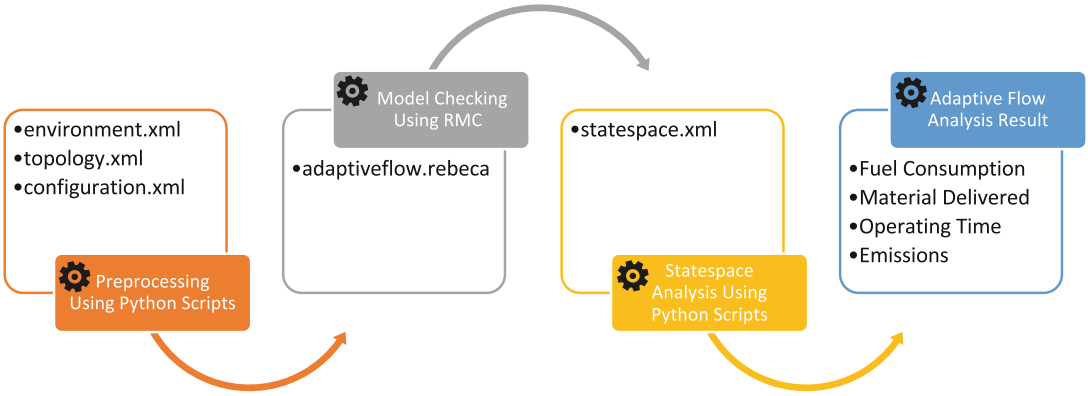

Fig. 13.6 AdaptiveFlow workflow, after [Sir+19]

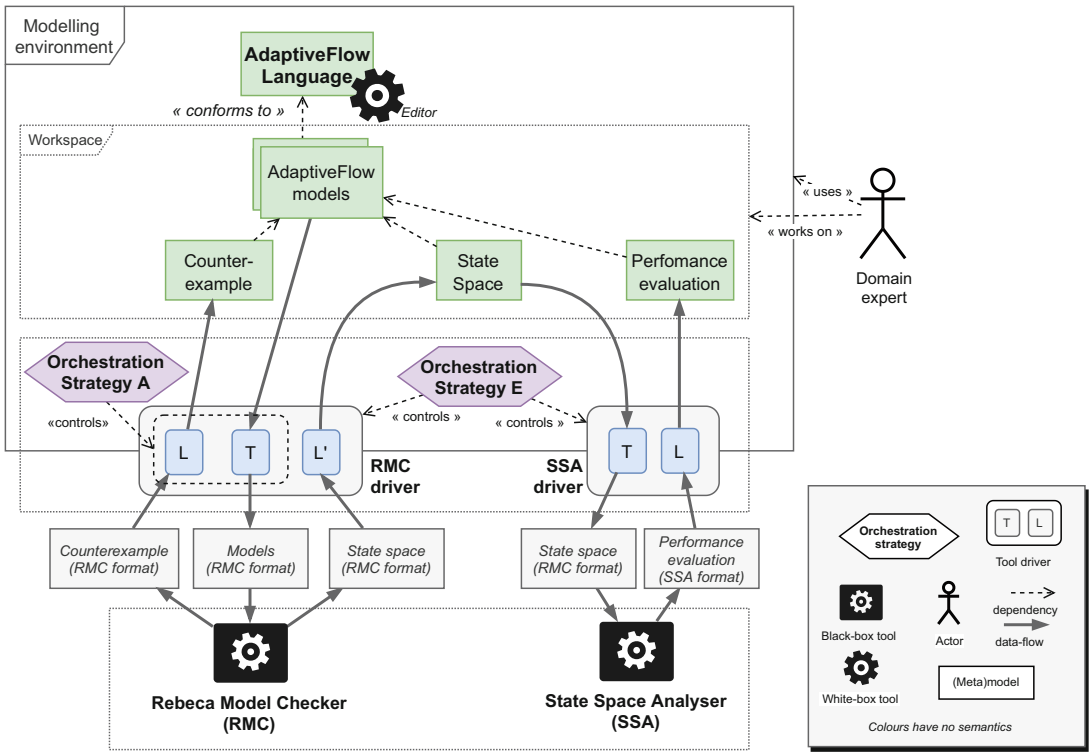

Fig. 13.7 Orchestration of tools and components for AdaptiveFlow (Presented in Fig. 5.4)

machines, correct machine movement, absence of machine collision with obstacles, and no-starvation property [For+20]. Reviewing counterexamples and fixing model errors iteratively, a modeller can develop functionally correct models. As mentioned before, model checking tools also generate state space of models used for the final, post-processing phase. In this phase, a state space which is generated from a functionally correct model goes through the Python script that analyses each state. The state space file, generated by RMC, is analysed with a Python script that extracts the evaluation of performance properties. The performance evaluation includes total CO2 emissions of machines, the amount of consumed fuel, moved material, and operating time of the collaborative system.

The orchestration of components for AdaptiveFlow is presented in Fig. 13.7 (Note that this figure is the same as Fig. 5.4 in Sect. 5.7 with some minor modifications). As shown in this figure, the orchestration strategy in AdaptiveFlow is nested orchestration strategies; a smaller cycle with **Single Analysis Orchestration** (strategy A) within a larger cycle of **Sequential Analysis Orchestration** (strategy E). The pre-processing python script of AdaptiveFlow works as the transformer component of RMC driver. The lifting component L of RMC driver translates counterexamples from XML format to Afra counterexample viewer format and L' only performs no modification to the state space file. On the other hand, *state space analyser* (SSA) driver components only feed and retrieve data to/from SSA components.

13.7 Safe Scenarios for Volvo CE Simulator: Sequential Analysis Orchestration

AdaptiveFlow can be used for the analysis of any track-based flow management system. In VMap project, AdaptiveFlow is extended to make it appropriate for the analysis of the behaviour of Volvo construction equipment, as an example of track-based flow management systems [Mrv20]. The *Volvo Construction Equipment Simulator*[2] (VCE Simulator) is a high-fidelity platform for simulating and testing Volvo construction equipment in a virtual environment. The simulator's core system is a distributed component-based system that is made up of several tasks. Each task has a single well-defined purpose and can communicate with other tasks by passing messages. The simulator is equipped with an editor that permits to create new scenarios. A scenario is a sequence of actions that are organised in tracks where tracks are executed in parallel and actions inside each track is executed in sequence. Scenarios are built manually for testing some properties of construction machines working on the desired environment or to measure the productivity of a working plan in a construction environment for example.

The orchestration of tools and components for VMap is shown in Fig. 13.8. In VMap, the iterative development of AdaptiveFlow is used to develop a correct model with an acceptable level of performance. Then, XMLs of AdaptiveFlow models are transformed to the VCE simulator input format for the simulation purpose. Finally, the results of the VCE simulator are given as feedback to the designer to improve the model. The scenario in the VCE simulator is described by an XML file, namely `dynamic.content`. The `dynamic.content` file contains a list of the objects inside the scenario, the components of each object with its properties, and the communication between the objects. It could also include links to objects defined in other files.

As a result, the orchestration strategy of AdaptiveFlow and VMap is **Sequential Analysis Orchestration** (strategy E). The transformer of VCE driver is responsible for transforming AdaptiveFlow specifications and other simulation-specific files to the `dynamic.content` format. The lifting component of VCE driver makes the simulation results human readable.

[2] VCE Simulator: https://www.volvoce.com/europe/en/services/volvo-services/productivity-services/volvo-simulators.

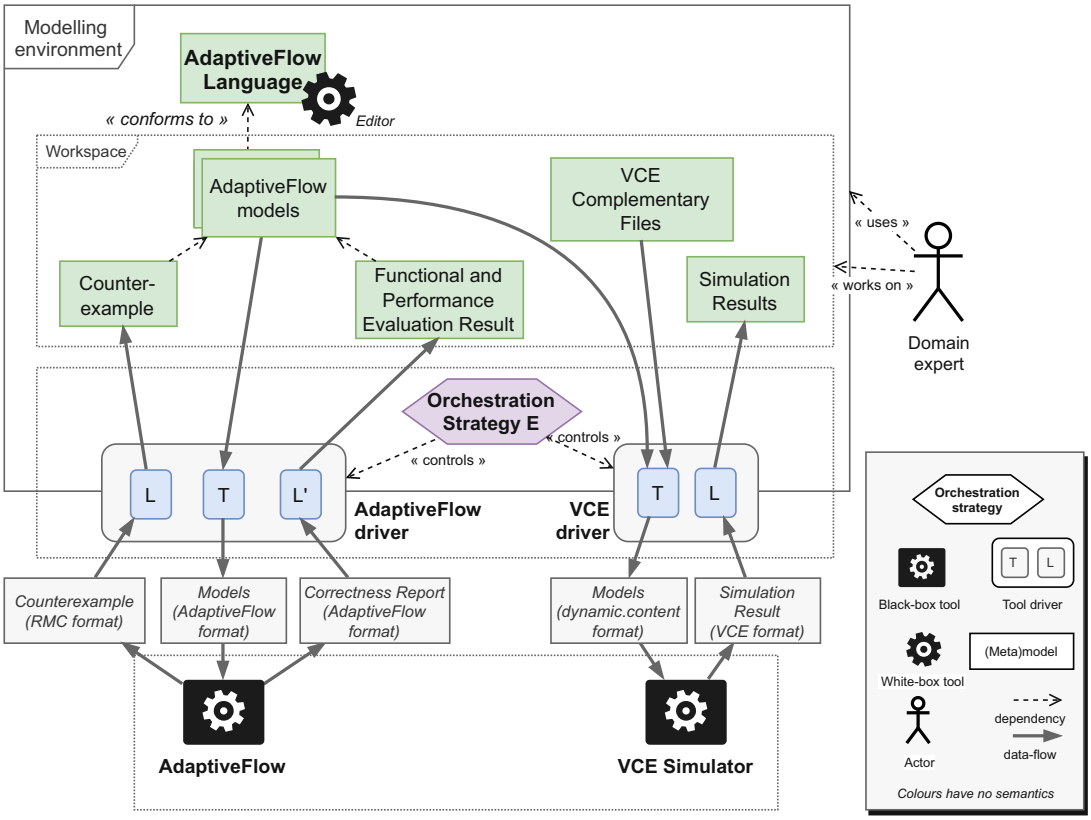

Fig. 13.8 Orchestration of tools and components for VMap

13.8 Conclusion

Different verification engines are developed for Rebeca models using orchestrations of a set of analysis tools. In Chap. 5 of this book [Hei+21], a catalogue of strategies for tools orchestration is proposed. For each of them, strategies name, explanation, and examples are proposed in a systematic way. We studied a few Rebeca analysis tools and classified them as one of the orchestration strategies presented in this catalogue.

Using the proposed patterns makes it easier to reuse the existing tools and put them together in different ways. This way, the future analysis tools of Rebeca will be developed easier and faster. Orchestration strategies also improved the documentation and maintenance of the existing verification engines by furnishing an explicit specification of tools interactions.

References

[Agh] Gul A. Agha. ACTORS - *A Model of Concurrent Computation in Distributed Systems*. MIT Press.

[AH87] Gul Agha and Carl Hewitt. “Concurrent programming using actors”. In: (1987), pp. 37–53.

[Akk09] Akka. *Typesafe, Inc. Akka*. 2009.

[For+20] Giorgio Forcina, Ali Sedaghatbaf, Stephan Baumgart, Ali Jafari, Ehsan Khamespanah, Pavle Mrvaljevic, and Marjan Sirjani. "Safe Design of Flow Management Systems Using Rebeca". In: *Journal of Information Processing* 28 (2020), pp. 588– 598. https://doi.org/10.2197/ipsjjip.28.588.

[Guc+12] Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer. "Quantitative timed analysis of interactive Markov chains". In: *4th international conference on NASA Formal Methods*. 2012, pp. 8–23.

[Hei+21] Robert Heinrich, Francisco Durán, Carolyn L. Talcott, and Steffen Zschaler (eds.) *Composing Model-Based Analysis Tools*. Springer, 2021. https://doi.org/10.1007/978-3-030-81915-6.

[Hew72] C. Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. MIT Artificial Intelligence Technical Report 258. Department of Computer Science, MIT, 1972.

[Jaf+14] Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, and Holger Hermanns. "Performance Analysis of Distributed and Asynchronous Systems using Probabilistic Timed Actors". In: (2014). http://journal.ub.tu-berlin.de/eceasst/article/view/984.

[Jaf+16] Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, Holger Hermanns, and Matteo Cimini. "PTRebeca: Modeling and analysis of distributed and asynchronous systems". In: *Science of Computer Programming* 128 (2016), pp. 22–50.

[Kha+15] Ehsan Khamespanah, Marjan Sirjani, Mahesh Viswanathan, and Ramtin Khosravi. "Floating Time Transition System: More Efficient Analysis of Timed Actors". In: *12th International Conference Formal Aspects of Component Software*. 2015, pp. 237–255. https://doi.org/10.1007/978-3-319-28934-2_13.

[Kha+18] Ehsan Khamespanah, Marjan Sirjani, Kirill Mechitov, and Gul Agha. "Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking". In: *International Journal on Software Tools for Technology Transfer* 20.5 (2018), pp. 547–561.

[KKS18] Ehsan Khamespanah, Ramtin Khosravi, and Marjan Sirjani. "An efficient TCTL model checking algorithm and a reduction technique for verification of timed actor models". In: *Science of Computer Programming* 153 (2018), pp. 1–29.

[Mrv20] Pavle Mrvaljevic. *Tool Orchestration for Modeling, Verification, and Analysis of Collaborating Autonomous Machines*. Master Thesis. Mälardalen University, 2020.

[ÖM07] Peter Csaba Ölveczky and José Meseguer. "Semantics and pragmatics of Real-Time Maude". In: *Higher-Order and Symbolic Computation* 20.1-2 (2007), pp. 161–196.

[Qui+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. "Workshop on Open Source Software in Robotics". In: vol. 3. 3.2. 2009, p. 5.

[Rey+14] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. "Modelling and Simulation of Asynchronous Real-Time Systems Using Timed Rebeca". In: *Science of Computer Programming* 89 (2014), pp. 41–68.

[Sir+04] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. "Modeling and Verification of Reactive Systems using Rebeca". In: *Fundamenta Informaticae* 63.4 (2004), pp. 385–410.

[Sir+19] Marjan Sirjani, Giorgio Forcina, Ali Jafari, Stephan Baumgart, Ehsan Khamespanah, and Ali Sedaghatbaf. "An Actor-Based Design Platform for System of Systems". In: *43rd IEEE Annual Computer Software and Applications Conference*. 2019, pp. 579–587. https://doi.org/10.1109/COMPSAC.2019.00089.

[SK16] Marjan Sirjani and Ehsan Khamespanah. "On Time Actors". In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. 2016, pp. 373–392. https://doi.org/10.1007/978-3-319-30734-3_25.

[VK12] Mahsa Varshosaz and Ramtin Khosravi. “Modeling and Verification of Probabilistic Actor Systems Using pRebeca”. In: *14th International Conference on Formal Engineering Methods*. 2012, pp. 135–150. https://doi.org/10.1007/978-3-642-34281-3_12.

[You+20] Farnaz Yousefi, Ehsan Khamespanah, Mohammed Gharib, Marjan Sirjani, and Ali Movaghar. “VeriVANca framework: verification of VANETs by property-based message passing of actors in Rebeca with inheritance”. In: *International Journal on Software Tools for Technology Transfer* 22.5 (2020), pp. 617–633. https://doi.org/10.1007/s10009-020-00579-8.