Integration and Orchestration of Analysis Tools

Robert Heinrich, Erwan Bousse, Sandro Koch, Arend Rensink, Elvinia Riccobene, Daniel Ratiu, and Marjan Sirjani

Abstract This chapter addresses the integration and orchestration of external analysis tools into modelling environments. We first give a detailed overview of the considered context and problem statement. Then, a solution in the form of a reference architecture for the integration of analysis tools into modelling environments is presented. We collect a set of requirements that analysis tools must satisfy in order to enable (a) the integration of these analyses into modelling environments and (b) the orchestration of these analysis tools to produce overall results. Finally, we give an overview of different orchestration strategies for the integration of analysis tools and show examples.

This core chapter addresses Challenge 2 introduced in Chap. 3 of this book (*the practical implications*—how to integrate and orchestrate existing analysis tools).

R. Heinrich (✉) · S. Koch
Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: robert.heinrich@kit.edu; sandro.koch@kit.edu

E. Bousse
University of Nantes, Nantes, France
e-mail: erwan.bousse@ls2n.fr

A. Rensink
University of Twente, Enschede, Netherlands
e-mail: arend rensink@utwente.nl

E. Riccobene
Università degli Studi di Milano, Milano, Italy
e-mail: elvinia riccobene@unimi.it

D. Ratiu
CARIAD, Wolfsburg, Germany
e-mail: ratiud@googlemail.com

M. Sirjani
Mälardalen University, Västerås, Sweden
e-mail: marjan.sirjani@mdh.se

5.1 Introduction

Sophisticated modelling environments, often based on the principles of *model-driven engineering* (MDE) and *software language engineering* (SLE), are becoming increasingly ubiquitous. More and more disciplines, may it be avionics, automotive, constructional engineering, automation engineering, or natural sciences, rely on such tools. These tools become all the more valuable if they provide deep insights into the correctness and fitness-for-purpose of the models[1] used and apply model-based analysis to forecast properties of the things to be built. At the same time there is a community of analysis tool builders who distil mathematical and logic experience into analysis tools (cf. Chap. 2 of this book [Hei+21]) that rely on formalisms such as *satisfiability modulo theories* (SMT) formulae, transition systems, or discrete-event systems. Many of these analysis tools can be used beneficially in the aforementioned modelling environments if they are suitably integrated. In practice this usually means that user-facing models must be translated to the input formalism of the analysis tool, and the result of the analysis must be lifted back to the domain level. In addition, there are many use cases like in portfolio solvers, model checkers, simulation coupling, model-based testing, and runtime verification where multiple existing analysis tools must be orchestrated to deliver value in the context of the modelling environment.

This chapter addresses the challenge of how to integrate and orchestrate external analysis tools into modelling environments. We first give a detailed description of the considered context and problem statement in Sect. 5.2. The state of the art of integrating and orchestrating model-based analysis tools is discussed in Sect. 5.3. Then, we provide a solution in the form of a reference architecture for the integration of analysis tools into modelling environments in Sect. 5.4. Based on our professional experience, both in academia and industry, with building and using modelling environments and integrating analysis tools into existing modelling environments, we collect a set of requirements in Sect. 5.5 that analysis tools must satisfy in order to enable (a) the integration of these analyses into modelling environments and (b) the orchestration of analysis tools to produce overall results. Tools that apply the reference architecture may adhere to different orchestration strategies. We give an overview of several orchestration strategies for the integration of analysis tools into modelling environments in Sect. 5.6 and show examples of existing tools to illustrate the application of these orchestration strategies in Sect. 5.7. The chapter concludes with a summary and outlook in Sect. 5.8.

[1] Note, while Chap. 2 of this book [Hei+21] postulates analysis input as three kinds of models—of system, of property, and of context—we stay with the term model in this chapter since a distinction of the kind of model is not relevant here.

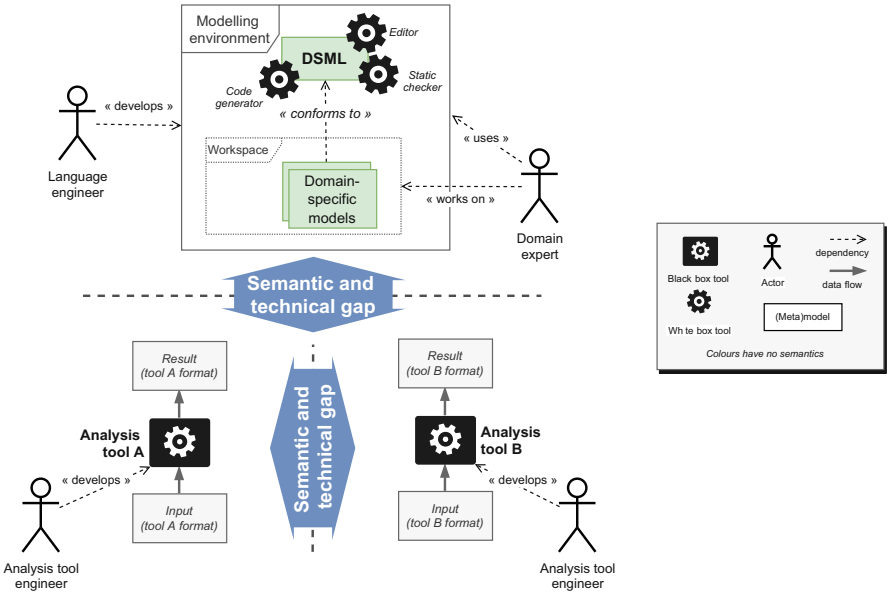

Fig. 5.1 Overview of the context and problem statement

5.2 Context and Problem Statement

To give a better global understanding of the focus of this chapter, Fig. 5.1 depicts the considered context and problem statement. First, at the top half of the figure, the key roles and concepts of a typical modelling process are shown. We assume that a *modelling environment* is used by the *domain expert* in order to work on one or multiple *domain-specific models*. While the models are being worked on by the domain expert, the models are stored in a *workspace* provided by the modelling environment. A classical modelling environment provides one or several[2] *domain-specific modelling languages* (DSMLs), along with a set of tools—editor, checker, code generator, etc.—to create, manipulate, or verify models conforming to these DSMLs. The development and maintenance of the modelling environment and the DSMLs it uses are taken care by one or several *language engineers*.

Then, at the bottom half of the figure, a common choice to gain insight[3] into the models is to rely on existing proven and powerful *analysis tools*, such as model checkers, solvers, or theorem provers. Analysis tools can even be expertly *combined* in order to bring more interesting, more complete, or faster results. In this chapter, we make the assumption that the considered analysis tools are *external*—

[2] Note that Figs. 5.1 and 5.2 only show a single DSML for better readability.

[3] We may be interested in insights into models such as correctness and well-formedness of models, or quality properties of the modelled system.

i.e., developed by different persons and communities than the ones involved in the modelling process shown in the top half—and *black-boxes*—i.e., they are taken off-the-shelf and their internals are not known. An analysis tool typically takes an *input* conforming to a specific input format, and produces a *result* in either a loose (e.g., raw textual description) or a well-defined format. Some tools may also simultaneously require multiple different sources of input (e.g., a configuration file and a model), produce multiple different result artefacts (e.g., a counterexample and the state space used to discover it), or may even function in an incremental fashion. Analysis tools are developed by *analysis tool engineers*, which are experts in the theories and techniques implemented in the tools.

In order to enable the use of single or combined external tools for the analysis of models created in the modelling environment, there are at least two compelling prerequisites that must be fulfilled. First, we call *tool integration* the problem of actually being able to make use of each separate analysis tool (i.e., exchange data, make queries, start and stop tasks, etc.) within the modelling process. For instance, using a model checker requires at least to be able to (1) send it the model and the property to be checked, (2) ask it to start the analysis, and (3) retrieve the result. Second, we call *tool orchestration* the problem of configuring *when* and *how* analysis tools should be used and/or combined in a considered modelling process, which includes how these analysis tools should interact with each other. For instance, it must be possible to drive a sequence of actions such as "give the model in a certain format to the model checker, start the analysis, get the counterexample, translate it to a second format, feed it to a second tool to replay the trace, translate the replay result back to the domain expert".

Unfortunately, both, in between a modelling environment and analysis tools, and in between analysis tools themselves, there are *semantic gaps*—i.e., differences between their semantics— and *technical gaps*—i.e., differences between the technical spaces where each environment and tool operates, such as runtime environments, *application programming interfaces* (APIs), or frameworks—to take into account. Consequently, there are many obstacles to overcome in order to solve the tool integration and tool orchestration problems, such as:

1. A model created in the modelling environment conforms to a DSML that may entirely differ from the input format expected by a given analysis tool, thus first requiring a *model transformation* to make the model understandable by the tool.
2. Since a given analysis tool is not aware of the DSML and of the domain of expertise of the modeller (i.e., the domain expert), the result it produces is likely to be written in "words" that the domain expert cannot easily understand, thus requiring a second model transformation to *lift* the low-level result back into a format fitting the domain of interest, and thus the domain expert.
3. When combining analysis tools, the input and output formats that they use are rarely compatible among themselves, and thus require model transformations as well.
4. Each analysis tool may expose a specific *interface* (e.g., Java API, command line interface, network socket, etc.) for programmatically interacting with it, and

possesses its own explicit or implicit *protocol* to use this interface (i.e., which sequences of actions provided by the interface are valid to achieve certain tasks).
5. The modelling environment and analysis tools may work in very different *technical spaces*, such as different data representations (e.g., graphs *vs.* trees), execution environments (e.g., Java *vs.* Python), or file formats (e.g., XMI *vs.* JSON). These differences add technical complexity over the task of defining sound transformations, both towards and from analysis tools.

All these concerns are rather well known, and have all been dealt with in the past in an ad hoc basis in a great number of modelling environments—AF3, ASMETA, mbeddr, or Palladio to name a few (all described in Sect. 5.3.2). However, to our knowledge, little work has been made to provide *general and systematic* answers that could help dealing with the integration and the orchestration of analysis tools. Hence, as an exploratory attempt to address this issue, we present the following three contributions in this chapter. First, we propose a *reference architecture*—along with important concepts—that can be used to methodically integrate and orchestrate analysis tools into a modelling process. Second, we propose a set of *requirements* that qualify which analysis tools can be properly integrated in such an architecture. Third and last, we propose and formalise a first set of *strategies* that can be used to answer common integration and orchestration cases, especially when multiple analysis tools are combined together to provide one or multiple results. These strategies are illustrated using a selection of real-world examples of existing ad hoc integrations and orchestrations of analysis tools.

5.3 State of the Art

This section provides a discussion of the state of the art of integrating and orchestrating model-based analysis tools before we propose our concepts in the sections that follow. We first give an overview of related research on integrating and orchestrating tools and then give examples of existing modelling environments with integrated analysis tools that may serve as inspiration and illustration for the concepts proposed in this chapter.

5.3.1 Research on Integrating and Orchestrating Tools

A first step to systematically deal with the integration and orchestration of black-box analysis tools is to define how to generically interact with tools. To this end, significant work has been done in different research communities to consider tools as *first-class entities*.

Two early endeavours from the late 1990s are ToolBus [BK96] and the *electronic tool integration* (ETI) platform [SMB97, BMW97] (with some extensions made in

the 2000s [MNS05, Mar05]). Both these approaches have assumptions and goals rather similar to what we stated in the previous section: Being able to integrate existing tools into foreign processes is an important challenge, which requires proper data exchange and communication mechanisms with said tools. These approaches already sketch important concepts such as tool adapters, type transformers, tool coordination, or coordination universe. However, these approaches try to tackle a more generic problem, as they make no assumptions on the context in which tools are integrated and combined. They notably do not discuss the problem of lifting analysis results to the domain of interest. While we do take inspiration from these early generic proposals, the present chapter specifically focuses on the integration and orchestration of analysis tools into a modelling environment. Moreover, our proposal also aims at providing a set of requirements for integrating analysis tools, along with a set of interesting re-usable strategies for orchestrating them.

In the 2000s, a slightly similar proposal was made, called Model Bus [BGS05]. In a pure MDE context, this approach aims at providing an environment where both a set of metamodels and a set of services built for these metamodels—such as model transformations and code generators—can be registered. These services can then easily be called and chained thanks to a communication bus called the Model Bus. This approach is mostly targeting MDE practitioners who need to organise a set of model manipulation services, and does not discuss the case of external tools, or the problem of lifting back analysis results.

More recently, some approaches solely focus on the problem of combining the analysis tools. Dwyer et al. [DE10] proposed a vision where tools can be combined using the notion of *evidence* as a pivotal concept. In other words, the authors advocate for a common representation and storage of analysis results, and means to compose these results in a meaningful way. Rather aligned with this vision, and following a proposal from Rushby [Rus05], Cruanes et al. [Cru+13] designed the *evidential tool bus* (ETB), a "distributed framework for integrating diverse tools into coherent workflows for producing claims supported by explicit evidence". While the approach is very interesting, and in some ways in the steps of ETI, it mostly focuses on the problem of storing and sharing analysis results between distributed formal analysis tools. Questions such as the lifting of results back to the domain, or how to soundly transform domain-specific models for analysis tools, or what common orchestration strategies can be used, are not considered.

5.3.2 *Examples of Modelling Environments with Integrated Analysis Tools*

In the following, we provide examples of modelling environments that integrate various external analysis tools. All our examples are based on open-source and

freely available environments. However, commercial environments (e.g., Simulink[4] and SCADE[5]) face the same challenges when integrating external analysis tools. These examples can be seen as existing ad hoc applications of the general concepts presented in this chapter.

AF3[6] [Ara+15] is an environment for modelling and specification of embedded systems. It offers support for modelling requirements, the logical and technical architectures and deployment. AF3 integrates NuSMV [Cim+02] for verifying models and Z3 [MB08] for generating optimal deployments.

ASMETA[7] [Arc+11, GRS08] (ASM mETAmodeling) is a modelling environment for the *abstract state machines* (ASMs) formal method. It is based on the integration of different tools for performing validation and verification activities on ASM models; it integrates different external analysis tools such as the NuSMV [Cim+02] model checker for performing property verification and SMT solvers to support correct model refinement verification [AGR16] and runtime verification [AGR14].

FASTEN[8] [RGS19] is a modelling environment for the specification and design of safety-critical systems. Regarding formal analyses, the main focus of FASTEN is to experiment with usability of formal specification and transition between informal to formal specifications. FASTEN integrates various external analysis tools such as NuSMV [Cim+02], Spin [Hol03], Z3 [MB08], and PRISM [KNP11].

mbeddr[9] is a modelling environment for the development of embedded systems. It integrates various formal analysis tools that work at model level as well as those that work on C code. Examples of model-level analyses are checking for consistency and completeness of decision tables [Rat+12a, Rat+12b] using Z3 [MB08]; examples of code-level analyses are checking assertions from C programs [Rat+13, MVR14] using CBMC [CKL04] or applying the model-driven code checking method [RU19] using Spin [Hol03].

OpenCert[10] is an integrated environment for specification and certification of *cyber-physical systems* (CPS). OpenCert uses modelling languages based on SysML [Obj12] and integrates the OCRA [CDT13] and NuXmv [Cav+14] formal verification tools for checking properties expressed using temporal logic.

Palladio is a tool-supported approach to modelling and analysing software architectures for various quality properties [Reu+16]. It integrates various analysis tools to predict and reason about these quality properties into a modelling environment.

[4] Simulink: https://www.mathworks.com/products/simulink.html.

[5] SCADE: https://www.ansys.com/products/embedded-software/ansys-scade-suite.

[6] AF3: https://download.fortiss.org/public/projects/af3/help/index.html.

[7] ASMETA: http://asmeta.sourceforge.net/.

[8] FASTEN: https://sites.google.com/site/fastenroot/home.

[9] mbeddr: http://mbeddr.com.

[10] OpenCert: https://www.eclipse.org/opencert/.

Details on the Palladio approach and the associated tooling are described in Chap. 11 of this book [Hei+21].

VCES[11] [GLO11] is an Eclipse-based environment for the modelling and analysis of software-intensive systems. It includes an implementation of a higher-level modelling language (named SAML—*system analysis and modelling language*) that is an intermediate, automata-based language between arbitrary high-level engineering languages like SysML [Obj12] and the input languages of analysis tools. VCES features model transformations from SAML to the input of verification tools like NuSMV [Cim+02] and PRISM [KNP11]. Results of the verification are lifted in the VCES *integrated development environment* (IDE) and presented in a user-friendly manner.

Why3[12] [FP13] is a platform for deductive program verification for the WhyML language. It integrates a wide range of both automatic and interactive external theorem provers (more than 19 as of today), and any prover can be chosen to perform any of the proofs. While Why3 is not a modelling environment per se—since WhyML is a programming language mostly used as an intermediate language to verify programs written in C, Java, or Ada—it directly deals with the problem of integrating and orchestrating a great number of homogeneous external tools, here using an abstraction layer dedicated to theorem provers.

TopCased[13] [Far+06] is an environment for critical applications and systems development, using modelling languages such as UML [Obj15], SysML [Obj12], or AADL [FGH06]. The environment relies on the Fiacre language [Ber+08] as an intermediary language to translate models to analysis tools—such as model checkers—and to lift verification results back to the domain expert. While TopCased is not maintained since 2013, it was one of the first successful attempts to bridge MDE and formal verification in a single environment.

5.4 A Reference Architecture for Integrating Analysis Tools

A *reference architecture* is known in software engineering as a general structure for applications in a particular domain, which may partially or fully implement the reference architecture [Som15]. We transferred the notion of a reference architecture to the problem of integrating analysis tools into modelling environments. The reference architecture for the integration of one or multiple analysis tools into a modelling environment is depicted in Fig. 5.2. Note, in the figure we depict two

[11] VCES: https://cse.cs.ovgu.de/cse/researchareas/vecs/.
[12] Why3: http://why3.lri.fr/.
[13] TopCased: http://www.topcased.org/.

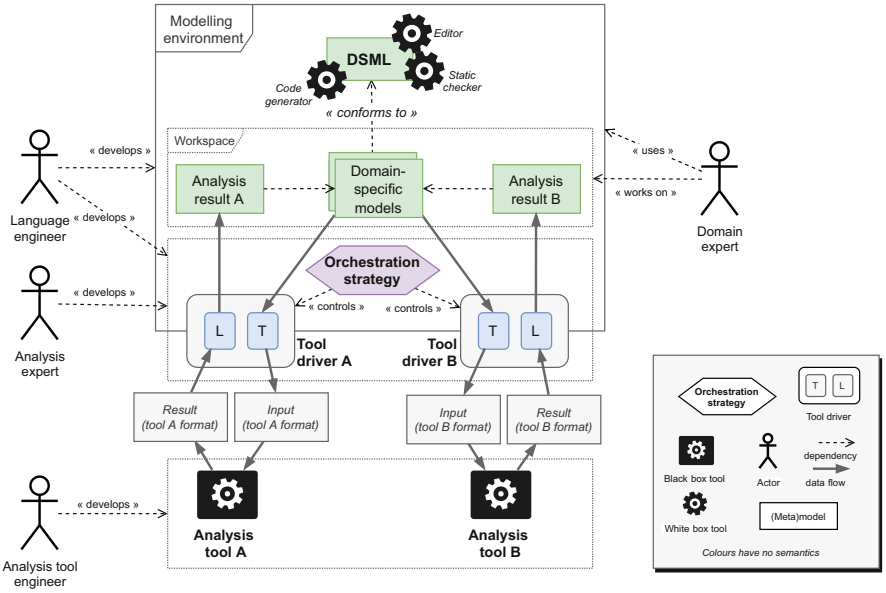

Fig. 5.2 Reference architecture for the integration of analysis tools into a modelling environment

analysis tools to indicate that multiple analysis tools can be integrated, while the number of analysis tools to be integrated is not limited.

The modelling environment is responsible for both, interacting with analysis tools and interacting with the domain expert wishing to perform analyses based on domain-specific models. The modelling environment comprises four components: (a) the DSMLs, (b) a set of tools—e.g., editors, checkers and code generators—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.

The modelling environment, the DSMLs it uses, and the tools to work on models, are developed and maintained by language engineers. Often, the development and maintenance of tools to work on models is supported by tool developers which are not depicted in the figure as they are not in the focus of this chapter. The modelling environment follows some *orchestration strategy* that defines which analysis tools should be used for a given analysis task, how these tools should be used for a given analysis task, in which order these tools should be used, and how the analysis results they produce should be combined or exchanged. Analysis tools are assumed to already exist and to have been created externally by analysis tool engineers to satisfy specific analysis tasks. The integration of analysis tools into the modelling environment is accomplished by a set of *tool drivers*, each tool driver being responsible for interacting with one external analysis tool. This includes how to use the interfaces of the analysis tool, how to translate a domain-specific model

used by domain experts in a valid input for the tool ($\top$ in Fig. 5.2), how to lift back the analysis result in a form that makes sense at the abstraction level of the domain-specific model ($\bot$ in Fig. 5.2),[14] as well as the protocol to exchange messages and information with the tool. Tool drivers, along with orchestration strategies that control them, are jointly developed by language engineers and *analysis experts*, who are versed in the analysis tools that must be integrated into the modelling environment.

The proposed reference architecture serves as a solution template and structural overview of constituents required for integrating analysis tools into modelling environments. It therefore addresses aforementioned obstacles for tool integration and tool orchestration by providing a template for model transformation, result lifting, explicit interfaces, and protocols of analysis tools as well as hiding technical complexity of the different tools involved. We do not go into the details of soundness of the transformations and liftings in the reminder of this chapter. The interested reader is referred to Chap. 4, on composition of languages, models, and analyses, as well as to Chap. 7, on exploiting tool results, of this book [Hei+21] for further details.

5.5 Requirements for Analysis Tool Integration and Orchestration

The reference architecture described in the previous section presumes that several requirements are satisfied by the considered analysis tools. In this section, we describe such a set of requirements, such that analysis tools satisfying these requirements can be easily integrated and orchestrated into modelling environments. We base these requirements on our experience as authors—both in academia and in industry—with building and using modelling environments and integrating analysis tools. We categorise these requirements along two dimensions: requirements for the integration of analysis tools (Sect. 5.5.1), and requirements for the orchestration of analysis tools (Sect. 5.5.2). The first set of requirements is about the integration of individual tools, the second set is focused on orchestrating complex use cases with one or more analysis tools.

[14] Note that while Fig. 5.2 does show the analysis results lifted at the domain level by the $\bot$ transformation, the figure does not show what language such domain-level results conform to, for the sake of brevity. Yet, an important task here is to provide domain-level concepts that are able to represent analysis results in a domain-specific fashion, if possible through explicit relationships with concepts of the DSMLs.

5.5.1 Integration Requirements

The following are basic requirements to analysis tools which are essential to enable their integration into modelling environments.

R1.1 Explicit Input Language In order to enable the integration of an analysis tool into a modelling environment, it shall have a precisely defined input language (both when the analysis tool consumes textual files as well as when it exposes its functionality via APIs).

R1.2 Explicit Output Language While most analysis tools have a well-defined input language (i.e., to specify models to analyse), much fewer have an explicitly defined language (syntax and semantics) in which analysis results are presented. We need the syntactic definition of the result representation (aka. "output format") in order to enable parsing—e.g., having XML or JSON format for the output of the analysis tool dramatically helps in parsing the results. Besides the syntax, the semantics of the output needs to be precise enough in order to enable interpretation of analysis results.

R1.3 Explicit Protocol to Interact with the Analysis Tool We have identified three scenarios when an explicit protocol is needed for the integration. (1) Many analysis tools can be used in an interactive fashion, and in these cases the protocol of commands accepted by the tools shall be explicitly defined—e.g., the NuSMV model checker [Cim+02] offers an interactive mode that is superior to the automated interaction mode by providing a finer granular interaction protocol (i.e., sequence of NuSMV commands) that can be used to guide the analysis. (2) In the case when the analysis tool provides an API, the order in which the API functions should be called is essential for the integration—e.g., the Z3 solver [MB08] comes with a Java API that specifies the order of function calls and this eases the integration into modelling environments. (3) Many analysis tools can be called several times for performing a certain analysis, each call using some information provided in a previous call—e.g., *C bounded model checker* (CBMC) [CKL04], a bounded model checker for C and C++, can be called first to collect the properties to analyse and subsequently to analyse certain properties of interest.

R1.4 Robustness in Handling Long-Running Analyses Many times the analyses to be performed are complex and, such as the case of formal verification, they might take hours, days, or more to complete. From a tool-integrator perspective, an analysis tool needs to provide mechanisms to handle such situations either by, e.g., setting timeouts, giving feedback about the analysis progress, or enabling a "nice" cancel of analyses—e.g., the Z3 solver enables its users to specify timeouts for managing long-running analyses.

R1.5 Witness for Certification and Assurance When an analysis tool is used for checking properties of safety-critical systems, it shall provide an independently checkable witness of all analysis results. In this manner, the confidence in the correct functioning of analysis tools can be drastically increased—e.g., the software

verification competition requires competing tools to provide both correctness and violation witnesses.[15]

5.5.2 *Orchestration Requirements*

In the following, we present a set of requirements for analysis tools that aim to facilitate their orchestration in modelling environments. Orchestration is often about result exchange and coordination which brings us to these requirements.

R2.1 Reuse Partial Results Between Analyses Many uses of analysis tools via an IDE imply the integration of the analysis tool into a modelling workflow. Modelling happens today in an incremental and agile fashion, with continuous changes. The interested reader is referred to Chap. 6 of this book [Hei+21]. Ideally, the efforts required to re-analyse a model once changes are performed shall be proportional with the size of the change and not the size of the input model—e.g., caching partial verification results which make subsequent analyses faster or more tractable; another example is the ordering of *binary decision diagrams* (BDDs) variables can be saved by NuSMV [Cim+02].

R2.2 Provide Partial Results In case when the analysis is incomplete, partial results about what has been successfully covered are essential for the users—e.g., there are tools like CPAChecker [Bey16] that support their users by providing partial verification results for the cases when, e.g., the verification is untractable.

R2.3 Coordination of Portfolio Analysers In case of portfolio solvers, several tools can be started simultaneously to analyse the same property. Coordinating these tools needs to be done today at low level. Furthermore, depending on the kind of the input models and the checked property, one or another of the integrated solvers in a portfolio might be more efficient. Having explicit information about the strengths of the different solvers (with respect to the model under analysis and the checked property) could be used to increase the analysis efficiency at high-level. A notable example of portfolio solvers is, e.g., Why3 [Bob+12], that offers a unified interface for integrating and coordinating portfolios of SMT solvers.

5.6 Orchestration Strategies

A modelling environment may require the use of a single analysis tool or the coordinated use of a number of analysis tools. In the reference architecture presented earlier in this chapter, such coordination is achieved using so-called orchestration

[15] SV-COMP 2020: https://sv-comp.sosy-lab.org/2020/rules.php.

strategies, each one responsible for controlling the different tool drivers and exchanging data between tool drivers and the workspace of the modelling environment. In this section, we describe and define a selection of orchestration strategies that, from our own experience, are both, common and relevant, for composing analysis tools.

5.6.1 Orchestration Strategy Overview

An overview of the different types of strategies marked by capital letters is depicted in Fig. 5.3 and introduced in an informal way hereafter before we give precise definitions. Note that, when multiple analysis tools can be involved, we merely use two tools in the figure and the text for the sake of brevity. Furthermore, in practical settings combinations of these orchestration strategies are possible and orchestration strategies may be nested.

Single Analysis Orchestration (A) For this orchestration strategy, in order to perform a model-based analysis, the modelling environment uses a tool driver to translate a domain-specific model into a valid input for an external black-box analysis tool. Then, the modelling environment translates back (lifts) the result of the analysis tool by using the tool driver again. An example of this strategy is to translate a model conforming to a DSML into a model checker input, and then translate back the response into DSML concepts.

Separate Parallel Analysis Orchestration (B) This strategy is similar to (A), but with multiple different tools. These tools are getting the same input model, but they can run completely separately. This way we obtain separate results (both expressed in terms of the same DSML) which we can compare and from which we can select the most appropriate one. Portfolio solvers are an example of strategy B where we

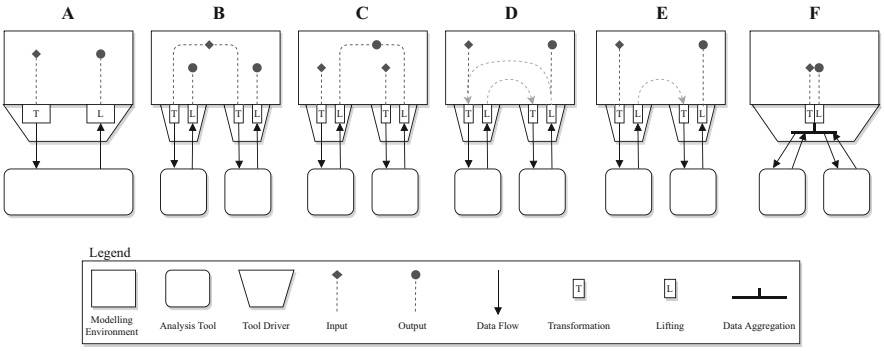

Fig. 5.3 Different types of orchestration strategies

can run the same analysis in different tools, and we make them compete, and we keep the most interesting result (e.g., fastest, more precise, etc.).

Combined Analysis Orchestration (C) In this strategy, the results of different tools that may use different input models are combined into a single output model. This strategy can be used to obtain a more comprehensive result (by combining results) than a single analysis could provide. Combination of qualitative and quantitative results of analysis tools is an example for which the combined analysis strategy can be applied.

Cooperating Analysis Orchestration (D) With this strategy the modelling environment invokes one tool, then translates the result into an input of another tool, and then translates the result of the second tool back into an input of the first tool to run another analysis. This cooperation between analysis tools can be repeated until a certain stop criteria is reached. Coupled simulations are an example of strategy D where the combination of interacting black-box simulations can create a more precise result.

Sequential Analysis Orchestration (E) With this strategy the modelling environment invokes one tool, then translates the result into an input of another tool, and then translates the result of the second tool back to a domain-specific model to provide it to the domain expert. Refining the result of one analysis tool by another is an example of the sequential analysis orchestration strategy.

5.6.2 Orchestration Strategy Definition

After giving an overview and introduction to different orchestration strategies, we now give a precise definition of each strategy.

Elements We identify the following primitives in order to define the orchestration strategies from Fig. 5.3.

- *Analysis Tool*, A_T. It has explicit (i.e., precise format and semantics) language to express the tool input I_{A_T} and explicit language to express the tool output O_{A_T} (by basic requirements); we can define $A_T = (I_{A_T}, activity(A_T), O_{A_T})$;
- *Input*, I. It is the input given to an analysis tool and is expressed in a DSML of the modelling environment;
- *Output*, O. It is the output result of an analysis tool and is expressed in a DSML of the modelling environment;
- *Transformation*, $T(I, I_{A_T})$. It is a mapping from the input I given in a DSML of the modelling environment to the input I_{A_T} of the analysis tool A_T;
- *Lifting*, $L(O_{A_T}, O)$. It is a mapping from the output O_{A_T} of the analysis of the tool A_T to the output O expressed in a DSML of the modelling environment;
- *Tool Driver*, $TD(A_T)$. It is a software component that defines how to make use of a specific analysis tool, including how to translate a domain-specific model

into a valid input for the tool, how to lift back the analysis result into a form that makes sense at the abstraction level of a domain-specific model, as well as the protocol to exchange messages and information with the tool; therefore, it is defined as a tuple

$$TD(A_T) =< T(I, I_{A_T}), L(O_{A_T}, O), protocol(A_T) >$$

where $T(I, I_{A_T})$ is the transformation to provide adequate input to the analysis tool A_T, $L(O_{A_T}, O)$ is the lifting to make the analysis result useful at the abstraction level of a domain-specific model, and $protocol(A_T)$ is the protocol to exchange messages and information with the tool A_T. Transformation T and lifting L may be the identity mapping in case the tool A_T uses/returns a model in the same language that the modelling environment uses.

A tool driver can make use of a number of different transformations and liftings, not necessarily coupled, since the supported tool A_T may require more than one input model and may return more than one result model. Therefore, we can extend the definitions of transformation and lifting in the aforementioned definition of tool driver as follows while $m, n \in \mathbb{N}_{>0}$:

$$T(I, I_{A_T}) =< T_1(I, I_{1A_T}), \ldots, T_n(I, I_{nA_T}) >$$

$$L(O_{A_T}, O) =< L_1(O_{1A_T}, O), \ldots, L_m(O_{mA_T}, O) > .$$

Further, the input to a given analysis tool may comprise more than one domain-specific model. Also the output of a given analysis tool may comprise more than one domain-specific model. In this chapter, we consider input and output each to be a single domain-specific model to not overly complicate the explanations and definitions given.

Orchestration Strategies We define the orchestration strategies depicted in Fig. 5.3 as follows while $i \in \mathbb{N}_{>0}$, $j, k, m, n \in \mathbb{N}_{>1}$ and $m \leq n$.

- *Single analysis orchestration (A):* a single tool driver $TD(A_T)$ is used, which provides input I to a tool A_T by applying transformation $T(I, I_{A_T})$ and gets the output O by applying the lifting $L(O_{A_T}, O)$.
- *Separate parallel analysis orchestration (B):* a number n of (not necessary different) tool drivers $TD(A_{T_1}), TD(A_{T_2}), \ldots, TD(A_{T_n})$ are used in parallel; each $TD(A_{T_i})$ provides input to the analysis tool A_{T_i} by applying the transformation $T(I_i, I_{A_{T_i}})$ and lifts back the output O_i to the modelling environment by applying the lifting $L(O_{A_{T_i}}, O_i)$. The input I_i can be the same input I for all $TD(A_{T_i})$, or suitably customised by applying $protocol(A_{T_i})$ of $TD(A_{T_i})$. It is up to the domain expert to decide about the use of the outputs $O_1, O_2, \ldots, O_m$ (m can be equal to n): to keep all of them, to select one of them, etc.
- *Combined analysis orchestration (C):* a number n of (not necessary different) tool drivers $TD(A_{T_1}), TD(A_{T_2}), \ldots, TD(A_{T_n})$ are used in parallel; each

$TD(A_{T_i})$ provides input to the analysis tool A_{T_i} by applying the transformation $T(I_i, I_{A_{T_i}})$ and gets the output O_i by applying the lifting $L(O_{A_{T_i}}, O_i)$. The modelling environment assembles the output O as the result of the (internal) operation `combine`$(O_1, O_2, \ldots, O_m)$ for a number m of outputs (m can be equal to n).

- *Cooperating analysis orchestration (D):* a number n of (not necessary different) tool drivers $TD(A_{T_1}), TD(A_{T_2}), \ldots, TD(A_{T_n})$ are used in a cooperating way. The modelling environment provides input I to a first analysis tool A_{T_1} by a tool driver $TD(A_{T_1})$ which applies the transformation $T(I, I_{A_{T_1}})$. According to a cooperation schema of the modelling environment, at each cooperation step, the output of a tool A_{T_i} is then given as input to another tool A_{T_j} (even if already used in previous steps) by applying the transformation $T(L(O_{A_{T_i}}, O_i), I_{A_{T_j}})$ which involves also the cooperation between $protocol(TD(A_{T_i}))$ and $protocol(TD(A_{T_j}))$; upon a stop criteria (e.g., a fixed point)—defined in the cooperation schema—is reached, the modelling environment gets the output O by applying the lifting $L(O_{A_{T_k}}, O)$ of the tool driver $TD(A_{T_k})$ for a given k.
- *Sequential analysis orchestration (E):* a number n of (not necessary different) tool drivers $TD(A_{T_1}), TD(A_{T_2}), \ldots, TD(A_{T_n})$ are used in sequence. By $TD(A_{T_1})$ the modelling environment provides input I to a first analysis tool A_{T_1} (the transformation $T(I, I_{A_{T_1}})$ is applied); at each sequential step, the output of the tool A_{T_i} is then given as input to the subsequent tool $A_{T_{i+1}}$ by applying the transformation $T(L(O_{A_{T_i}}, O_i), I_{A_{T_{i+1}}})$ in a cooperation between $protocol(TD(A_{T_i}))$ and $protocol(TD(A_{T_{i+1}}))$; upon end of the sequential use of the orchestrated tools, the modelling environment gets the output O by applying the lifting $L(O_{A_{T_n}}, O)$.

Combinations or nested compositions of the above orchestration strategies are possible in order to perform complex model-based analyses that require tools orchestrated in a more sophisticated way. Combined strategies might also require more powerful tool drivers that can share transformations towards specific formats and are able to combine output results. Indeed, in case different tools are used according to a complex schema such as the orchestration strategy F in Fig. 5.3, where a number n of (not necessary different) analysis tools $A_{T_1}, A_{T_2}, \ldots, A_{T_n}$ are used, the unique tool driver needs a complex transformation able to transform the input model (or suitable parts of it) into the inputs of specific tools, and combine/aggregate (in a suitable way) all or some output results before lifting the analysis result back to the modelling environment.

However, orchestration strategies whose transformation and lifting require to share information and combine results are not in the focus of this chapter, and therefore the formalisation of orchestration strategies such as the case F is an open topic that needs to be addressed in future research.

5.7 Examples of Orchestration Strategy Application

This section provides examples for each of the strategies defined in the previous section to illustrate their application in existing modelling environments.

Single Analysis Orchestration (A) Model-based simulation is an example of the application of the single analysis orchestration strategy. For example, the Palladio [Reu+16] software architecture modelling and analysis approach (cf. Chap. 11 of this book [Hei+21]) uses various analysis techniques to predict quality properties of software systems. The Palladio-Bench corresponds to the modelling environment. To conduct a quality analysis of a software system, an architectural model of the system is created by domain experts in the Palladio-Bench. Several analysis tools can be selected to be executed based on the model. One of these analysis tools is the performance simulator SimuCom [Bec08]. The Palladio-Bench transforms the domain-specific model (i.e., architectural model) into simulation code of SimuCom, which is executed for performance simulation. After the simulation has been finished, the result is lifted back to the Palladio-Bench. The Palladio-Bench in turn displays the result to the domain experts.

Similarly, the single analysis orchestration strategy is used in the ASMETA modelling environment [Arc+11] to perform model-based analysis of the ASM specifications. The ASMETA modelling environment can invoke a number of tools for model validation (e.g., interactive or random simulation by the simulator AsmetaS, animation by the animator AsmetaA, scenario construction and validation by the validator AsmetaV) and verification (e.g., static analysis by the model reviewer AsmetaMA, proof of temporal properties by the model checker AsmetaSMV, proof of correct model refinement [AGR16]). All these tools are orchestrated in a similar way: An ASM model is given as input to a given tool by means of a transformation that translates the input model into an adequate input for the target tool; the result of the analysis is then lifted back in a way that it is understandable by the domain expert.

The modelling environments mbeddr [Voe+12], FASTEN [RGS19], and AF3 [Ara+15] all also use this orchestration strategy. Domain-specific models are translated into the input language of analysis tools, let them run and subsequently lift the results at model level so that they are understandable to domain experts.

Separate Parallel Analysis Orchestration (B) Typical examples of strategy B are portfolio solvers which use multiple solver tools to run in parallel in order to tackle computationally difficult problems. A well known modelling environment for different analysis tools—e.g., SMT solvers—is Why3 [Bob+12]. Why3 takes input models described in a high-level language which aims at maximal expressiveness without sacrificing efficiency of automated proof search. Based on the input models, Why3 applies transformations that will gradually translate Why3's logic into the logic of different provers (e.g., Z3 [MB08], CVC4 [Bar+11], Yices [Dut14]). The transformations are controlled by a configuration file, called a driver, associated with any prover supported by Why3. The results of the external provers are then interpreted and (to some extent) lifted at the level of the input language.

The ASMETA modelling environment also implements the separate parallel analysis orchestration strategy for ASM model validation. The domain expert can invoke the parallel execution of the simulator AsmetaS and the animator AsmetaA on a same input model; results of these analysis tools are lifted back to the modelling environment to show possible states where inconsistent updates (i.e., the same location is simultaneously updated to different values) or invariant violations are detected.

mbeddr features analyses both at model level [Rat+12a] and at code level [MVR14]. Model-level analyses such as checking the consistency and completeness of decision tables are faster but less precise than analyses on code level since they do not take into account the C language semantics with respect to arithmetic, floating points or pointers. Tools that implement these analyses can be run in parallel to combine the advantages of analyses on both levels. Results of the analysis tools can be collected and presented at the level of the domain-specific model.

Combined analysis orchestration (C) CoMA [AGR11] is a tool for runtime verification of Java code with respect to its ASM specification. It observes the behaviour of a Java object O and checks whether it conforms to the expected behaviour captured by an ASM specification M_O. CoMA works as modelling environment having two languages: Java for specifying the structure and the behaviour of the object O, and AsmetaL to model the ASM M_O. Code annotation in O is used for establishing a suitable link between fields and methods of O and the state signature (i.e., a set of locations) of M_O. The operation of CoMA exploits the orchestration strategy C on two tool drivers: that of the *Java virtual machine* (JVM) and that of the AsmetaS simulator. Transformations T are the identity mappings in both cases, while lifting L of the JVM tool driver reports back the state (i.e., a set of memory values) of a Java object and that of the AsmetaS tool driver lifts back the state of an ASM (i.e., a set of locations' values). At a generic step of the runtime verification, CoMA invokes the simulation of O on the JVM. When a changing method (i.e., a method that the domain expert wants to observe and that has been linked to the model) of O is executed, the tool driver of JVM lifts back the current state s_O of O, and the modelling environment invokes the simulator AsmetaS on the model M_O to perform a computation step. The tool driver of AsmetaS lifts back the current model state s_{M_O}. The modelling environment then checks whether a conformance relation holds between current states s_O and s_{M_O}. If they conform, the simulation of the Java object can continue and the orchestration of the two tools starts again, otherwise a lack of conformance between code and specification is reported, so concluding the runtime monitoring. According to our formal definition of the orchestration strategy C, the function `combine` is the conformance checking predicate since its truth value is computed by combining information from the outputs of the tool drivers of the JVM and the AsmetaS simulator.

Further, the IDE VCES [GLO11] follows the combined analysis orchestration strategy. VCES can be used for both qualitative and quantitative analyses by using two analysis tools, namely NuSMV [Cim+02] and PRISM [KNP11]. Results of these analysis tools are lifted back to and can be combined in VECS.

Cooperating Analysis Orchestration (D) An example of the cooperating analysis orchestration strategy is simulation coupling. For instance, *Maritime Simulation* (MariSim) [TO17] comprises several simulation tools that are related to Navy and maritime scenarios. These simulation tools can interact, for example, in order to analyse tactical formations at sea. The MariSim modelling environment is used to model and control the interaction between the simulation tools. Simulation parameters (e.g., time of day or wind direction) are described in a domain-specific model in the MariSim modelling environment and transformed for the corresponding simulation tools by tool drivers. Simulation results are lifted back to the modelling environment and passed on to another simulation tool for interaction purposes. Simulating tactical formations at sea requires continuous interaction among the simulation tools, i.e., exchange of information by transformation and lifting, until a certain stop criteria is reached.

Another example of exploiting the cooperating analysis orchestration strategy is CoMA-SMT [AGR14], which has been developed for runtime verification of Java code with respect to an ASM model in case of nondeterministic behaviour. Coma-SMT is a modelling environment using, as languages, Java for specifying the structure and the behaviour of an object, and Yices for representing (initial state and transitions of) a nondeterministic ASM capturing the code behaviour as context of the SMT solver. Coma-SMT orchestrates the tool driver of the JVM for the simulation of Java code and that of the SMT solver Yices for satisfiability checking of a context theory. Transformations T are the identity mappings in both cases, while lifting L of the JVM tool driver reports back the state (i.e., a set of memory values) of a Java object and that of the Yices tool driver lifts back the result of a context satisfiability checking. At a generic step of the runtime verification, the CoMA-SMT modelling environment invokes the Java simulation on the JVM. When a changing method (i.e., a method that the domain expert wants to observe) is invoked, the tool driver of the JVM lifts back the current state of the Java object, and the SMT solver is triggered by the modelling environment: The transformation consists in extending the (current) Yices logical context by asserting a set of formulas stating the values of the observed elements in the current state of the Java object. Yices is then used to check satisfiability of the logical context. If the context is unsatisfiable, then the implementation does not conform with the model and the runtime verification stops (failure fixed point is reached, see definition of strategy D); otherwise the modelling environment continues the runtime verification by invoking a new computation step of the Java program (in this case the transformation from the output model of Yices to the input model of JVM is empty) until an end point of the computation (successful fixed point) is reached.

Sequential Analysis Orchestration (E) The ASMETA modelling environment also exploits this kind of strategy to orchestrate the sequential use of two different tools to implement an approach for the automatic generation of scenarios (or abstract test cases) for refined ASM models starting from abstract scenarios of abstract ASM models [AR19]. This approach is extremely useful to allow reuse of artefacts in model refinement, and is based on a classical test generation technique by exploiting counterexample generation by model checking. In this approach, the ASMETA

modelling environment first invokes a tool that transforms the abstract scenario S_A of the abstract ASM model A into a suitable temporal logic formula ψ; $\not\psi$ (usually called *trap property*) is then model checked against the refined model R of A, and a counterexample c_{ex} is returned to the modelling environment. The counterexample c_{ex} represents a simulation trace of R characterised by ψ. c_{ex} is then transformed into a scenario S_R and given as input to the validator tool AsmetaV on the refined model R. AsmetaV then reports back the result of the scenario execution to the modelling environment.

The sequential analysis orchestration strategy is also used in ASMETA for model-based testing of Java code [AGR18]. The modelling environment invokes the ASM-based test generator *ASM tests generation tool* (ATGT) to derive a test suite T from an ASM specification model of a piece of Java code. Tests in T are then instrumented as JUnit tests by suitable transformation. The results of running JUnit tests on the Java code are then lifted back to the modelling environment. A similar orchestration strategy has been used in [BGM20] to implement an approach that translates abstract test sequences, either generated randomly or through model checking, and scenarios to concrete C++ unit tests using the Boost library.[16] In this case, the orchestrated tools are ATGT or AsmetaV on one side, and the platform to run C++ test drivers on the other side.

Nested Orchestration Strategies AdaptiveFlow [Sir+19, For+20] is a modelling environment for flow management in track-based systems. In AdaptiveFlow, we have nested orchestration strategies; a smaller step using single analysis orchestration (strategy A) within a sequential analysis orchestration (strategy E), together being executed in a loop. AdaptiveFlow can be used in different application domains like for fleet management of collaborating heavy machines in a quarry, coordinating robots in factory aisles, and resource management of smart transport hubs in a city.

Figure 5.4 shows AdaptiveFlow and two analysis tools, the *Rebeca model checker* (RMC) and the *state space analyzer* (SSA). RMC [Reb19] is a customised analysis tool for the Rebeca language and its timed extension [Sir+04, SK16]. SSA [For+20] is developed specifically for AdaptiveFlow but can be reused in other modelling environments as well. The input to SSA is the exact same output from RMC.

As shown in Fig. 5.4, the modelling environment invokes one analysis tool (RMC), and then the output of RMC is fed into another analysis tool (SSA). Here no translation between the output of RMC and the input of SSA is necessary, because SSA is designed in a way to accept the output of RMC. The initial model is revised iteratively and automatically, and in each iteration the revised model is fed as an input to the first tool (RMC), and the output of RMC is fed to the second tool (SSA). This is an instance of orchestration strategy E executed in a loop. Within each instance of the strategy E in the loop, there is a smaller step using the orchestration strategy A, this is debugging of Rebeca models using RMC which may come back

[16] Boost library: https://www.boost.org/.

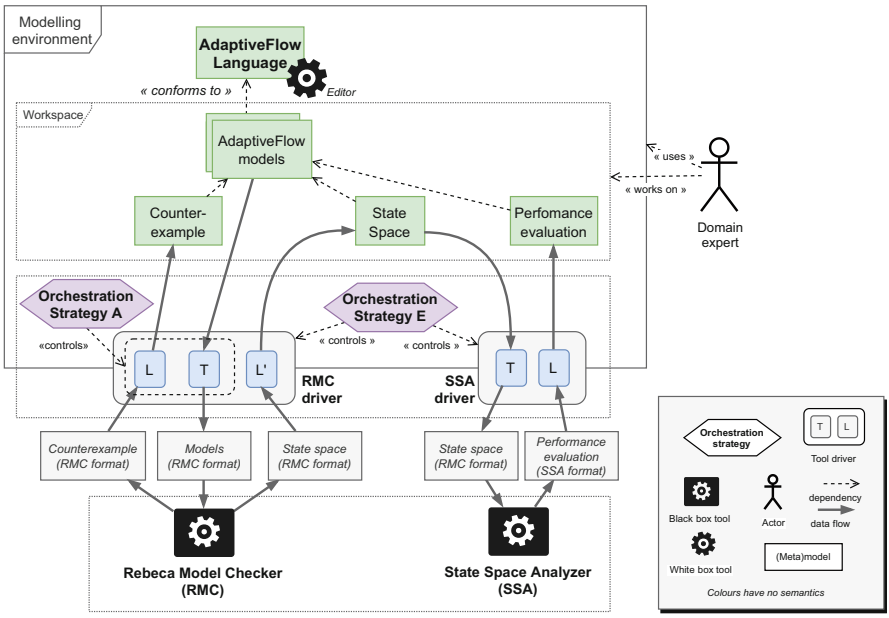

Fig. 5.4 AdaptiveFlow example

by creating a counterexample and jump out of the sequential orchestration (shown as the dashed box in Fig. 5.4).

The RMC driver is responsible for the transformation of AdaptiveFlow models to Rebeca models (RMC format). The driver receives the AdaptiveFlow model as input, generates the output as a Rebeca model, and feeds the Rebeca model to RMC. The AdaptiveFlow model consists of three files including information about the environment, points of interests, and configuration of the system and the moving objects. There is no well-defined DSML for modelling this information, but the specified format of the inputs can be considered as the language. The content and the terminology are selected based on the specific domain of flow management of track-based systems. The Rebeca model is generated by model transformation based on these inputs. RMC receives the Rebeca model and generates the state space and checks the correctness properties. The correctness properties include the safety (lack of collision) and progress (guarantee of no deadlock). The lifting here is an identity mapping, the semantic gap between the Rebeca model and the problem domain is not large, and the domain expert can understand and use the counterexample for debugging. When the correctness of the model is verified, the state space is fed into the SSA tool for performance evaluation of the system. The output of this tool shows the performance measures for different configurations that are checked in different iterations. This output is lifted to be usable for the domain expert once a certain stop criteria is reached. The output is checked by the domain expert, and helps the

domain expert in decision making for adjusting the configuration and improving the performance.

This example is explained in more detail in Chap. 13 of this book [Hei+21], together with other similar examples representing different orchestration strategies.

5.8 Conclusion and Outlook

This chapter discussed the challenge of how to integrate and orchestrate external analysis tools into modelling environments. We first gave a detailed overview of the considered context and problem to be addressed. Then, we proposed a reference architecture along with important concepts that can be used to methodically integrate and orchestrate analysis tools into modelling environments. We specified a set of requirements that qualify which analysis tools can properly be integrated and orchestrated based on the reference architecture. Finally, we proposed and formalised a first set of strategies that can be used to answer common integration and orchestration cases and showed examples of the application of these strategies in real-world modelling environments.

Further investigation on additional ways of tool integration and orchestration is needed. These include strategies whose transformations and liftings require to share information and combine results like we sketched for case F in this chapter. The formalisation of orchestration strategies as the case F needs to be addressed in the future. Another open topic is to support the specification of new orchestration strategies by providing primitives, languages, and processes for defining orchestration strategies that may build upon the concepts proposed in this chapter. Furthermore, the soundness of the transformations and liftings proposed in this chapter may be examined in the future. Work on language engineering is required to precisely define the transformation of analysis inputs and the lifting of analysis results.

References

[AGR11] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. “CoMA: Conformance Monitoring of Java Programs by Abstract State Machines”. In: *Runtime Verification—Second International Conference*. 2011, pp. 223–238. https://doi.org/10.1007/978-3-642-29860-8_17.

[AGR14] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. “Using SMT for dealing with nondeterminism in ASM-based runtime verification”. In: *Electronic Communications of the EASST* 70 (2014), pp. 1–15. https://doi.org/10.14279/tuj.eceasst.70.970.

[AGR16] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. “SMT-Based Automatic Proof of ASM Model Refinement”. In: *Software Engineering and Formal Methods - 14th International Conference*. 2016, pp. 253–269. https://doi.org/10.1007/978-3-319-41591-8_17.

[AGR18] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. “Closing the gap between the specification and the implementation: the ASMETA way”. In: *Models: Concepts, Theory, Logic, Reasoning and Semantics - Essays Dedicated to Klaus-Dieter Schewe on the Occasion of his 60th Birthday*. 2018, pp. 242–263.

[AR19] Paolo Arcaini and Elvinia Riccobene. “Automatic Refinement of ASM Abstract Test Cases”. In: *IEEE International Conference on Software Testing, Verification and Validation Workshops*. 2019, pp. 1–10. https://doi.org/10.1109/ICSTW.2019.00025.

[Ara+15] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. “AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems”. In: *8th International Workshop on Model-based Architecting of Cyber-Physical and Embedded Systems*. 2015, pp. 19–26. http://ceurws.org/Vol-1508/paper4.pdf.

[Arc+11] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. “A model-driven process for engineering a toolset for a formal method”. In: *Software: Practice and Experience* 41.2 (2011), pp. 155–166. https://doi.org/10.1002/spe.1019.

[Bar+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. “CVC4”. In: *Proceedings of the 23rd International Conference on Computer Aided Verification, CAV*. 2011, pp. 171–177.

[Bec08] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. Universitätsverlag Karlsruhe, 2008. https://publikationen.bibliothek.kit.edu/1000009095.

[Ber+08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. “Fiacre: an Intermediate Language for Model Verification in the Topcased Environment”. In: *4th European Congress ERTS Embedded Real Time Software*. Jan. 2008, 8p. https://hal.inria.fr/inria-00262442.

[Bey16] Dirk Beyer. “Partial Verification and Intermediate Results as a Solution to Combine Automatic and Interactive Verification Techniques”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. 2016, pp. 874–880.

[BGM20] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. “Design and validation of a C++ code generator from Abstract State Machines specifications”. In: *Journal of Software: Evolution and Process* 32.2 (2020). https://doi.org/10.1002/smr.2205.

[BGS05] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. “Model Bus: Towards the Interoperability of Modelling Tools”. In: *European Workshop on Model Driven Architecture*. 2005, pp. 17–32. https://doi.org/10.1007/11538097_2.

[BK96] Johannes Aldert Bergstra and Paul Klint. “The ToolBus coordination architecture”. In: *Coordination Languages and Models*. 1996, pp. 75–88. https://doi.org/10.1007/3-540-61052-9_40.

[BMW97] Volker Braun, Tiziana Margaria, and Carsten Weise. “Integrating tools in the ETI platform”. In: *International Journal on Software Tools for Technology Transfer* 1.1-2 (Dec. 1997), pp. 31–48. https://doi.org/10.1007/s100090050004.

[Bob+12] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages* (May 2012).

[Cav+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. “The nuXmv Symbolic Model Checker”. In: *Computer Aided Verification*. 2014, pp. 334–342.

[CDT13] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. “OCRA: A Tool for Checking the Refinement of Temporal Contracts”. In: International Conference on *Automated Software Engineering*. 2013, pp. 702–705.

[Cim+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. “NuSMV 2: An

OpenSource Tool for Symbolic Model Checking". In: *14th International Conference on Computer Aided Verification, CAV, Proceedings*. 2002, pp. 359–364.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2988. 2004, pp. 168–176.

[Cru+13] Simon Cruanes, Gregoire Hamon, Sam Owre, and Natarajan Shankar. "Tool Integration with the Evidential Tool Bus". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. 2013, pp. 275–294. https://doi.org/10.1007/978-3-642-35873-9_18.

[DE10] Matthew B. Dwyer and Sebastian Elbaum. "Unifying verification and validation techniques". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 2010. https://doi.org/10.1145/1882362.1882382.

[Dut14] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification*. 2014, pp. 737–744.

[Far+06] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe LE Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. "The TOPCASED project: a Toolkit in Open source for Critical Aeronautic SystEms Design". In: *European Congress on Embedded Real Time Software*. 2006. https://hal.archivesouvertes.fr/hal-02270461.

[FGH06] Peter Feiler, David Gluch, and John Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Tech. rep. CMU/SEI-2006-TN-011. Software Engineering Institute, Carnegie Mellon University, 2006. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879.

[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: *J. Inf. Process.* 28 (2020), pp. 588–598.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3— Where Programs Meet Provers". In: *European Symposium on Programming Languages and Systems*. 2013, pp. 125–128. https://doi.org/10.1007/978-3-642-37036-6_8.

[GLO11] Matthias Güdemann, Michael Lipaczewski, and Frank Ortmeier. "Tool Supported Model-Based Safety Analysis and Optimization". In: *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing*. Jan. 1, 2011. http://ieeexplore.ieee.org/abstract/document/6133100/.

[GRS08] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. "A Metamodel-based Language and a Simulation Engine for Abstract State Machines". In: *Journal of Universal Computer Science* 14.12 (2008), pp. 1949–1983. https://doi.org/10.3217/jucs-014-12-1949.

[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.

[Hol03] Gerard Holzmann. The Spin Model Checker: *Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification*. Vol. 6806. 2011, pp. 585–591.

[Mar05] Tiziana Margaria. "Web services-based tool-integration in the ETI platform". In: *Software & Systems Modeling* 4.2 (May 2005), pp. 141–156. https://doi.org/10.1007/s10270-004-0072-z.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340.

[MNS05] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. "jETI: A Tool for Remote Tool Integration". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2005, pp. 557–562. https://doi.org/10.1007/978-3-540-31980-1_38.

[MVR14] Zaur Molotnikov, Markus Völter, and Daniel Ratiu. "Automated domain-specific C verification with mbeddr". In: *International Conference on Automated Software Engineering*. 2014, pp. 539–550. https://doi.org/10.1145/2642937.2642938.

[Obj12] Object Management Group. OMG Systems Modeling Language (OMG SysML), *Version 1.3*. 2012. http://www.omg.org/spec/SysML/1.3/.

[Obj15] Object Management Group. *UML 2.5*. Tech. rep. formal/2015-03-01. Object Management Group, 2015.

[Rat+12a] Daniel Ratiu, Bernhard Schaetz, Markus Voelter, and Bernd Kolb. "Language engineering as an enabler for incrementally defined formal analyses". In: *1st International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*. 2012, pp. 9–15. https://doi.org/10.1109/FormSERA.2012.6229790.

[Rat+12b] Daniel Ratiu, Markus Voelter, Zaur Molotnikov, and Bernhard Schaetz. "Implementing Modular Domain Specific Languages and Analyses". In: *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*. 2012, pp. 35–40. https://doi.org/10.1145/2427376.2427383.

[Rat+13] Daniel Ratiu, Markus Voelter, Bernd Kolb, and Bernhard Schaetz. "Using Language Engineering to Lift Languages and Analyses at the Domain Level". In: *NASA Formal Methods Symposium*. 2013, pp. 465–471. https://doi.org/10.1007/978-3-642-38088-4_35.

[Reb19] Rebeca. *Afra Tool*. http://rebeca-lang.org/alltools/Afra.2019.

[Reu+16] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. https://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[RGS19] Daniel Ratiu, Marco Gario, and Hannes Schoenhaar. "FASTEN: An Open Extensible Framework to Experiment with Formal Specification Approaches: Using Language Engineering to Develop a Multi-Paradigm Specification Environment for NuSMV". In: *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering*. 2019, pp. 41–50. https://doi.org/10.1109/FormaliSE.2019.00013.

[RU19] Daniel Ratiu and Andreas Ulrich. "An integrated environment for Spin-based C code checking - Towards bringing model-driven code checking closer to practitioners". In: *International Journal of Software Tools for Technology Transfer* 21.3 (2019), pp. 267–286. https://doi.org/10.1007/s10009-019-00510-w.

[Rus05] John Rushby. "An Evidential Tool Bus". In: *Formal Methods and Software Engineering*. 2005, pp. 36–36. https://doi.org/10.1007/11576280_3.

[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.

[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.

[SMB97] Bernhard Steffen, Tiziana Margaria, and Volker Braun. "The Electronic Tool Integration platform: concepts and design". In: *International Journal on Software Tools for Technology Transfer* 1.1-2 (Dec. 1997), pp. 9–30. https://doi.org/10.1007/s100090050003.

[Som15] Ian Sommerville. *Software Engineering*. Pearson, 2015.

[TO17] Okan Topçu and Halit Oğuztüzün. *Guide to Distributed Simulation with HLA*. Springer, 2017. https://doi.org/10.1007/978-3-319-61267-6.

[Voe+12] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. "Mbeddr: An Extensible C-Based Programming Language and IDE for Embedded Systems". In: *3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. 2012, pp. 121–140. https://doi.org/10.1145/2384716.2384767.