Model Execution and Debugging: A process to leverage existing tools

Faiez Zalila¹, Eric Jenn¹* and Marc Pantel²
¹IRT Antoine de Saint Exupéry, Toulouse, France
²IRIT, Université de Toulouse, Toulouse, France
{faiez.zalila, eric.jenn}@irt-saintexupery.com, marc.pantel@enseeiht.fr

Keywords: Modeling, Formal verification, Model-checking, Debugging, Simulation, Model Execution, IDE

Abstract: Model checking is an effective technique for the verification of critical systems. However, it relies on behavioral models which writing and verification is most of time costly. Thus, those models shall be validated and debugged thoroughly, and simulation, i.e. model execution, can be used for that purpose. To reduce the development costs of simulators and ensure their behavioral consistency with model verifiers, we advocate the reuse of parts of the model verification toolchain to implement them. To support this claim, this paper proposes a method illustrated with a realistic case study applied to FIACRE behavioral models. The approach relies on the creation and exploitation of relations between models representing the information required by the user on the one hand, and information produced by the tools, on the other hand.

1 INTRODUCTION

1.1 Problem statement

Early validation and verification (V&V) activities reduce development costs, as specification and design errors can be detected and fixed as soon as possible in the development process. To that purpose, these activities are performed on various system models (requirements, architecture, design, function, etc.) expressed using Domain Specific Modeling Languages (DSMLs).

Whenever complex behavioral properties are at stake, model-checking is an efficient approach to prove the absence of errors on those models. However, to overcome scalability issues, it is usually mandatory to create multiple models, at various abstraction levels, covering several kind of properties, etc.

Animating those models is one of the best means to remove trivial modeling bugs, to ensure that the model indeed expresses the designers intents, and eventually to reduce the overall cost of verification (Bourdil et al., 2016b).

To be verified or validated, models expressed using abstract, user-level, languages are usually transformed into the more concrete formalisms of model-checkers and/or simulators (Visser et al., 2012). Then, to be exploited, the results produced by these tools must be re-interpreted in terms of the abstract language. Obviously, such roundtrip between abstract and concrete models could be avoided by developing V&V means directly applicable on abstract languages.

However, we advocate the roundtrip strategy for two main reasons: (i) developing a new model checker or simulator and ensuring the semantics consistency of both tools is a very complex task, and (ii) there already exists a plethora of model-checkers and/or simulators. Unfortunately, the necessary re-interpretation phase is not trivial as information get lost during the successive transformations to verification languages: it relies on the appropriate use, combination, and possibly completion of available data. In this paper, we propose an approach to implement the roundtrip strategy based on the analysis of annotated metamodels.

1.2 Our contribution

Our approach combines and leverages existing low-level verification, validation, and transformation means to provide the end-user with appropriate debugging information. It relies on the construction of transitive relations between the data produced by the various tools, and the user requirements for the model.
We illustrate this approach on the development of a model simulator for the Fiacre language (Berthomieu et al., 2008) using the existing model checking toolbox TINA (Berthomieu et al., 2004).

This paper is structured as follows. Section 2 presents the context of the study and the use-case. It gives an overview of the user requirements for the model simulator. Section 3 proposes different design methods for such tools. It details their implementation and discuss about the adopted solution. Section 4 gives some related works in the domain of the simulators design. Section 5 concludes the paper.

2 THE CONTEXT AND THE CASE-STUDY

The work presented in this paper is carried out in the framework of the INGEQUIP project at the Institut de Recherche Technologique Saint-Exupry in Toulouse, France. The project experiments and assesses various engineering methods and tools in the domain of hardware/software co-design, virtual integration and formal verification in the automotive, space, and aeronautics domains.

A small three-wheeled rover is used as demonstrator. Its architecture is representative of a significant family of real systems. It is composed of a mission subsystem in charge of the computation of the rover mission, trajectory tracking, etc. and a power subsystem in charge of the management of the powertrain.
The two subsystems are interconnected by a unique CAN bus.

To comply with the availability and safety requirements, the mission subsystem is broken down into two channels (left/right) with two units per channel (COM/MON). A clock synchronization (CS) protocol (Rodrigues et al., 1998) ensures a synchronous behavior of all units.

This CS protocol model is around 700 lines of Fiacre code covering both the units to be synchronized and the communication network (CAN). Verification is performed using the TINA toolchain. Even though directly inspired from (Rodrigues et al., 1998), building the model of the CS protocol required a significant design and debugging effort due to the various abstractions and simplifications that were required to obtain a tractable model (Bourdil et al., 2016a). In the rest of the document, we will take small samples of this model as illustrate specific issues encountered during the design of the model simulator.

2.1 The Fiacre modeling language

Fiacre is the French acronym for Intermediate Format for Embedded Distributed Components Architectures. It was designed as the target language for model transformations from different DSMLs such as (Rangra and Gaudin, 2014), AADL (Berthomieu et al., 2010; Bodeveix et al., 2015) or LADDER (Farines et al., 2011). Fiacre is used to describe the behavioral and timing aspects of concurrent systems for formal verification and simulation purposes. It is built around two main constructs:

- **Processes** modelling sequential behaviors from states and transitions. A transition contains deterministic statements (assignments, conditionals, loops, and sequential compositions), non-deterministic statements (non-deterministic choice and non-deterministic assignments), communication statements, and state transitions.

- **Component** modelling concurrent and hierarchical composition of communicating processes.

2.2 The TINA verification toolbox

Verifications are performed by the TINA toolbox, a set of tools used to edit and analyze Timed Transition System (TTS), an extension of Time Petri Nets (TPN) with data manipulation. The following toolbox components are considered in this paper:

- **TINA** constructs reachability graphs and Kripke transitions systems from TTS and TPN.

- **PLAY** is a TTS and TPN animator.

To be processed by TINA, a Fiacre model must be translated to TTS using the dedicated tool FRAC. Due to the semantic gap between the two languages, some constructs present in the Fiacre input may be hidden from the TTS output. Fortunately, FRAC can also generate transformation traceability data (using

2The complete Fiacre model is accessible at http://projects.laas.fr/fiacre/examples/2016-twirtee/twirtee/claims/c1.fcr

3http://projects.laas.fr/fiacre/download.php
the -G option) that can be exploited to build the Fiacre simulator feedback. Now, let us consider the designers needs in terms of debugging features.

2.3 Requirements for the Fiacre simulator

As stated before, even though verification is highly automated thanks to model checking techniques, building the model remains a manual activity. The model developer requires means to assess that the model indeed expresses its intention and eliminate modeling errors before starting the formal verification phase (which might be quite costly).

Moreover, after the model checking phase, s/he also needs means to interpret the counter-examples that may be produced by the model-checker. To some extent, debugging models is similar to debugging programs: the user needs capabilities to observe the sequences of states during the model execution, step through these sequences, stop the execution when some condition occurs, etc.

More precisely, it relates to debugging a multithreaded software since the execution of a Fiacre model is the composition of multiple processes executed concurrently. However, some differences are worth mentioning: (i) the user has a full control of time, (ii) some transitions within processes may be selected non-deterministically (select clauses), (iii) some state transitions may occur synchronously between processes, etc.

From now on, we will focus on a few key requirements for such an execution/debugging environment and see how we managed to implement them with a minimal development effort.

Let FS be the Fiacre Simulator under design.

- **REQ-1**: The FS shall refer to modeling elements using user-level designation. For instance, it shall present values according to the representation used in the Fiacre source model. This applies in particular to data types like structs, unions, etc.

- **REQ-2**: When applicable, the FS shall display the location of the modeling elements in the source model. Reciprocally, the FS shall provide the user with the capability to select or designate an element directly on the source model.

- **REQ-3**: The FS shall visualize the evolution of Fiacre variables and states along time.

- **REQ-4**: The FS shall allow breakpoint to be placed on any transition in the source model. Breakpoints shall be triggered when the transition is fired. (Breakpoints are not placed on statements.)

Listing 2: Excerpt of a TTS execution trace produced by the TTS simulator (PLAY)

```
| date: 0 | state 5: Can_1_srcv, Can_1_vstates=0, Can_1_vm={mtype=Adjust, nid=-1, omissions=0, round=0, sid=-1}, Can_1_vi=0, Can_1_vfo=0, Can_1_vomissions=[0,0,0], Can_1_vomission=false enabled: Can_1_t0 [0,0] StartRound_1_t4 [0,w] StartRound_1_t5 [44999955,45000045] StartRound_2_t0 [0,0] StartRound_2_t1 [0,0] StartRound_3_t0 [0,0] StartRound_3_t1 [0,0] firable: Can_1_t0 StartRound_1_t4 StartRound_2_t0 StartRound_2_t1 StartRound_3_t0 StartRound_3_t1 ? # 0 do firing: Can_1_t0
```

The previous list of requirements is (partially) described on Figure 2: a debugging session is a sequence of debugging steps, each step being a triple (observation, analysis, action) corresponding to the usual scenario where: (i) the system is executed, (ii) some observations are obtained from this execution,
and (iii) those observations determine the next step of execution.

Of course, part of the triple may be ignored in an execution step: observations may be ignored during some specific phases (e.g., case of initialization), actions may be automated (e.g., random selection of transitions), etc.

In the rest of the document, focus is placed on the Observation part (in blue on Figure 2). It is expanded in Figure 3 where it is linked to the data provided by the other available models.

3 DESIGN SOLUTIONS

Figure 3 shows the Fiacre and TTS technical domains involved in the implementation. The approach consists in analyzing TTS execution information (computed using the TTS Simulation model and stored in the TTS Trace model) to obtain the corresponding execution information at the Fiacre level. Figure 4 shows the three solutions that are presented and analyzed hereafter.

3.1 Solution 1: Use the TTS simulation model

The first solution (the blue part at the top of Figure 4) exploits two sources of information: the TTS description that represents the structure of a TTS model, and the TTS simulation model. Listing 2 shows a sample of the TTS simulation model corresponding to the CAN Fiacre process introduced earlier.

From these two sources, information about the execution of the Fiacre model is obtained by a sequence of three phases: extraction, identification, and construction.

Extraction consists in analyzing the textual output of the PLAY tool to produce the TTS simulation model and, thus, instantiate TTS simulation metaclasses (DynamicTTSPlace, DynamicTTSVariable, etc.). This phase is implemented using Xtext4.

Identification associates the TTS description elements with the Fiacre model elements. To do that, some knowledge is required about (i) how the TTS description is built from the Fiacre model, and (ii) how the Fiacre model elements are encoded in the TTS description. For example, state “txtime” of the first instance of the CAN process declaration in Fiacre is encoded as TTS place “Can1_stxtime” (the “s” in the id means that corresponds to a Fiacre state declaration). Using this naming convention, one is able to retrieve the source elements in the Fiacre model.

Finally, construction produces the simulation information at Fiacre level by instantiating the appropriate elements of the Fiacre simulation meta-model using the Identified Simulation Model. Unfortunately, this first solution only complies with user requirement REQ-3 due to missing data in the TTS description and

4https://www.eclipse.org/Xtext/
3.2 Solution 2: Use traceability data

Solution 2 (in red in Figure 4) extends solution 1 by combining information given by the initial trace model (see Listing 2) with traceability information generated by FRAC (-G option).

Historically, this information was used for debugging purposes during the development of FRAC. Later, it was also used to feed verification results from the TTS level back to Fiacre (Zalila et al., 2012; Zalila et al., 2013). It is produced during the last step of the translation phase, before the generation of the TTS description. It contains TPN and data processing constructs (guards, assignments, etc.).

Listing 3 shows a subset of a compilation trace model related to the CAN process shown in Listing 1. It contains two TTS transitions: Can_1_t0 and Can_1_t1. In order to locate these transitions in the Fiacre source model using the initial compilation trace model, it is necessary to understand how the FRAC compiler generates TTS identifiers from Fiacre-level identifiers.

Once this relation is established, the transitions are immediately located in the source code. For example, transition Can_1_t0, which identifies ends with t0, corresponds to the first transition on the first instance of a CAN process (see lines 9-13 in Listing 1). Similarly, transition “Can_1_t1” corresponds to the transition located at lines 14-16 in Listing 1.

To analyze the hybrid TTS of Listing 3, both syntactic and semantic analysis are required. Syntactic analysis raises no particular difficulty. Semantic analysis can be achieved in two ways. First, the transition body may be analyzed line-by-line. This solution requires a significant effort and in-depth knowledge on the internals of FRAC. For example, FRAC sometimes adds internal guards (e.g., guard on true for transition Can_1_t1), enriches existing guards (e.g., guard of transition Can_1_t0), adds internal assignments, replaces constant identifiers by their value, etc.

Second, the index given in the transitions identifiers (t0, t1, t2, etc.) may be used as the rank of the transition in the source code of the process.

However, the presence of nested non-deterministic constructs containing quite similar source code makes this task extremely difficult. Unfortunately, this solution satisfies user requirement REQ-2 only partially as it fails to reach the source code level.
3.3 Solution 3: Use the ECT model

Solutions 1 and 2 have not succeeded to satisfy all user requirements. Used as black-boxes, the existing tools do not provide sufficient information to associate univocally the Fiacre model elements with the TTS model elements: information is so degraded that it cannot be reconstructed. The last proposal is then to extend the FRAC tool in order to export the information lost in translation. Accordingly, we introduce a new trace model, called Extended Compilation Trace (ECT), that is generated when the option j of FRAC is activated.

The ECT model offers a direct mapping between the Fiacre source code elements and the corresponding constructs in the TTS model. The structure of the ECT reflects the hierarchical organization of the Fiacre model. Listing 4 shows a subset of the generated ECT model related to the CAN process. In this example, lines 8-12 associate the generated TTS place identifier Can_1_srcv with its corresponding Fiacre source code (sourcename) and its location (character 10 on line 272). This information allows retrieving the Fiacre source code information directly from the TTS. Finally, the path from the TTS model to the debugging model is complete, as shown on Figure 5: all user requirements are satisfied.

3.4 Comparison of solutions

In this subsection, we compare the previous solutions by estimating the cost required to recover information degraded during the compilation phase. Figure 5 illustrates this process. In this example, we consider a Fiacre source code containing a process, proc, instantiated in the compo component. The proc process has a transition containing a non-deterministic statement in which each choice contains the same source code block (x:=x+1; to s1). Let us consider the following user requirements:

- **REQ-a**: the FS shall display the executed Fiacre statements.
- **REQ-b**: the FS shall display the executed Fiacre statements in the source model.

The problem consists in satisfying those requirements when the TTS transition proc_1_t1 is fired during animation. First, we generate the abstract syntax tree (AST) of the Fiacre model using Xtext. This AST is flattened in order to generate the TTS transitions. Flattening consists in assigning an identifier number to each component/process instance according to its rank in the container component. A similar flattening activity is performed on the choices of the non-deterministic statements in order to generate the body of each TTS transition. As shown on Figure 5, The flattened Fiacre model represents a pivot model between the Fiacre and TTS levels. To satisfy REQ-a, solution 1 consists in completing the parsing and numbering activities by correlating the different available information (proc, 1 and t1) to the generated ones in the flattened Fiacre model. This process allows identifying the concerned TTS transition and thus extracting the corresponding statements. Moreover, as
the source code information (Line 6, Line 11, etc.) is already available in the identified TTS transition after the parsing, numbering, correlation and extracting activities, satisfying REQ-a and REQ-b represent the same costs. For solution 2, the cost to satisfy REQ-a is negligible because the text of the executed Fiacre statements are already available in the TTS transition. However, the cost to satisfy REQ-b is the same as for solution 1. For solution 3, the cost to satisfy all requirements is negligible because the new generated traceability information is sufficient to identify the related information at the source code level.

Therefore, solution 3 is adopted now to develop an integrated development environment for the Fiacre language because it can satisfy all end-user requirements on the one hand, and it represents the lowest cost to resolve Fiacre to TTS mappings on the other.

5 CONCLUSION AND FUTURE WORK

In this paper, we shared our experience about reusing existing low-level formal verification and validation tools in order to provide model simulation capabilities to the end-user. This work enabled us to develop a Fiacre simulator which is the result of a long research to hide all TTS information to the Fiacre end-user during the animation of his model. This work has resulted in the implementation of a Fiacre simulator tool\footnote{A demo of the simulator can be found here} This is part of an ongoing work to develop a complete Fiacre IDE that will eventually integrate advanced features of model animation like the guided-simulation and the multi-branch simulation.

REFERENCES


