Towards the qualification of an AADL model transformation tool with contracts

Guillaume Brau[1,2], Christophe Garion[1], and Jérôme Hugues[1]

[1] Université Fédérale Toulouse Midi-Pyrénées – ISAE-SUPAERO, DISC
10 avenue E. Belin, 31055 Toulouse, France.
{christophe.garion, jerome.hugues}@isae-supaero.fr
[2] LAAS-CNRS, IC/ISI
7 avenue du Colonel Roche 31031 Toulouse, France.
guillaume.brau@laas.fr

Abstract. Model-Based Engineering (MBE) can be used to build complex and critical systems. At the core of MBE, model transformation allows for the automatic processing of models to automatically generate code of the application or to perform analysis. In the context of safety-critical applications, qualification of MBE tools must provide evidence that the model transformation process is "correct". Its implementation must (i) document the transformation process itself, and support validation (ii) to infer properties on this process. We propose to use *contracts* to support qualification of the model transformation process. Contracts, by providing a formal specification of a model transformation, can be used in different ways, either to detect contract violations at run time or to demonstrate properties of the model transformation at the design time. This approach has been experimented in the context of Ocarina, an AADL toolset (Architecture Analysis and Design Language).

Keywords: embedded systems, model-based engineering, model transformation, qualification, contracts, AADL

1 Introduction

Building embedded systems is a complex and critical task that requires dedicated engineering techniques in order to fulfill both the functional and non-functional requirements (e.g. performance, timing, safety or security requirements). The process of developing embedded systems can be eased by tools implementing a Model-Based Engineering (MBE) approach. In such an approach, models of the system are created, then verified and finally transformed into an implementation of the system as a collection of executable programs and configuration files. An important challenge now is to *qualify* MBE tools for use by engineers, especially in the aerospace domain.

In this article, we concentrate on model-based engineering using the Architecture Analysis and Design Language (AADL), a language dedicated to the architecture of real-time safety-critical embedded systems [11, 27]. With AADL, system designers can use tools such as OSATE [30] to create a model of the system architecture, both on the software and hardware point of views. The Ocarina toolset [18] can then be used to either map the AADL model to analysis tools like model checkers or schedulability analyzers, or generate C or Ada code for real-time operating systems.

At its core, Ocarina relies on model transformation to map an AADL model to analysis tools or execution platforms. An open research challenge is the qualification of such a tool that can be seen as a high-level compiler.

There is a dynamic research community studying how to use formal methods to verify model transformation (see surveys on this topic [6] or [1]). We note that these works focus on general and theoretical aspects of model transformation verification and their multiple criteria, but do not provide a straightforward solution to qualify model transformation in terms of the DO-330 aerospace standard [25] and support documents like [24].

The DO-330 standard lists a number of objectives that must be addressed by the design and implementation steps, including a precise definition of the tool operational requirements (the tool objectives) and the tool requirements (low-level activities performed by the tool). This requires a proper definition of:

1. *the subset of the input language supported by the tool*
2. *the transformation rules towards the output model*

We actually note that there is a direct relationship between these two questions: transformation rules must operate on the whole subset of the language in order to be complete, whereas the transformation rules must be self-consistent and provide a one-to-one correspondence between input and output models, and thus address a well-defined subset of the input language. The complexity of transformation rules makes this effort challenging.

Contribution The core idea presented in this paper is to use *contracts* to move towards qualification of the model transformation process. Contracts are leveraged to 1) extract the subset of the input language from the transformation rules, and 2) to analyze the consistency of these rules using contracts.

Contracts provide formal artifacts to *specify* a model transformation and to *reason* on it. A contract involves assumptions over a model to be eligible for transformation, and guarantees over the output model specifying valid results. Contract reasoning can then be used for two purposes: detect contract violations at run time and prove properties of the model transformation at the design level. In particular, we present two implementations of contracts around Ocarina: using Ada 2012 for run time checking, or using SMT solvers for proving general properties of the transformation at the design time.

Outline The paper is organized as follows. We firstly discuss related works in Section 2. Section 3 then introduces contracts with formal definitions. Section 4 presents the implementation of contracts in both Ada 2012 and SMT-LIB, and the properties that can be achieved with these implementations. We finally conclude the paper and sketch possible future works in Section 6.

2 Related works

In this section, we perform a first survey on the verficiation of model transformation, and the use of contracts to support this goal.

Verification of model transformation, in particular through formal methods, is one possible way to achieve qualification. [6] and [1] present a literature review on this topic. As [1] explains, a solution is a matching between a *transformation*, the *properties* to be verified, and the *technique* used to verify these properties. Therefore, there is a large number of approaches depending on the combination of these components that could be used to verify model transformations.

In the following, we only consider *contracts* for modeling and reasoning on model transformations, because (a) contracts are well-studied and used in software engineering [19] or system design [3, 23] and (b) Ada 2012, our target language for writing transformations, has built-in support for contracts.

Originally, assume-guarantee and contract reasoning have their roots in the Floyd-Hoare logic [12, 16]. As regards theoretical aspects, a general description of contracts together with a meta-theory can be found in [3]. A well-known application of contracts is *design-by-contract*, an approach to design software popularized by [19]. Since then, contracts have been investigated beyond the scope of computer programming, for example to design Cyber-Physical Systems [10, 28], or to manage analysis activities in Model-Based Systems Engineering [4, 26]. In particular, we note that the techniques explained in the two later works combine contracts with SMT solving (resp. SAT solving) to analyze contracts at different levels of abstraction. Contracts have also been investigated in the context of model transformation. We review hereinafter some works that use contracts to support verification of model transformations.

Cariou et al. [7–9] studied model transformation contracts written in OCL. In their approach, transformation operations are associated with contracts. In fact, contracts are constraints expressed on three kinds of components: constraints on the source model, constraints on the target model and constraints on the evolution of elements from the source to the target models. Part of these constraints can be implemented as OCL invariants and then checked using a standard OCL evaluator. The other part requires to use a set of OCL utility functions. Thus, the approach enables to ensure that a model transformation involving source and target models conforms to a set of contracts with no need to execute the transformation. Yet, this approach suffer limitations such as being applicable in a single expression context. Indeed, the use of OCL obliges to concatenate the source and target model in an amalgamated model where to express OCL invariants and perform verification. Another limitation is that the approach is applicable for endogenous model transformation only, *i.e.* models with same source and target metamodels.

The contribution in [15] is a language called PaMoMo to specify visual transformation contracts and a process to compile and verify them. At first, contracts enable to specify requirements on a model transformation through preconditions, postconditions and invariants. The definition of a contract is implementation-independent, that is a contracts is not specific to a particular transformation language. When defined later, the model transformation is to be compiled with its contract(s) into the QVT-R executable language for checking whether it fulfills the requirements or not (providing the user with information on which part of contract failed and where).

These contributions propose high-level framework to reason on transformations. Yet, we note their qualification is itself a challenge: they rely on domain-specific languages, on top of large Java-based framework built around Eclipse, or academic languages. Hence, every single elements must be carefully designed and tested. This adds a significant burden on the tool designer.

We opted for a different approach: we perform model transformation, but in an "old-school" way. We consider metamodels as regular AST of a compiler, and transformation rules as internal AST-to-AST mappings. We have implemented Ocarina, an AADL model processor, in Ada 2012. We leverage well-known compiler design principles for managing AST, and performing model transformations: the model transformation is a set of organized visitor routines that process the AST, and execute on-the-fly transformation rules; and meta-model elements are specific AST nodes. This design strategy proved its efficiency for several transformation backends (C, Ada, LNT, Petri Nets, Alloy, …).

In the next section, we present how this design choice allows us to attach executable contracts to leaf transformation rules, and how to later perform verification activities so as to extract (1) the subset of the input language required to perform the transformation, and (2) the set of operations to be performed on the input model to produce the output model. For sake of readability, we considered only the AADL to Cheddar ADL [13] set of transformation rules.

3 Transformation contracts

This section introduces "transformation contracts", the central notion in our approach. We begin with a reminder about the concept of model transformation with a special focus on its implementation in Ocarina. We then provide the formal definition of contracts that supports our work.

3.1 Model transformation

In general terms, "a model transformation is the automatic generation of a target model from a source model, according to a transformation definition" [17]. More precisely, an input model, defined by a source metamodel (or language) $\mathcal{S}_i$, is transformed into an output model, defined by a target metamodel (or language) $\mathcal{S}_o$, by executing a transformation definition.

Let us take an example involving AADL and Ocarina. An AADL model describes the architecture of a system as a hierarchy of interacting software and hardware components. For example, an UAV application can be modeled in AADL with a `process` that includes several `threads` such as the one presented in Listing 1.1. A thread behavior is defined with important timing parameters

(`Period`, `Compute Execution Time`, etc.) and refers to the actual source code (`Source_Text` and `Compute Entrypoint Source Text`).

```
-- Thread implementation instantiated as Alt_Ctrl_Th

thread implementation altitude_control_task.Impl
properties
        Dispatch_Protocol=> Periodic;
        Dispatch_Offset => 0 ms;
        Period => 250 ms;
        Compute_Execution_Time => 1478 us .. 1660 us;
        Source_Text => ("autopilot/main.c");
        Compute_Entrypoint_Source_Text =>"altitude_control_task";
end altitude_control_task.Impl;
```

Listing 1.1: Example of a `thread` implementation in AADL (taken from [5,22]).

Several analyses can be performed from this model. For example, we may want to analyze the schedulability of the modeled application, *i.e.* determine whether a set of tasks will meet their deadlines according to a given scheduling algorithm. In this case, the AADL model is to be transformed into an analysis-computable model in specialized tools as Cheddar [29] or MAST [14]. Only relevant data (in short, properties of threads, scheduling algorithms and various structural information about threads) are extracted from the AADL model and translated into a real-time task model like the one represented in Figure 1.

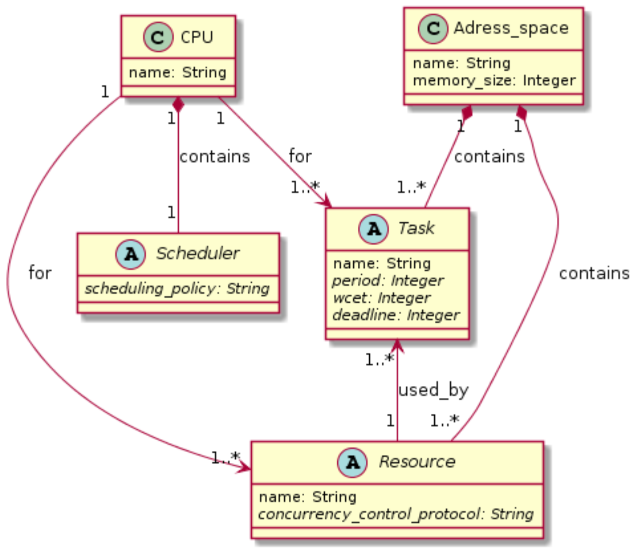

Fig. 1: Task model used to evaluate the schedulability of a real-time application.

Ocarina is the toolset used to perform transformation of AADL models. In Ocarina, a transformation definition T involves a set of *transformation rules*

$T = \{R_1, R_2, ..., R_n\}$. A transformation rule maps a subset of the source language to a subset of the target language $R : \mathcal{S}_i \mapsto \mathcal{S}_o$. For example, Ocarina executes the transformation rules described in Table 1 to translate an AADL model into its equivalent real-time task model in Cheddar ADL.

Name	Mapping	Operation
Map_Thread	R_1 : `AADLThread` $\mapsto$ `CheddarTask`	- translate every `thread` into a `task`
		- link the `thread` to a `CPU`
		- link the `thread` to an `address_space`
Map_Processor	R_2 : `AADLProc` $\mapsto$ `CheddarCPU`	- translate every `processor` into a `CPU`
Map_Process	R_3 : `AADLProcess` $\mapsto$ `CheddarAddSpace`	- translate every `process` into an `address_space`
Map_Data	R_4 : `AADLData` $\mapsto$ `CheddarResource`	- translate every shared `Data` into a `Resource`
		- link the `Data` to a `CPU`
		- link the `Data` to an `address_space`
		- link the `Data` to a list of `threads`

Table 1: Rules to transform an AADL model into a Cheddar model.

3.2 Contracts

Generally speaking, *contracts* are formal specifications for software components that use *assumptions* and *guarantees* [19]. Assumptions describe properties expected by a given component on its environment, whereas guarantees specify properties provided by the component under these assumptions.

Contracts can be used to document transformation rules (and reason on them), in particular to specify in which conditions the transformation rule apply and what is the expected result. More precisely, assumptions describe the properties of the initial model or the output model that must be true in order to apply the rule and guarantees define the properties verified by the output model after the application of the transformation rule.

Let us formalize transformation contracts. We first define the domain used to formally model the rules.

Definition 1. *A* contract domain *is a first-order signature* $\langle \mathcal{S}_i, \mathcal{S}_o, \mathcal{F}_t, \mathcal{P}_t, \mathcal{F}, \mathcal{P} \rangle$ *where*

- $\mathcal{S}_i$ *is a set of sorts typing the elements of the input model*
- $\mathcal{S}_o$ *is a set of sorts typing the elements of the output model*
- $\mathcal{F}_t$ *is a set of functions* $\mathcal{S}_i \mapsto \mathcal{S}_o$ *representing the transformation rules*
- $\mathcal{P}_t$ *is a set of unary predicates apply_t over* $\mathcal{S}_i$. *apply_t*(x) *means that transformation rule* t *has been applied on element* x.
- $\mathcal{F}$ *is a set of function symbols over* $\mathcal{S}_i$, $\mathcal{S}_o$ *and other classic sorts (integers for instance). These functions are used to describe the input model.*
- $\mathcal{P}$ *is a set of predicate symbols over* $\mathcal{S}_i \cup \mathcal{S}_o$ *representing properties of input and output models. All sorts are equiped with an binary predicate representing equality and denoted by* $=$.

Definition 2. *Let $\mathcal{D}$ be a contract domain. The language of contracts built on $\mathcal{D}$ is a many sorted first-order language with equality on sorts [31].*

For example, the transformation rule R_1 translates an `AADLThread` element into a `CheddarTask` element. In order for the output model to be valid, the Cheddar task must be linked to a Cheddar CPU obtained from an AADL processor that should be linked to the thread.

In this example, we manipulate 4 sorts: `AADLThread` and `AADLProc` in $\mathcal{S}_i$ and `CheddarTask` and `CheddarCPU` in $\mathcal{S}_o$. The transformation rule R_1 will be represented by a function R_1 : `AADLThread` $\mapsto$ `CheddarTask`. Notice that the function is *uninterpreted*: there is no particular first-order semantics associated to the function. The predicate *apply_R₁* takes an element of `AADLThread` as parameter and means that R_1 has been applied on a particular thread.

In order to be correctly applied, the thread t given as a parameter of R_1 must respect some conditions:

– if t is a periodic or a sporadic thread, then its period must be specified to be transformed into a Cheddar task.
– t must be linked to an AADL processor p and there must be a Cheddar CPU c resulting of the transformation of p. There is thus a transformation rule R_2 : `AADLProc` $\mapsto$ `CheddarCPU`.

Notice that these two conditions belong to two different families of assumptions. The first one is directly linked to the input model and thus restrict the possible input models (a periodic thread without a defined period cannot be used with this transformation). The last one specifies properties of the output model (the fact that a Cheddar CPU must already exist) by possibly using a constraint of the input model (every thread has an associated processor in an AADL model).

Considering t to be the parameter of R_1, we can thus define two assumptions for R_1:

$$\begin{aligned} A^i_{R_1}(t) \equiv\ & \mathit{dispatch\_protocol}(t) = \texttt{Periodic} \vee \mathit{dispatch\_protocol}(t) = \texttt{Periodic} \\ & \rightarrow \neg(\mathit{period}(t) = 0) \\ A^o_{R_1}(t,p,c) \equiv\ & \mathit{linked}(t,p) \wedge c = R_p(p) \end{aligned}$$

where *dispatch_protocol* is a function `AADLThread` $\mapsto$ `AADLDispatchProtocol` and `Periodic` and `Sporadic` are constants built on `AADLDispatchProtocol`, *period* is a function `AADLThread` $\mapsto$ `Int` and *linked* is a predicate on `AADLThread`$\times$ `AADLProc` specifying that a thread is linked to a particular processor in the AADL model.

Notice that $A^o_{R_1}$ has 3 free variables: t represents the AADL thread, and p and c represents respectively the AADL processor and the Cheddar CPU. This is necessary, as the Cheddar CPU will be used in the guarantee of the transformation rule, we thus need to be able to quantify existentially p and c "over" $A^o_{R_1}$.

The guarantee of R_1 should express that there is a Cheddar task that results from the transformation of the thread and this task is linked to the Cheddar processor resulting from the transformation of the AADL processor associated to the thread and the created Cheddar task cannot be a previously created Cheddar task

If t represents the parameter of the rule R_1, the guarantee $G_{R_1}(t, c)$ is then $\exists t_c \in \texttt{CheddarTask}\ t_c = R_t(t) \wedge \mathit{linked}(t_c, c) \wedge (\forall t' \in \texttt{AADLThread}\ t \neq t' \rightarrow t_c \neq R_t(t'))$ where c is the Cheddar CPU that should be associated to the task[3].

The contract of R_1 can be viewed as the following formula: $\forall t \in \texttt{AADLThread}\ \forall p \in \texttt{AADLProc}\ \forall c \in \texttt{CheddarCPU}\ A^i_{R_1}(t) \wedge A^o_{R_1}(t, p, c) \rightarrow (\mathit{apply\_R_1}(t) \rightarrow G_{R_1}(t, c))$.

The semantics of this formula is the following: for all ADDL thread t, AADL processor p and Cheddar CPU c, if t verifies the assumptions $A^i_{R_1}(t)$, p is the processor associated to t and c is the CPU obtained by transforming p, then if t is transformed then the guarantees of R_1 hold. Notice that the formula respects the intuitive meaning of the contract: if the assumptions hold, applying the transformation rule implies that the guarantee holds. If the assumptions do not hold, then applying the rule does not imply that the guarantee holds.

4 Use of contracts in Ocarina

This section presents the application of contracts in Ocarina. We firstly present the general model transformation approach. Then, we detail two use cases of contracts in Ocarina: either to detect contract violations at run time or to demonstrate general properties of the model transformation at the design level.

4.1 Approach overview

The Ocarina model transformation workflow (Figure 2) that includes contracts encompasses three levels of operation: the execution level (Level 0), the definition level (Level 1) and the analysis level (Level 2).

At Level 0, the tool executes a transformation to translate an input AADL model into an output model, e.g. a real-time task model in Cheddar ADL, or ARINC-653 code. The transformation is defined at level 1. The transformation definition includes two parts: (1) a set of transformation rules describing how a model in AADL is to be translated into a model in the target language; (2) contracts specifying the conditions to fulfill prior to execute transformation rules. Both the transformation rules and the contracts must be defined in terms of the source and target metamodels. At level 3, we are able to analyze the transformation definition to demonstrate properties of the model transformation process or find errors.

The next sections present an implementation of this workflow in Ocarina. Section 4.2 explains how the Ocarina code is reorganized to make transformation rules explicit and then extended with contracts and Section 4.3 presents the analysis of contracts using SMT solving.

[3] $t_c \in \texttt{CheddarTask}$ is a notation abuse to specify that t_c is a variable on sort `CheddarTask`.

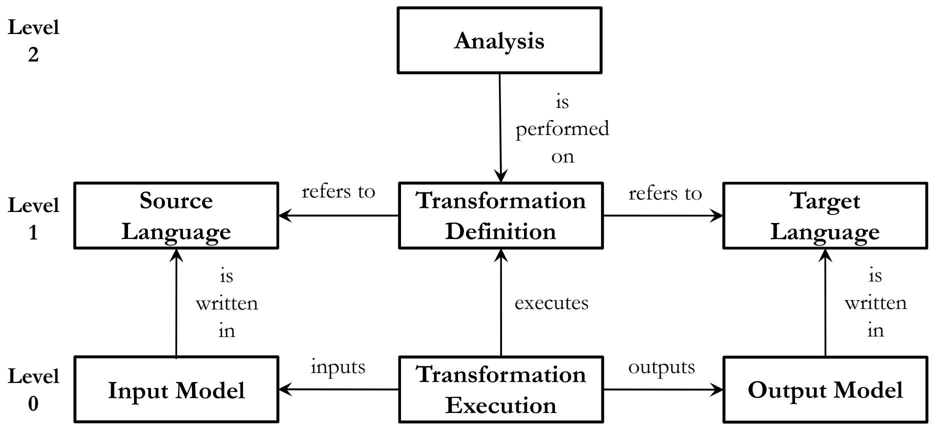

Fig. 2: Model transformation workflow implemented in Ocarina.

4.2 Integration of contracts in Ocarina

Ocarina is developed in Ada. The Ocarina code implementing model transformations can be reorganized to make both the transformation rules and the contracts explicit: 1) transformation rules can be easily written as functions, 2) we can express contracts directly in the code as Ada 2012 preconditions and postconditions, e.g. `Pre` in Listing 1.2. Notice that preconditions and postconditions in Ada are semantically equivalent to assumptions (boolean predicates on input AST nodes) and guarantees (boolean predicates on output AST nodes) in our definition of contracts.

In Listing 1.2, the transformation rule concerning threads is implemented via a dedicated `Map_Thread` function that takes a `Node_Id` parameter which is an element of the AADL Instance Tree (the Ocarina internal representation of the AADL model). In addition, the definition of the function involves a contract expressed through `Pre` and `Post`. Preconditions in the example specify requirements on the input model and can be classified in three categories. First, *typing* constraints specify the type of the input element in terms of AADL component types. For instance, the input element `E` must be a `thread`, which requirement is expressed by the `Is_Thread (E)` predicate at line 4. *Property* requirements define all the properties that must be verified by this element (with possible property values) in order to execute the transformation. For example, the assertion `Get_Thread_Dispatch_Protocol (E) \= Thread_None` states that the `Dispatch_Protocol` property must be specified for the thread element, just as `Compute_Execution_Time` and `Period` properties must be defined (lines 11 to 15). Finally, *architecture* constraints describe prerequisites at the architectural level. For example, a model is ready for transformation if the parent process of the thread is bound to a processor which means that this specific thread is to be executed on this particular processor. Thus, the precondition at line 19 asserts that there is a processor P that is bound to the parent process of the thread E. Postconditions must be expressed in a similar way on the output model.

```
    Pre =>
      (-- 1/ Typing
       AINU.Is_Thread (E) and then

       -- 2/ Property requirements
       -- The thread a) has dispatch protocol specified, b)
       -- has compute_execution_time specified, if it is
       -- either periodic or sporadic, then it has a period

       (Get_Thread_Dispatch_Protocol (E) /= Thread_None) and then
         (Get_Execution_Time (E) /= Empty_Time_Array) and then
         (if Get_Thread_Dispatch_Protocol (E) = Thread_Periodic or else
             Get_Thread_Dispatch_Protocol (E) = Thread_Sporadic then
             Get_Thread_Period (E) /= Null_Time) and then

         -- 3/ Architecture requirements
         -- a) There is a linked processor P for E
         (for some P of Processors (Get_Root_Component (E)) =>
             AINU.Is_Processor (P) and then
               P = Get_Bound_Processor
                (Corresponding_Instance
                   (Get_Container_Process (Parent_Subcomponent (E))))));
```

Listing 1.2: Definition of a transformation rule in Ada, with preconditions.

With this approach, we are able to check contracts during a transformation execution. Any condition evaluated to false at run time will raise an execution error and the transformation will terminate. Therefore, an execution of transformation rules with contracts allow us to state whether the input language and the output language meet the specification of transformation rules or not. In other words, a transformation will complete if and only if both the input and output models fulfill all the contracts.

4.3 Analysis of transformation contracts

We use the Z3 SMT solver [20] to perform analysis of transformation contracts. In order to do so, we first express input and output languages and contracts in SMT-LIB [2] and then define algorithms to verify properties with Z3. This activity is manual for the moment. Future work will consider its automation.

The SMT-LIB specification must describe contracts together with the subsets of input and output metamodels that will be required by the contracts.

First, the specification in Listing 1.3 describes elements of the input and output models (`AADLThread`, `AADLProc`, ...) with sorts and possibly with associated constants. In addition, static properties such as periods or dispatch protocols, as well as structural properties, *i.e.* relations between elements, are described through functions. For example, `dispatch_protocol` is a function that associates a thread to a dispatch protocol; `linked` is a predicate asserting that a particular thread is associated to a processor. Finally, assertions involving sorts and functions are added to specify all the corrects instances of the metamodels, *i.e.* valid models. For instance, a valid model must respect the following assertion: *"every AADL thread in the set of AADL threads must be linked to an AADL processor in the set of AADL processors"*, which is expressed at line 16.

```
; input model
;; input sorts and constants
(declare-sort AADLProc 0)
(declare-sort AADLThread 0)
(declare-sort AADLDispatchProtocol 0)
(declare-const periodic AADLDispatchProtocol)
(declare-const sporadic AADLDispatchProtocol)
(declare-const aperiodic AADLDispatchProtocol)

;; predicates and functions on input sorts
(declare-fun linked (AADLThread AADLProc) Bool)
(declare-fun dispatch_protocol (AADLThread) AADLDispatchProtocol)
(declare-fun period (AADLThread) Int)

;; constraints on input sorts
(assert (forall ((t AADLThread))
        (exists ((p AADLProc)) (linked t p))))
```

Listing 1.3: Definition of the input model in SMT-LIB using sorts, constants, predicates, functions and constraints.

The specification is then completed with all contracts. For instance, the assumptions, guarantees and contract of rule R_1 tranforming AADL thread to Cheddar tasks is presented in Listing 1.4. This is a direct traduction of contract modeling presented in Section 3.2.

The properties we want to analyze are the following:

- *executability* of the transformation: all the defined rules can be executed,
- *determinism* of the transformation: the behavior is fully deterministic, only one rule can be fired at a time,
- *non-redundancy*: each rule supports a specific objective.

Analysis of these properties is achieved by checking satisfiability of our previous SMT-LIB modeling *under assumptions* [21]. In such an approach, analysis of a given property is based on a set of assumptions that hold for a specific invocation of the solver. The specification must therefore be completed with boolean variables `XXX-trigger` to take particular formulas into account in the SMT solver. For instance, the assertion

```
(assert (or (not pre-R1-input-trigger)
            (forall ((thread AADLThread)) (pre_R1_input thread))))
```

defines a trigger for the contract assumptions of the rule R_1 about the input model. If we want to use this assumption in an analysis, we just need to check satisfiability under the assumption `pre-R1-input-trigger`.

Executability. A transformation is *executable* if all rules can be, that is all contracts' assumptions can be satisfied at run time. In addition to checking contracts at run time (see Section 4.2), we must pay particular attention to assumptions of transformation rules that are guaranteed by other transformation rules. This is the case for instance when a rule requires an element produced by another rule. For example, the `Map_Thread` transformation rule translates an `AADLThread` into a `CheddarTask` and link it to a `CheddarCPU`. To succeed, the rule requires a suitable `CheddarCPU` to be linked to the task. Thus, we must prove that there exists a

```
; AADL thread transformation rule
;; mapping function for transformation
(declare-fun apply_R1 (AADLThread) Bool)

;; mapping function for transformation
(declare-fun R1 (AADLThread) CheddarTask)

;; assumptions
(define-fun pre_R1_input ((thread AADLThread)) Bool
            (implies (or (= (dispatch_protocol thread) periodic)
                         (= (dispatch_protocol thread) sporadic))
                     (not (= (period thread) 0))))
(define-fun pre_R1_output
             ((thread AADLThread) (proc AADLProc) (cpu CheddarCPU)) Bool
             (and (linked thread proc)
                  (= cpu (R2 proc))))
(define-fun pre_R1
    ((thread AADLThread) (proc AADLProc) (cpu CheddarCPU)) Bool
    (and (pre_R1_input thread) (pre_R1_output thread proc cpu)))

;; guarantees
(define-fun post_R1
    ((thread AADLThread) (cpu CheddarCPU)) Bool
    (exists ((task CheddarTask))
       (and
        (= task (R1 thread))
        (linked task cpu)
        (forall ((thread2 AADLThread))
                (implies (not (= thread thread2))
                         (not (= task (R1 thread))))))))
```

Listing 1.4: Contract for the transformation rule R_1 with assumptions and guarantees.

transformation rule in the transformation definition that meets this requirement. In our example, this is achieved through another rule named `Map_Processor` that translates every `AADLProc` into a `CheddarCPU`.

Executability can be verified by checking for each contract that its assumptions can be obtained using the properties of the input and output models and the guarantees of the other contracts. The procedure is described in Algorithm 1 and is intuitively the following: assert the *negation* of the each atomic assumption of the given contract, assert the guarantees of the other contracts and ask the SMT solver to check satisfiability of the set of produced assertions (with constraints on input and output models). If the solver answers `UNSAT`, then the transformation is executable, as it means that the assumption of the contract is falsified by some guarantee or domain constraint, otherwise the assumption cannot be obtained.

Algorithm 1: Check executability

```
Data: An ordered list of contracts $\mathcal{C}$.
Result: YES if the transformation is executable, otherwise the assumption
        that cannot be obtained and its contract
1  begin
2      foreach $C_i \in \mathcal{C}$ do
3          foreach $C_j \in \mathcal{C} |\ C_i \neq C_j$ do
4              assert guarantees $G_j$ of $C_j$ in the solver
5          foreach assumption $A_j$ of $C_i$ do
6              assert $\neg A_j$ in the solver
7              if SMT check is SAT then
8                  return $(C_i, A_j)$
9              else
10                 remove $\neg A_j$ from the solver
11         remove all guarantees previously asserted
12     return YES
```

Determinism. The transformation process is *deterministic* if, at run time, only one rule can be fired on a particular element. As corollary, nondeterminism occurs when more than one transformation rules can be fired at the same time because these rules have compatible assumptions. In the same spirit than the previous algorithm, determinism can be checked by asserting, for each contract, the negation of the assumptions of the rule together with other contracts assumptions one by one, as described in Algorithm 2. The transformation is deterministic if the SMT solver check returns `SAT`.

Non-redundancy. We expect each rule to do a specific job in the model transformation process. Two rules doing the same job is a design error because it

Algorithm 2: Check determinism

```
Data: An ordered list of contracts $\mathcal{C}$.
Result: YES if the trans. is deterministic, otherwise the redundant contracts.
1  begin
2      foreach $C_i \in \mathcal{C}$ do
3          assert assumption $\neg A_i$ of $C_i$ in the solver
4          foreach $C_j \in \mathcal{C} | \; C_i \neq C_j$ do
5              assert assumption $A_j$ of $C_j$ in the solver
6              if SMT check is UNSAT then
7                  return $((C_i, C_j))$
8              else
9                  remove $A_j$ from the solver
10         remove $A_i$ from the solver
11     return YES
```

is a replicated, possibly unused, procedure or can cause indeterminate behaviors. Again, non-redundancy can be checked by considering, for each contract, the negation of its guarantees together with all other contracts guarantees, as described in Algorithm 3.

Algorithm 3: Check non-redundancy

```
Data: An ordered list of contracts $\mathcal{C}$.
Result: YES if there is no redundancy, otherwise the contract that is
        redundant.
1  begin
2      foreach $C_i \in \mathcal{C}$ do
3          assert $\neg G_i$ for guarantee $G_i$ of $C_i$ in the solver
4          foreach $C_j \in \mathcal{C} | \; C_i \neq C_j$ do
5              assert guarantees $G_j$ of $C_j$ in the solver
6          if SMT check is UNSAT then
7              return $C_i$
8          else
9              remove all guarantees previously asserted
10     return YES
```

5 Lessons learnt

We applied this approach to analyse the AADL-to-Cheddar backend. This backend is made of 6 rules, one per input element to map (processor, threads, ports, ...). We note that the complexity of transformation rules is similar in other Ocarina backends. This experiment allowed us to evaluate our contribution on a restricted set of rules.

We have defined contracts for these rules both in Ada2012 and in SMT-Lib. Contracts are integrated in the Ocarina code written in Ada for run time verification, whereas the analysis of contracts is performed through the SMT-LIB language and the Z3 SMT solver.

Contracts can be used in two modes. At run time, we are able to detect contract violations, thereby ensuring that the transformation process will fulfill all the transformation contracts. At the design level, we are able to analyze the transformation contracts to demonstrate several properties of the model transformation process: *executability* and the *determinism* of the transformation process, and *non-redundancy* of transformation rules.

In its current form, all SMT rules are proved by Z3, and corresponding Ada2012 contracts are correctly processed. Z3 processing time for all rules is a couple of seconds. Some of the checking algoritm have a quadratic compleixty in the number of rules to be processed. Thus, we expect a significant increase as the number of rules will increase.

We note that the Ada2012 contracts are close to the SMT-Lib ones, allowing for traceability review. We will also evalute the capability to automatically generate SMT-Lib elements from Ada2012 ones. The latter being the one closer to the tool developer habits.

6 Conclusion and perspectives

This paper dealt with the problem of qualifying model-based engineering tools, in particular tools that use model transformations. To justify the transformation process itself, we investigated the use of contracts in two dimensions: first to attach contract at run-time using Ada2012 pre/post condition mechanisms; then to express contracts of transformation rules using SMT.

We presented several algorithms to analyze the *completeness* and the *determinism* of the transformation process, and *non-redundancy* of transformation rules to assess the consistency of the transformation rules. We applied this approach to the AADL-to-Cheddar backend of the Ocarina toolchain. This approach allowed us to better characterize the subset of the input languages being used by this backend, and to assess its completeness.

Future works may take several directions. A first task will be to improve the connection between Ada 2012 pre/post conditions and SMT-Lib rules. In fact, we will be able to automatically extract contracts from the Ocarina code in Ada so as to generate the SMT-LIB specification that interfaces with SMT solvers.

A further task will be to perform a thorough review of the DO330 standard regarding tool qualificaiton, and propose a stronger connections between engineering artefacts (rules, pre/post conditions) and certification objectives.

References

1. Amrani, M., Combemale, B., Lúcio, L., Selim, G.M., Dingel, J., Le Traon, Y., Vangheluwe, H., Cordy, J.R.: Formal verification techniques for model transformations: A tridimensional classification. Journal of Object Technology 14(3) (2015)

2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016), `http://www.SMT-LIB.org`
3. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.: Contracts for systems design: Theory. Tech. rep., Inria Rennes Bretagne Atlantique (2015)
4. Brau, G., Hugues, J., Navet, N.: A contract-based approach for goal-driven analysis. In: 18th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC). IEEE (2015)
5. Brau, G.: Integration of the Analysis of Non-Functional Properties in Model-Driven Engineering for Embedded Systems. Ph.D. thesis, University of Luxembourg (March 2017)
6. Calegari, D., Szasz, N.: Verification of model transformations: A survey of the state-of-the-art. Electronic notes in theoretical computer science 292, 5–25 (2013)
7. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: Ocl contracts for the verification of model transformations. Electronic Communications of the EASST 24 (2010)
8. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: Model transformation contracts and their definition in uml and ocl. Techn. Ber 8, 125–127 (2004)
9. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: Ocl for the specification of model transformation contracts. In: OCL and Model Driven Engineering, UML 2004 Conference Workshop. vol. 12, pp. 69–83 (2004)
10. Derler, P., Lee, E.A., Tripakis, S., Törngren, M.: Cyber-physical system design contracts. In: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems. pp. 109–118. ACM (2013)
11. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)
12. Floyd, R.W.: Assigning meanings to programs. Mathematical aspects of computer science 19(19-32), 1 (1967)
13. Fotsing, C., Singhoff, F., Plantec, A., Gaudel, V., Rubini, S., Li, S., Tran, H.N., Lemarchand, L., Dissaux, P., Legrand, J.: Cheddar architecture description language. Lab-STICC technical report (2014)
14. González Harbour, M., García, J.G., Gutiérrez, J.P., Moyano, J.D.: Mast: Modeling and analysis suite for real time applications. In: 13th Euromicro Conference on Real-Time Systems (ECRTS). pp. 125–134. IEEE (2001), software available at `http://mast.unican.es/`
15. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Automated Software Engineering 20(1), 5–46 (2013)
16. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
17. Kleppe, A.G., Warmer, J.B., Bast, W.: MDA explained: the model driven architecture: practice and promise. Addison-Wesley Professional (2003)
18. Lasnier, G., Zalila, B., Pautet, L., Hugues, J.: Ocarina : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In: 14th International Conference on Reliable Software Technologies Ada-Europe. Springer (2009), software available at `http://www.openaadl.org/ocarina.html`
19. Meyer, B.: Applying "Design by Contract". Computer 25(10), 40–51 (1992)
20. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of System. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), `http://dx.doi.org/10.1007/978-3-540-78800-3_24`

21. Nadel, A., Ryvchin, V.: Efficient SAT Solving under Assumptions, pp. 242–255. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-31612-8_19
22. Nemer, F., Cassé, H., Sainrat, P., Bahsoun, J.P., De Michiel, M.: Papabench: a free real-time benchmark. In: 6th International Workshop on Worst-Case Execution Time Analysis (WCET) (2006)
23. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer International Publishing (2018), https://doi.org/10.1007/978-3-319-63588-0
24. Pothon, F.: DO-330/ED-215 – benefits of the new tool qualification document (2013), https://www.adacore.com/uploads_gems/do-330-ed-215-tool-qualification-document.pdf
25. RTCA: Do-330 software tool qualification considerations (Dec 2011)
26. Ruchkin, I., De Niz, D., Chaki, S., Garlan, D.: Contract-based integration of cyber-physical analyses. In: 14th International Conference on Embedded Software (EMSOFT). p. 23. ACM (2014)
27. SAE International: Architecture Analysis and Design Language (AADL) AS-5506A (2009)
28. Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming dr. frankenstein: Contract-based design for cyber-physical systems*. European journal of control 18(3), 217–238 (2012)
29. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: a flexible real time scheduling framework. In: McCormick, J.W., Sward, R.E. (eds.) Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies 2004, Atlanta, GA, USA, November 14-14, 2004. pp. 1–8. ACM (2004), https://doi.org/10.1145/1032297.1032298
30. Software Engineering Institute: OSATE2 : An open-source tool platform for AADLv2. https://wiki.sei.cmu.edu/aadl/index.php/Osate_2 (june 2016)
31. Wang, H.: Logic of many-sorted theories 17(02), 105–116 (Jun 1952), http://dx.doi.org/10.2307/2266241