Contract-Based Verification of Model Transformations: A Formally Founded Approach

Guillaume Brau, Mohammed Foughali

Contract-Based Verification of Model Transformations: A Formally Founded Approach

Guillaume Brau*
LAAS-CNRS, Université de Toulouse, CNRS, France
gbrau@laas.fr

Mohammed Foughali*
Université Grenoble Alpes, CNRS, VERIMAG, France
mohammed.foughali@univ-grenoble-alpes.fr

ABSTRACT

In safety-critical applications, using a *Model-Driven Engineering (MDE)* approach requires a high-level of trust in its underlying *model transformations*, *i.e.* the latter's *correctness* should be verified formally. Yet, the applicability of formal methods to transformations correctness remains limited due to the absence of formal foundations of popular MDE languages and frameworks such as AADL and SysML. In this paper, we propose a formally founded environment to verify model transformations in MDE. First, we define a transformation in a formal way: this involves formalizing *input* and *output* models at some level of abstraction, as well as the *transformation rules*. Then, we build a verification environment, formalized as a *transition system (TS)*, by extending the transformation with *contracts*. Finally, we formulate and verify some correctness properties which we reduce, based on the previous steps, to reachability properties over the TS. We show how our approach can be implemented in Ocarina, an open-source transformation tool for AADL, and how it applies, for example, to build correct transformations from AADL models to their Cheddar ADL counterpart.

KEYWORDS

model-driven engineering, model transformation, formal methods

1 INTRODUCTION

Designing, developing and implementing complex systems can be eased by Model-Driven Engineering (MDE) approaches in which *model transformations* play a prominent role [30]. A transformation operates on an *input model* to produce an *output model*. Moreover, a transformation has an *intent* [20], typically enabling some activity on the output model that is unfeasible on the input model (e.g. transforming an AADL [11] model into Petri nets enables model checking on the latter). In safety-critical applications, model transformation *correctness* is a key property.

Model transformation correctness englobes a large number of properties the relevance of which vary according to the application at hand [23, 24]. In this paper, we propose a focus on two correctness properties. First, the input model must be "transformable" w.r.t the transformation intent (*prop1*). As an example of *prop1*, if the intent of the transformation is to carry out real-time analysis, a given input model must allow to produce an output model exempt from anomalies such as a periodic task with no period specified. Second, the output model must be "equivalent" to the input model (*prop2*). An example of *prop2* is an input model in UML and an output model in Java where the latter contains all the classes specified in the former.

Once defined, correctness properties must be verified in a rigorous manner, *i.e.* through integrating *formal methods* in MDE. Such integration remains however limited due to the absence of formal foundations in popular languages and frameworks used in MDE such as AADL [11], which is in fact a consequence of software engineers and developers detaining little knowledge of formal methods. As a side effect, "classical" verification of correctness properties, proposed from a formal-method expert point of view, focuses on the equivalence between input and output models (*prop2* above) following a "correctness types" [21]. For instance, Adam et al. [1] perform a series of (robotic) model-to-model transformations, where only *syntactic correctness* (the simplest type and thus "highest level" of correctness) is verified. At the opposite "extreme", Foughali et al. [13–15] ensure *semantic correctness* between robotic models and their timed-automata output. Syntactic and semantic correctness verification approaches have each their own pros and cons, but share the disadvantages of being not accessible to software engineers and leaving property *prop1* unattended. *Contracts*, e.g. with pre- and post-conditions, are a promising alternative: they are well-known by software engineers [22] and may be tailored to reason on both *prop1* and *prop2*. However, contracts definition and deployment remain mostly informal and/or attached to particular languages/tools (Section 7).

In this paper, we propose a contract-based approach to verify both *prop1* and *prop2* of correctness of model transformations in MDE. Our solution aims at overcoming the challenges described above by enabling fully formal yet engineer-friendly verification. We formally define a transformation as an input model, an output model and a set of rules. Then, we extend the transformation with contracts to build a lightweight environment formalized as a Transition System TS. Such environment enables the verification of both (i) *prop1*, equivalent to the "transformability" of the input model (w.r.t the transformation intent) and (ii) *prop2*, equivalent to the correct implementation and execution of the transformation rules. Verifying *prop1* and *prop2* boils down to checking contracts *assumptions* and *guarantees*, which we reduce to reachability properties over the TS "at runtime", *i.e.* when the transformation is performed on some input model instance. To remain engineer friendly, the formal theory is mirrored with an implementation in a well-known programming language (e.g. Ada). While our approach is presented with a particular focus on the AADL to Cheddar ADL [31] transformation and implemented in Ocarina [19], it remains generalizable to e.g. other input and output models (Section 6).

The rest of this paper is organized as follows. In Section 2, we provide an introduction to AADL and Cheddar ADL, the source and target languages considered as an example of transformation in this paper. Section 3 deals with the formalization of the transformation and Section 4 presents the verification environment built from the formalized transformation extended with contracts. An application to Ocarina is presented in Section 5 with an implementation and a case study. Finally, we discuss the generalizability of our approach

*Authors contributed equally.

(Section 6), compare it to related work (Section 7) and conclude with possible directions for future work in Section 8.

2 PRELIMINARIES

For illustration in this paper, we focus on *AADL2Cheddar*, the transformation from AADL (Section 2.1) to Cheddar ADL (Section 2.2). We consider a subset of AADL and Cheddar where we keep only the entities/mechanisms that are necessary for *AADL2Cheddar*.

2.1 AADL

AADL (Architecture Analysis and Design Language) is a standardized architecture description language for the modeling, analysis, and code generation of real-time, embedded and concurrent computer systems [11]. An AADL model describes the architecture of a *system* with *components*. Components can describe *software* or *hardware* entities that can be hierarchically assembled to build complex systems describing possible *links* between their building components. In addition, *properties* can be defined for components. Let us review these elements in greater detail.

Components. Software components may encapsulate components called `processes`, each process describing a part of the program being executed and encapsulating itself (sub)components: `threads` of execution and possibly `data` structures. Hardware components, on the other hand, include components known as `memories` and `processors` specifying physical storage and execution resources. Physical memory components do not intervene in *AADL2Cheddar* and are therefore discarded. Figure 1 shows a simple AADL example of a system component `s1` which is built from:

- software: with one process `p1`, encapsulating two `threads` of execution (`th1`, `th2`) and one `data` structure (`d1`),
- hardware: represented by one `processor` component `pr1`.

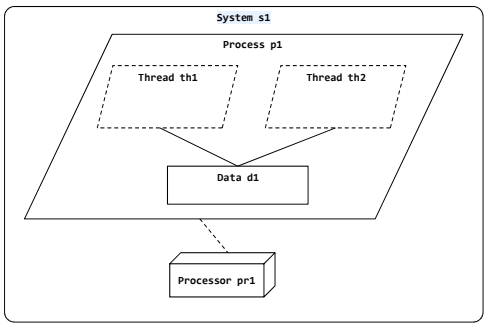

Figure 1: AADL model example.

```
1  thread th1
2  features
3    d1_access : requires data access d1;
4  properties
5    Dispatch_Protocol=> Periodic;
6    Dispatch_Offset => 2 ms;
7    Period => 25 ms;
8    Compute_Execution_Time => 1478 us .. 1660 us;
9    Deadline => 10 ms;
10 end th1;
```

Listing 1: An example of a `thread` declaration.

AADL allows to describe a system that includes several processes and processors, e.g. to represent multiprocessor systems. The composition of systems is also possible, e.g. to describe interconnected systems. However, since *AADL2Cheddar* applies to one system at a time, we restrict an AADL model to only one system, which can still have any number of `processes` and `processors`.

A component declaration is textual and includes *features* and *properties*, as we can see in the declaration of the thread `th1` in Listing 1. Features provide the interface of a component, thus allowing to link components, while properties define the characteristics of a component. More explanation below.

Links. A link can be either a *connection* or a *binding* (both connections and bindings are optional).

Connections represent logical flows (e.g. event and/or data) between components through their features. The *access* feature is reserved to link threads with data. For instance, in Figure 1, both threads `th1` and `th2` are *connected* to the data structure `d1` within the process `p1` (solid-line links): the connection between `th1` and the data `d1` occurs via the feature `d1_access` of `th1` specified in Listing 1 (line 5). A component may be involved in any number of connections. Since only access connections are relevant to *AADL2Cheddar*, we restrict connections in AADL to only access connections, to which we may thus refer simply as connections. Also, in the context of *AADL2Cheddar*, connections are *intra-process* only, *i.e.* between threads and data of the same process.

Bindings, on the other hand, link software to hardware components. For example, in Figure 1, process `p1` is bound to processor `pr1` which means that `p1` is to be executed on `pr1`. AADL bindings are compatible with *partitioned scheduling*: a process can be bound to only one processor whereas a processor can be bound to several processes (*i.e.* may execute several processes).

Properties. The description of a system is completed with *properties* (optional). For example, the properties of `th1` in Listing 1 provide timing information about the thread. A `Periodic` `Dispatch_Protocol` (line 7) means that the thread jobs occur on a regular basis, with a constant interval of time between jobs called the `Period` (line 9) and possibly a `Dispatch_Offset` (line 8) from the time origin, and a job duration is within a time interval called `Compute_Execution_Time` (line 10). A temporal constraint may be specified for the thread as a `Deadline` which is the time by which the thread must be completed (line 11). We see in Listing 1 that properties are typed: `Dispatch_Protocol` must be within an enumeration of *Supported_Dispatch_Protocols*, `Period` has a time value that is an *aadlinteger* plus a time unit (which itself may be defined from the base picosecond (ps) time unit, e.g. $us => ps * 1000000$), `Compute_Execution_Time` must be of "range of time" value, etc. Since timing constraints are – naturally – positive, we restrict the use of *aadlinteger* to naturals when describing time.

2.2 Cheddar ADL

Cheddar ADL is the modeling language of the Cheddar tool [31]. In the remainder of this paper, we refer to Cheddar ADL simply as Cheddar and use "Cheddar tool" to describe its underlying analysis tool. Cheddar allows to describe a "real-time task model" [6] on which schedulability analysis can be performed. A Cheddar model is "flat" (no hierarchies) and is made of *entities*, *links* and *parameters*, which we represent using a class diagram (Figure 2). As for AADL,

we consider in this paper a subset of Cheddar including all the elements needed for *AADL2Cheddar*.

Entities and links. Entities and links form a *real-time application*. The latter is made of:

- a number of `tasks` executing on `execution_unit(s)`, and possibly sharing some `resources`,
- a number of `address_spaces` for tasks and resources.

A real-time task is the basic entity of a real-time application. It represents a set of program instructions to be executed by execution unit(s). In this model, an execution unit is an abstraction of both the hardware required to execute the tasks (*i.e.* the central processing unit, CPU) and the software needed to build an execution order of the tasks (*i.e.* the scheduler). In addition, tasks can optionally share `resources` such as shared memories. `Address spaces` are virtual memory spaces allocated to tasks and resources.

Links define relations between entities, e.g. execution unit(s) to which a task is statically allocated (affinity). The cardinalities of associations in Figure 2 reflect restrictions on links, e.g. a task/resource cannot be allocated to more than one address space.

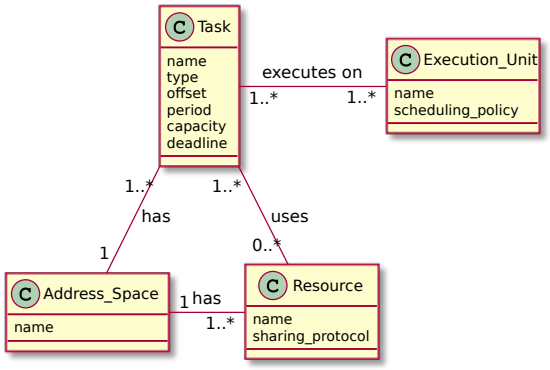

Figure 2: Elements of a real-time application in Cheddar.

Parameters. Entities of a real-time application can (optionally) have various parameters:

- task parameters: a constant interval of time between task jobs called the `period` according to the task type, a `capacity` (or worst-case execution time) for a task job, an `offset` that is an amount of time to the first release of the task, and a `deadline` to be respected,
- execution unit parameters: a `scheduling_policy` that dynamically assigns tasks to execution units as the execution goes (within the allowed static affinity, defined by links between tasks and execution units as explained above),
- resource parameters: a `sharing_protocol` to manage concurrent access to the shared resources between tasks.

Parameters are defined with usual types available in programming languages, e.g. unsigned integers for time-related parameters and enumerations for available task types, scheduling policies and sharing protocols.

3 *AADL2CHEDDAR* FORMALIZATION

The first step of our approach consists in formalizing the transformation (in this case, *AADL2Cheddar*). Thus, we formalize, at some level of abstraction, both the input and output model: AADL (Section 3.2) and Cheddar (Section 3.3), described informally in Section 2. Then, we present and formalize *AADL2Cheddar*: we define its rules and develop its formal semantics (Section 3.4). Before we get into the details, we first introduce some basic definitions and notations which we will use throughout the rest of this paper.

3.1 Basic definitions and notations

We first remind the notions of binary relations, partial functions and functions (Section 3.1.1), then provide definitions (simplified from [26]) of Transition Systems (Section 3.1.2) and Transition Diagrams (Section 3.1.3).

3.1.1 *Relations and functions.*

A *binary relation* $R \subseteq X \times Y$ is a set of ordered pairs $(x \in X, y \in Y)$.

R is called a *partial function* iff it satisfies the following: $\forall x \in X, y, y' \in Y : (x, y) \in R \wedge (x, y') \in R \Rightarrow y = y'$. That is, a *partial function* is a special type of binary relation that maps each element of X to at most one element of Y, we may then write $R : X \nrightarrow Y$. X (resp. Y) is the *domain* (resp. *codomain*) of R, and $y \in Y$ is the *image* of $x \in X$ iff $(x, y) \in R$. A partial function does not require that each element in X has an image in Y. $X_d \subseteq X$, the *domain of definition* of R, is the largest subset of X each element of which has an image in Y through R, that is: $\forall x \in X_d \, \exists y \in Y \; s.t. \; (x, y) \in R$ and $\forall x \in X \backslash X_d \nexists y \in Y \; s.t. \; (x, y) \in R$. We say that R is *defined* over X_d and *undefined* over $X \backslash X_d$. We use the standard computer science notation $R(x) \uparrow$ (resp. $R(x) \downarrow$) to denote that R is undefined (resp. defined) at $x \in X$.

A partial function R is called a function iff $X_d = X$, *i.e.* a function is a special type of partial function that is defined over all elements of its domain X. We may then write $R : X \mapsto Y$.

3.1.2 *Transition systems.*

Syntax. A transition system (TS) is a tuple $TS = \langle U, Q, q_0, \longrightarrow \rangle$ where $U = U_{ro} \cup U_{rw}$ is a finite set of *read-only* (U_{ro}) and *read-write* (U_{rw}) variables; Q a set of states, each state $q \in Q$ provides an interpretation $q(u)$ of each $u \in U$; $q_0 \in Q$ the initial state that maps each variable $u \in U$ to its initial value u_0 (each read-only variable $u \in U_{ro}$ satisfies $q(u) = q_0(u) = u_0$ for all $q \in Q$); and $\longrightarrow$ a set of transitions where each transition $t \in \longrightarrow$ is a binary relation that maps every state $q \in Q$ to a (possibly empty) set of successors $t(q) \subseteq Q$ (we write $q \xrightarrow{t} q'$ iff $q' \in t(q)$).

Semantics. A TS evolves through taking *enabled transitions*. A transition $t \in \longrightarrow$ is enabled iff the TS is at state q and $t(q) \neq \emptyset$. After taking t, the TS reaches a state $q' \in t(q)$. We may thus define the set of *reachable states* $Q_r \subseteq Q$ s.t. state q is reachable, *i.e.* $q \in Q_r$, iff there exists a (possibly empty) sequence of transitions seq such that $q_0 \xrightarrow{seq} q$.

3.1.3 *Transition Diagram.*

Syntax. We define a graphical notation for a TS, called a Transition Diagram (TD). A *TD* is a finite directed graph with V its set of vertices and E its set of edges. *TD* operates on $X = X_{ro} \cup X_{wr}$, a finite set of variables that may be read-only (in X_{ro}) or read-write (in X_{wr}). The vertex v_0 in V is the unique initial vertex of the *TD*. If e connects vertex v_a to vertex v_b, then we may write $v_a \xrightarrow{e(g_e, op_e)} v_b$

where (i) g_e is a Boolean expression over X and (ii) op_e an atomic sequence of operations over variables in X (read-only over X_{ro}).

Semantics. Let $q(g)$ denote the truth value of guard g at state q, and $q'_{|Y} = op(q_{|Y})$ denote that the valuation of each variable $y \in Y$ at state q' agrees with the result of op over y from state q. The semantics of a TD is then given by the TS $\langle U, Q, q_0, \longrightarrow \rangle$ (Section 3.1.2) where:

- $U = X \cup \{\pi\}$ s.t. π holds the *current vertex* of the TD,
- Each state q in Q is an interpretation of variables in $X \cup \{\pi\}$,
- q_0 maps π to v_0 and each variable in X to its initial value,
- $\longrightarrow$ results from mapping each edge e in E to a transition t_e in $\longrightarrow$ as follows. If $v_a \xrightarrow{e(g_e, op_e)} v_b$ then

$$q' \in t_e(q) \Leftrightarrow \begin{cases} (1)\ (q(\pi) = v_a \wedge q'(\pi) = v_b) \wedge \\ (2)\ q(g_e) \wedge \\ (3)\ (q'_{|X} = op_e(q_{|X})) \end{cases}$$

3.2 AADL

We define an AADL model compositionally, from the *process* up to the *model*, while formalizing the description given in Section 2.1.

Definition 1. ***Process.*** *A process is a tuple $p = \langle Th, D, C \rangle$ where $Th = \{th_1, \ldots, th_{|Th|}\}$ is a set of* threads*; $D = \{d_1, \ldots, d_{|D|}\}$ a set of* data*; and $C \subseteq Th \times D$ a binary relation representing connections from threads to data.*

As explained in Section 2.1, a process is made up of two sets: threads and data. C is the set of connections within a process, defined here as a binary relation (Section 3.1.1) to make it possible for a thread to access several data and, conversely, data to be accessed by several threads, as allowed in AADL (Section 2.1). Note how process (sub)components (threads and data) features do not appear in our definition. Indeed, features are mainly introduced to define the "type" of interface through which connections are made. Since only access connections are considered (Section 2.1), features become superfluous and are consequently abstracted away.

Definition 2. ***System.*** *A system is a tuple $s = \langle P, Pr, PB \rangle$ where $P = \{p_1, \ldots, p_{|P|}\}$ is a set of processes (Definition 1);*
$Pr = \{pr_1, \ldots, pr_{|Pr|}\}$ a set of processors; and $PB : P \nrightarrow Pr$ a partial function representing processes bindings to processors.

Thus, the system is built from processes tied to hardware elements, that is processors. Defining *PB* as a partial function is deliberate, as it is neither restrictive nor permissive w.r.t AADL. Indeed, since *PB* enjoys the properties of partial functions (Section 3.1.1), (i) a process in P may not bind to more than one processor and (ii) creating a binding for a given process is optional, which agrees with the AADL restrictions and permissions (Section 2.1). Therefore, in a given AADL system, the domain of *PB* is P while its domain of definition is the largest subset of P all the elements (processes) of which are bound to processors.

Flattening a system. In order to have a direct access to sets of components at a lower level (*i.e.* at the process level), we define a *flat* AADL system. This "raw" flat representation is only provided to ease the definition of AADL properties and *AADL2Cheddar* transformation rules later in this section: it does not replace our original definition (Definition 2) which reflects the real architecture of an AADL system.

To atomize the system, we adopt a generic notation following a downward propagation of indices: subscript i of each process p_i in the processes set P (Definition 2) is propagated as a superscript to each tuple element of p_i (Definition 1), e.g. $Th^i = \{th^i_1, \ldots, th^i_{|Th^i|}\}$ is the set of threads in the i^{th} process of P, the set of processes in s. We obtain consequently $\mathcal{TH} = \bigcup_{i \in 1\ldots|P|} Th^i$, the set of all threads (in all processes) in the system, $\mathcal{D} = \bigcup_{i \in 1\ldots|P|} D^i$, the set of all data (in all processes) in the system, and $C = \bigcup_{i \in 1\ldots|P|} C^i$, the set of all connections between threads and data (in all processes) in the system. We may then define the flat system by redefining P (Definition 2), as the set of sets of all threads, data and connections in all processes, that is $P = \{\mathcal{TH}, \mathcal{D}, C\}$.

Definition 3. ***Model.*** *An AADL model is a tuple $ma = \langle s, Prop \rangle$ where s is a system (Definition 2) and Prop a set of properties.*

A property in *Prop* is a partial function with some domain X of components or subcomponents of s, *i.e.* X can be taken directly from the flat representation of s (e.g. the domain of any thread property is $\mathcal{TH}$, of any processor property is Pr, etc.). The codomain Y varies depending on the property.

Let us exemplify with thread properties. To simplify notations, we assume that time values are uniformized to the smallest time unit, which permits rewriting them in $\mathbb{N}$ (e.g. in Listing 1, all values given in *period*, *dispatch_offset*, *compute_execution_time* and *deadline* are transformed to the μs scale). Thus, the codomain is $\mathbb{N}$ for the *period* property (that is $period : \mathcal{TH} \nrightarrow \mathbb{N}$), but also for *dispatch_offset* and *deadline*. For *dispatch_protocol*, the codomain is the set $\mathtt{DP} = \{aperiodic, sporadic, periodic\}$, formalizing the *Supported_Dispatch_Protocols* enumeration (Section 2.1). Finally, *compute_execution_time* associates a thread to an interval delimited by a minimum and a maximum execution time (example in Listing 1, Section 2.1). Its codomain is thus $\mathbb{I}$, the set of closed intervals of positive reals with natural bounds, which we may define formally: $\mathbb{I} = \{[a, b] \mid a, b \in \mathbb{N} \wedge a \leq b\}$ with $[a, b] = \{x \in \mathbb{R}_{\geq 0} \mid a \leq x \leq b\}$. We may give examples of other components properties. For instance, `concurrency_protocol` has $\mathcal{D}$ (the set of all data) as a domain and $CP = \{Maximum\_Priority, Priority\_Ceiling, Spin\_Lock, Priority\_Inheritance, Protected\_Access\}$, the enumeration of supported concurrency protocols, as a codomain.

The rationale of formalizing properties as partial functions is similar to the one used in bindings: on one hand, a component cannot have two values for the same property (*e.g.* a thread cannot be periodic and sporadic at the same time) and, on the other hand, AADL properties are optional (the user may choose not to define *e.g.* the period of a sporadic thread). Similarly to bindings, the domain of definition of a property is the largest subset of its domain for which the property is defined (*e.g.* for the *deadline* property, the largest set in $\mathcal{TH}$ where a deadline is provided for each thread).

3.3 Cheddar

To formalize a Cheddar model, we adopt a similar strategy to the one used for AADL. The real-time application firstly describes the entities in Figure 2 (Section 2.2) and their relationships. Then, parameters are defined at the model level.

Definition 4. ***Real-time application.*** *A real-time application is a tuple $\sigma = \langle T, EU, Re, AS, L \rangle$ where $T = \{\tau_1, \ldots, \tau_{|T|}\}$ is a set of tasks; $EU = \{eu_1, \ldots, eu_{|EU|}\}$ a set of execution units;*

$Re = \{re_1, \ldots, re_{|R|}\}$ a set of resources; $AS = \{as_1, \ldots, as_{|AS|}\}$ a set of address spaces; and L defined as follows. $L = \{TE, TR, TA, RA\}$ is a set of binary relations $TE \subseteq T \times EU$ and $TR \subseteq T \times Re$ and partial functions $TA : T \nrightarrow AS$ and $RA : Re \nrightarrow AS$ for links between, respectively, tasks and execution units, tasks and resources, tasks and address spaces and resources and address spaces.

Thus, the entities of Cheddar and their links appearing in Figure 2 (Section 2.2) are described as a tuple. In the latter, L, representing all links, contains four relations to reflect the associations in Figure 2 and their cardinalities. In particular, *TA* and *RA* are defined as partial functions since no task (resp. resource) can be linked to more than one address space.

Definition 5. ***Cheddar model.*** *A Cheddar model is a tuple $mc = \langle \sigma, Param \rangle$ were σ is a real time application (Definition 4) and Param a set of parameters.*

Parameters in *Param* are defined as partial functions over entities of σ (codomains vary depending on the parameter). For instance, *offset*, *period*, *capacity* and *deadline* have the same domain T (the set of all tasks) and the same codomain $\mathbb{N}$. As another example, parameter *sharing_protocol* maps resources (domain *Re*) to access protocols in the set $AP = \{$*First_In_First_Out*, *Priority_Ceiling_Protocol*, *Priority_Inheritance_Protocol*$\}$.

The rationale of using partial functions is the same as in AADL properties (Section 3.2), and the domain of definition of a parameter is the largest subset of its domain such that the parameter is defined for all elements (e.g. the domain of definition of *period* is the largest subset of T where a period has been provided for each task).

3.4 *AADL2Cheddar*

Now that we have formalized AADL and Cheddar models, we may define *AADL2Cheddar* formally: we develop its formal semantics involving an input AADL model, an output Cheddar model and a number of *rules*. We start by formally defining the latter.

3.4.1 Rules. In the following definition (and the remainder of this paper henceforth), we use the set membership symbol $\in$ for tuple memberships as well[1].

Definition 6. ***Rule.*** *A transformation rule $R : X \mapsto Y$ is a function with $X \in ma$ its domain, an element of the AADL model and $Y \in mc$ its codomain, an element of the Cheddar model.*

A rule is defined as a function because, intuitively, describing a transformation rule implies (i) a unique mapping of input elements and (ii) an "all" quantifier, for instance "transform all threads into tasks" (see rule R_3 in Table 1). That is, a rule is defined over all its domain of which each element has exclusively one image, which coincides with the definition of a function (Section 3.1.1). To compute the image of an element, a rule R may need to read the elements of another rule R', that is pairs $(x, R'(x)) \in R'$, which creates a rule *dependency* (examples in Section 3.4.2).

In total, the *AADL2Cheddar* transformation includes nine transformation rules, which are enumerated in Table 1 with their domains, codomains and a brief informal explanation (examples on what each rule does are given in Section 3.4.2). To ease access to lower level entities in an input AADL model $ma = \langle s, prop \rangle$, we rely on its flat representation (definition 3).

Rule	Mapping	Object
R_1	$Pr \mapsto EU$	maps every processor to an execution unit
R_2	$P \mapsto AS$	maps every process to an address space
R_3	$\mathcal{TH} \mapsto T$	maps every thread to a task
R_4	$\mathcal{D} \mapsto Re$	maps every data to a resource
R_5	$C \mapsto TR$	maps every connection to a task-resource link
R_6	$PB \mapsto TE$	maps every processor binding to a task-EU link
R_7	$\mathcal{TH} \mapsto TA$	maps every thread to a task-address space link
R_8	$\mathcal{D} \mapsto RA$	maps every data to a resource-address space link
R_9	$Prop \mapsto Param$	maps every property to a parameter

Table 1: Rules of *AADL2Cheddar* transformation.

3.4.2 Transformation. Before we define *AADL2Cheddar* formally, let us first clarify its technical aspects: how AADL hierarchies are dealt with and how the transformation works.

Technicalities. First, we discuss architectural issues. As one may notice from the informal and formal definitions of AADL and Cheddar, the latter is flat while the former has two architectural levels (the processes level and the system level, Definition 3). To cope with this issue, we reason as follows. It is rather intuitive that the counterpart of threads in AADL is tasks in Cheddar (rule R_3) and that of data is resources (rule R_4). Now, an address space in Cheddar can be linked to many resources and many tasks, but a resource/task can be linked to only one address space (Definition 4). Dually, in AADL, the relation between a process and threads/data is a *parent-child* one (a process encapsulates threads/data, Definition 1). Thus, a process in AADL should be translated into an address space in Cheddar (rule R_2), and the fact that a thread/data is encapsulated in it is translated into a link between the process counterpart (address space) and the thread/data counterpart (task/resource) in Cheddar (rules R_7, R_8). The process level is thus atomized in Cheddar as the parent-child relations transform into links. Consequently, a binding between a process and a processor in AADL transforms simply into a link between the (task) Cheddar counterpart of each of the process encapsulated threads and the (execution unit) Cheddar counterpart of the processor (rule R_6).

Second, we explain how *AADL2Cheddar* works. The transformation applies to an input AADL model *ma* to generate an output Cheddar model *mc*. The latter, initially empty, is constructed by *executing* the rules in table 1 sequentially. Executing a rule boils down to the *application* of the function formalizing it, e.g. the execution of R_1 (Table 1) is equivalent to computing for each $pr \in Pr$ from *ma* its image $R_1(th)$, which results in a set of execution units that are then written to *mc*. We explain more formally some of *AADL2Cheddar* rules, mainly those that translate bindings, connections and properties, which need to read the elements of other rules. R_5 translates connections into task-resource links: for each connection between a thread and a data in *ma*, a link between their task and resource counterpart, already written to *mc* (after executing, respectively, R_3 and R_4) is created. More formally, for each $C \in \mathcal{C}$ such that $(th \in \mathcal{TH}, d \in \mathcal{D}) \in C$, a link is created between $\tau \in T$, the image of *th* through R_3 ($(th, \tau) \in R_3$) and $r \in Re$, the image of *d* through R_4 ($(d, r) \in R_4$). In a similar way, the execution of R_6 generates links between tasks and execution units. Finally, through executing R_9,

[1] This shortcut can be straightforwardly motivated using a set-based representation of tuples such as nested ordered pairs.

the Cheddar counterpart parameters of the properties in *ma* are generated and written to *mc* (e.g. the *dispatch_protocol* property is translated into the *scheduling_protocol* parameter). The need of a rule to read the (pairs) elements of another rule creates a rule *dependency*, which makes the order of execution of rules important (e.g. R_5 may be executed only after R_3 and R_4). To simplify the presentation, *AADL2Cheddar* rules (Table 1) are executed sequentially following the increasing order of their indices[2].

Let us now define formal syntax and semantics of *AADL2Cheddar* in line with the explanations above.

Syntax. *AADL2Cheddar* is a tuple $\langle ma, mc, \mathcal{R} \rangle$ where *ma* is the input AADL model (Definition 3), *mc* the (initially empty) output Cheddar model (Definition 5) and $\mathcal{R} = \{R_1, \ldots, R_9\}$ a set of rules (Definition 6, Table 1).

Semantics. We can represent *AADL2Cheddar* with a TD that defines its semantics (Section 3.1.3), where:

- *Variables* the set X_{ro}, *i.e.* read-only (resp. X_{rw}, *i.e.* read-write) is the input (resp. output) model *ma* (resp. *mc*),
- *Vertices* $V = \{v_0, \ldots, v_9\}$, (the transformation *steps*),
- *Edges* $E = \{e_1, \ldots, e_9\}$ s.t. each e_i in E connects vertex v_{i-1} to vertex v_i ($i \in 1..9$), *i.e.* $v_{i-1} \xrightarrow{e_i(g_{e_i}, op_{e_i})} v_i$ where g_{e_i} is a tautology and $op_{e_i} = UP_i$ the operation of updating the output model by applying rule R_i over all its domain (from *ma*) and writing the result of such application to *mc*.

Figure 3 shows a part of the TD that gives the *AADL2Cheddar* transformation semantics (the dashed edge denotes missing edges and vertices). Operations are in blue and guards are not represented (always true here). Taking an edge e_i moves the underlying TS (Section 3.1.3) from state q (with $q(\pi) = v_{i-1}$) to q' (with $q'(\pi) = v_i$) such that the set (element of *mc*) corresponding to the codomain of rule R_i is empty at q and "filled" at q': from q to q', the operation UP_i executes R_i and "fills" the corresponding element of *mc* with the result of such execution. And so, by reaching v_9, all entities in *ma* are translated and the corresponding *mc* is fully constructed. The transformation is thus terminated.

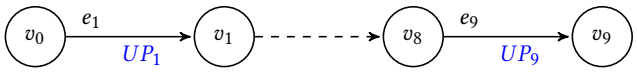

Figure 3: An example of transformation TD.

4 VERIFICATION ENVIRONMENT

Our objective is to verify properties *prop1* and *prop2* of correctness (Section 1): respectively, the transformability of the input model w.r.t transformation intent and the correctness of the transformation rules themselves. To make that possible, we build, by extending the transformation with *contracts*, a lightweight verification environment with formal semantics.

[2]Actually, since there are rules that do not depend on each other, other orders of execution are allowed, e.g. permuting R_3 and R_4 in Table 1. It can be easily shown that choosing any allowed order accordingly has no effect on the verification results, but this is omitted for the lack of space.

4.1 Contracts

A contract extends a transformation rule with an *assumption* on the input model *ma* to apply the rule and a *guarantee* provided on the output model *mc* after executing the rule.

Definition 7. ***Contract.*** *A contract is a triple $\gamma = \langle A, R, G \rangle$ with A and G are logical formulae called, respectively, an* assumption *and a* guarantee*; and R a transformation rule (Definition 6). The domain of discourse for any A (resp. G) is ma (resp. ma ∪ mc).*

Thus, a contract provides a means to reason on two types of anomalies: either in the input model (an assumption does not hold) or at the rule level (a guarantee fails). Notice how the domain of discourse of G is $ma \cup mc$ rather than just *mc*. This is because, oftentimes, a guarantee on the output model is given w.r.t the input model (see example below).

Examples. We give below two examples of contracts. The first example relates only to reasoning on the correctness of a transformation rule using a guarantee (*prop2*). For rule R_1, we may propose a contract $\gamma_1 = \{A_1, R_1, G_1\}$ where A_1 is a tautology and $G_1 = (|EU| = |Pr|)$. This means that R_1 may execute without a prior condition (A_1) and, after it executes, we have a "weak" equivalence guarantee between the input (AADL processors) and the output (Cheddar execution units) stating that the number of execution units written to *mc* is equal to the number of processors in *ma* (which can be strengthened according to the engineer needs).

In the second example, both *prop1* and *prop2* are considered. For rule R_5, we may propose contract $\gamma_5 = \{A_5, R_5, G_5\}$ where $A_5 = \forall p \in P : PB(p) \downarrow$ and $G_5 = \forall \tau \in T : TE(\tau) \downarrow$. Here, the assumption conditions the execution of R_5 to the fact that bindings are defined for all processes in the input AADL model. This is a typical example of *prop1* of correctness (Section 1), where correctness depends on the intent of the transformation: we require all processes to be bound to processors because the transformation to Cheddar is for real-time analysis purposes, and such analysis would not be possible if there is a process that is not allocated to any processor. G_5, on the other hand, is a guarantee on the correct implementation of R_5, ensuring that, after executing R_5, all tasks in the output model are bound to execution units, which gives a form of equivalence between input bindings and output links (*prop2*)[3]. We emphasize the importance of both assumptions and guarantees even if they look similar. For instance, A_5 may seem redundant because anyway G_5 will check a posteriori whether all tasks in the generated Cheddar model are linked to execution units. However, if we remove A_5 and G_5 does not hold, it will be impossible to know whether the error comes from the input AADL model or the implementation of R_5, whereas with both A_5 and G_5 localizing the source of the error is straightforward using a transition system (see below).

4.2 Building the Environment

We extend *AADL2Cheddar* with contracts (once all contracts are defined) to obtain a formal environment in which assumptions and guarantees can be verified. The transformation thus succeeds iff all contracts are *validated*, *i.e.* all assumptions and guarantees evaluate to true as the transformation goes.

[3]Note that since the elements of each pair in R_5 come from, respectively, *ma* and *mc*, all the literals of G_5 fall within the domain of discourse of a guarantee given in Definition 7, that is $ma \cup mc$

4.2.1 Syntax. The syntax of the verification environment is extended from that of *AADL2Cheddar* (Section 3.4.2): each rule in $\mathcal{R}$ is incorporated within a contract. Thus, the environment is a triple $E = \langle ma, mc, \Gamma \rangle$, where ma and mc are the input and output models of the transformation (Section 3.4.2) and $\Gamma = \{\gamma_1, \ldots, \gamma_9\}$ is the set of contracts each incorporating the rule having the same index from $\mathcal{R}$, that is $\gamma_i = \{A_i, R_i, G_i\}$ for each $i \in 1..9$.

4.2.2 Semantics. The semantics of the verification environment is obtained by extending the semantics (TD) of *AADL2Cheddar* (Section 3.4.2) so that for each rule R_i, assumption A_i (resp. guarantee G_i) is verified before (resp. after) its execution. This gives a new TD with a set of vertices V', a set of edges E' and a set of variables X' derived from the *AADL2Cheddar* TD (vertices V, edges E and variables X) following the rules below:

- *Variables*: $X' = X$ (with $X'_{ro} = X_{ro}$ and $X'_{rw} = X_{rw}$),
- *Vertices*: $V' = V \cup_{i \in 1..9} \{v_{i-1,i}\}$. Each additional vertex $v_{i-1,i}$ denotes an intermediate step between vertices v_{i-1} and v_i in V,
- *Edges*: Each edge $v_{i-1} \xrightarrow{e(g_e, op_e)} v_i$ in E is replaced with two edges $v_{i-1} \xrightarrow{e(A_i, op_e)} v_{i-1,i}$ and $v_{i-1,i} \xrightarrow{e(G_i, null)} v_i$ where *null* denotes any side-effect-free operation.

Thus, the resulting TD has the same input and output variables, and augments the original transformation TD with additional vertices, edges and guards to verify assumptions and guarantees. Figure 4 shows a part of the TD resulting from the extension of the TD in Figure 3, following the rules given in the semantics above (the dashed edge denotes missing edges and vertices, null operations and edges names are removed). Additional vertices are in red. Thus, rule R_1, for instance, will not be executed unless A_1 is true (guard in green), in which case the result of executing R_1 is written to mc (operation UP_1 in blue) and vertex $v_{0,1}$ is reached. Afterwards, v_1 may not be reached unless the guarantee following the execution of R_1 is satisfied (guard G_1 in green). Then the same behavior is repeated for all remaining rules until v_9 is reached.

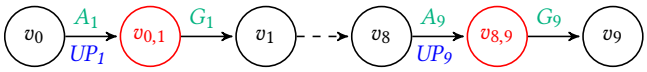

Figure 4: Example of verification environment TD.

4.2.3 Correctness verification. Using the verification environment semantics, we can reason on the transformation correctness w.r.t the transformability of the input model (assumptions) and the correctness of the execution rules implementation (guarantees) at "runtime" (*i.e.* on the fly when executing the transformation on an input model instance). For example, if G_9 is false, any state q in the underlying TS such that $q(\pi) = v_9$ (we recall that π is the variable, in the underlying TS, denoting the current vertex of the TD, Section 3.1.3) is unreachable, that is $q \notin Q_r$. Dually, if A_9 does not hold, states q s.t. $q(\pi) = v_{8,9}$ are also unreachable. And so, we can define from the set of reachable states exactly at which state the transformation failed and conclude on the error source, *i.e.* input model or rule implementation, and in the former case, more precisely from which part of the input model (e.g. threads, bindings, etc.). If any reachable state $q \in Q_r$ satisfies $q(\pi) = v_i$, this means that rule R_i is successfully executed with both A_i and G_i evaluating to true, in which case we say contract γ_i is validated.

Since the system is sequential, concluding on reachability of states is rather straightforward: if q is unreachable with $q(\pi) = v_{i-1,i}$ (resp. $q(\pi) = v_i$) then any other state q' with $q'(\pi) = v_{j-1,j}$ or $q'(\pi) = v_j$ is also unreachable for any $j \geq i$ (resp. $j > i$). Also, if there exists a reachable state q such that $q(\pi) = v_i$ where i is the largest index (9 for *AADL2Cheddar*), then the transformation is correct (all contracts are validated).

5 APPLICATION TO OCARINA

In this section, we propose an implementation of our formal environment (Section 4) within the transformation tool called Ocarina (Section 5.1), then illustrate with verifying *AADL2Cheddar* correctness on a case study (Section 5.2). All the implementation and case study elements presented in this section are publicly available at https://github.com/artxy/sac2021 with a notice to reproduce the experimentation results.

5.1 Implementation

We briefly introduce Ocarina and review the key elements of our implementation. Then, we explain how the abstract verification of reachability, explained in Section 4.2.3, can be achieved at the implementation level.

Ocarina. Ocarina is a "processor" of AADL models, written in Ada. In particular, it allows to generate, from AADL, various kinds of codes for embedded platforms and models for analysis/verification purposes such as real-time task models (e.g. for Cheddar) and Petri nets (e.g. for TINA [5]).

Ocarina is designed as a compiler. Figure 5 gives an overview of a typical process flow in Ocarina, that transforms an AADL model into a Cheddar model. Models are represented as Abstract Syntax Trees (AST) and transformation rules as AST-to-AST mappings, encoded as Ada functions. The transformation itself is then a set of organized visitor routines that process the AST and execute on-the-fly transformation rules.

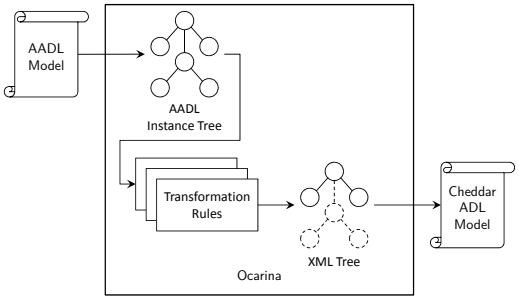

Figure 5: Process flow to transform an AADL model.

Therefore, the formal elements of *AADL2Cheddar* (Section 3.4) are implemented via three main mechanisms: ASTs (implementing the input and output models ma and mc), Ada functions (implementing the transformation rules R_1 to R_9) and a set of visitors (implementing reading the elements of ma and writing the rules execution results to mc). To deal with rule dependencies, the pairs $(x, y) \in R$ of each rule R on which at least another rule R' depends

are temporarily saved in a data structure which R' can access. Such structures are destroyed at the end of the transformation.

Now, in order to incorporate contracts, we extend their implementation, that is Ada functions, using *Ada 2012 contracts*: for each Ada function implementing rule R_i, we add a *precondition* (resp. *postcondition*) that maps A_i (resp. G_i), the assumption (resp. guarantee) of contract γ_i incorporating R_i (Section 4.2.1). This implementation coincides with the semantics of our environment in Section 4.2.2 (see below).

We have thus a full equivalent implementation to our environment in Ocarina the elements of which are further detailed below.

Models = AST. In Figure 5, an instance of model (either an AADL model in input or a Cheddar in output) is described with a tree structure, namely the AST, that is a collection of nodes (each one is identified by a `node_id`) that forms a tree structure via child-parent links descending from the root node. A node in the tree denotes an entity in the related model (e.g. a component, a connection, a property, ...), thus representing the structure of an AADL or Cheddar model, hierarchical in the case of AADL (Definition 3, Definition 5). In Figure 5, the AST of the input AADL model is an Instance Tree (on the top left), whereas the XML Tree (at the bottom right) is the AST of the output Cheddar model.

Rules = functions. Transformation rules are mapped to *functions* defined over AST nodes, thereby implementing the definition of rules (Definition 6, Section 3.4). For instance, rule R_3, that translates every thread (in an AADL model) into a task (in a Cheddar model), is implemented in a function called `Map_Thread`. A rule is "atomic" in a function: the function takes a `node_Id` in the input AST and returns the `node_Id` of the created node in the output AST. The application of a rule over its whole domain is ensured by a visitor (see below). For each rule R' that depends on the result of executing rule R, R' reads, besides the `node_Id` in the AST, the temporary data structure holding all pairs $(x, y) \in R$ as explained above.

(Ada) contracts. The function declaration is extended with preconditions and postconditions. According to the Ada 2012 Rationale [2], "a precondition is an obligation on the caller to ensure that it is true before the subprogram is called", whereas "a postcondition is an obligation on the implementer of the body to ensure that it is true on return from the subprogram". Thus, this implementation complies with the semantics of our environment (Section 4.2.2).

Let us see how this is done for e.g. assumption A_9 in contract γ_9, incorporating rule R_9 (transform all properties in *Prop* to parameters in *Param*, Table 1). Similarly to A_5 on bindings (see example in Section 4.1), A_9 stipulates that each property in *Prop* must be defined over all its domain, that is $\phi(x) \downarrow$ is satisfied for each x in X for each property $\phi : X \twoheadrightarrow Y$ in *Prop*. For e.g. the `compute_execution_time` property, that is the literal of A_9: $\forall th \in \mathcal{TH} : compute\_execution\_time(th) \downarrow$, the precondition in Ada checks, for each thread node *th*, whether the function *Get_Execution_Time*, giving the image of *th* via the property `compute_execution_time` in an array, returns a valid value (*i.e.* different from *Empty_Time_Array*, denoting that such property is not defined for *th*):

```
(Get_Execution_Time(th) /= Empty_Time_Array)
```

Transformation flow = visitors. Finally, a main *procedure* ensures the transformation flow by reading all the input AST nodes and invoking transformation rules sequentially as the visiting goes (which results in nodes creation in the output AST). Contracts are validated (or not) on the fly through preconditions and postconditions.

Verification = condition checks. If a precondition or postcondition fails, an execution error is raised as the flow is interrupted. Locating the function at which the error occurred allows to find the error source: a rule implementation or the input model. In the latter case, we may further locate at which element of the input model (threads, data, etc.) the transformation failed. This "error location" mirrors the state reached last in the underlying TS (Section 4.2.2) and enables thus an implementation-level verification of the reachability property explained in Section 4.2.3 (an example is given in Section 5.2). A transformation is then correct iff the flow terminates without errors, thus generating a complete Cheddar model.

5.2 Case Study

Let us illustrate our approach on a real AADL model describing the well-known Mars Pathfinder system[4]; with model transformation from AADL to Cheddar performed using our implementation within Ocarina (Section 5.1). We shortly present the AADL model of the system before we perform transformation verification.

5.2.1 Overview of the AADL model. The Mars Pathfinder system consists of a stationary *lander* and a *rover* named Sojourner. All the system elements are described with the usual AADL components (see Section 2). In particular, the overall `system mars_pathfinder` includes a main `process` (`prs_PSC`) describing the lander application and a `processor` (`rs_6000`) specifying the RS 6000 processor. The main (lander) process includes seven (periodic) threads, four of which may access a critical resource, described by a `data` (`data_rw`), in mutual exclusion (for read/write). The AADL model is completed with the description of connections/bindings and properties.

5.2.2 Verification. We apply our environment, implemented in Ocarina, to the AADL model of the case study as an input. We recall that our environment includes the *AADL2Cheddar* transformation extended with 9 contracts (Section 4) and implemented within Ocarina using ASTs and Ada functions and contracts (Section 5.1).

We run our environment on an initial AADL model `mars_pathfinder.init`. From the execution log (Figure 6), we see that an error is raised during the transformation of the AADL model: a precondition given at line 102 within the flow specification (`cheddar-mapping.ads`) fails. This precondition is found in the declaration of the function which implements rule R_9 and the error occurs at the line corresponding to the literal of A_9 requesting `concurrency_protocol` to be defined for all data in the input model, that is $\forall d \in \mathcal{D} : concurrency\_protocol(d) \downarrow$. Indeed, by investigating the AADL model, we find that such a property is not specified for `data_rw` which is shared between four `threads` in the main `process prs_PSC` of the `system mars_pathfinder.init`.

Once we locate the error source, we correct it by adding the missing property. Listing 2 shows an excerpt of the corrected AADL model `mars_pathfinder.correct` which extends the initial model `mars_pathfinder.init` and specifies a `concurrency_protocol`

[4]From the Mars Pathfinder mission by the NASA, https://mars.nasa.gov/mars-exploration/missions/pathfinder/(last accessed October 2020)

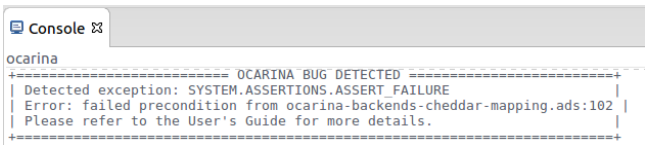

Figure 6: Transformation error raised by Ocarina.

property (value `Priority_Ceiling`) for `prs_PSC.data_rw` (`data_rw` in process `prs_PSC` in `mars_pathfinder.correct`). Now, we may re-apply our environment to the corrected AADL model. No error is raised as the flow terminates and a full Cheddar model is created.

```
1  system implementation mars_pathfinder.correct
2     extends mars_pathfinder.impl
3  properties
4    concurrency_protocol => Priority_Ceiling applies to prs_PSC.
         data_rw;
5  end mars_pathfinder.correct;
```

Listing 2: Corrected AADL model through an extension of the Mars Pathfinder system.

The error found in this example is at the input model level w.r.t the transformation intent, *i.e.* since the transformation to Cheddar is for real-time analysis purposes, the input model must define a concurrency protocol over data. Verifying the transformation allows to detect such anomaly at an early stage of the transformation and thus prevents using analysis techniques on an unreliable output model. In this case, if we run the transformation blindly (without contracts), the Cheddar tool still runs analysis on the output model on which no shared protocols are defined for resources and gives thus non exploitable results w.r.t the input model: in a real-time system with shared resources, the definition of a shared protocol is necessary in order to ensure that the system does not exhibit undesirable phenomena such as priority inversion [6].

6 GENERALIZABILITY

In this section, we explain how our approach can be generalized to other transformations and customized according to the engineer needs w.r.t to both *prop1* and *prop2* of correctness (Section 1).

Other input/output models. Reusing our approach for other input/output models requires formalizing such models like we did for AADL and Cheddar (Section 3). Though it might seem disabling, such formalization is rather achievable with a reasonable cost. Indeed, first, as we have seen in Section 3, our formalization is done at some abstraction level that is the highest necessary for the transformation: e.g. we did not need to formalize AADL memories since they play no role in *AADL2Cheddar*. Second, the cost of formalization may be further reduced on a property-driven basis: e.g. we did not need to formalize the internal behavior of AADL threads because the "equivalence" between AADL and Cheddar models here (*prop2* of correctness) relates to entities (AADL threads, Cheddar tasks, etc.), their relations (AADL bindings, Cheddar links, etc.) and properties/parameters rather than low-level behavioral models.

Other transformations. Though presented for *AADL2Cheddar*, our formal environment is rather generic. For instance, if the number of any transformation rules is parametrized with the natural n, it is sufficient to replace the upper bound of $i \in 1 \ldots 9$ (and consequently the indices of v_9, e_9, γ_9 and R_9) with n to make the transformation and environment syntax and semantics (Section 3.4.2, Section 4.2.1, Section 4.2.2) fully generic. Therefore, besides defining their input model, output model, transformation rules and contracts according to their needs, engineers may reuse our formal environment without any further effort.

Other "levels" of prop2. Engineers may need to verify *prop2* of correctness at different "levels", e.g. one may content, in some context, with only syntactic correctness, whereas semantic correctness is necessary in a different context. This may be achievable following our approach without any change made to the underlying definitions of rules, contracts, and the formal environment. Indeed, since contracts guarantees can be any logical formulae (including quantifiers), they may be written to express most syntactic and semantic checks (and anything in between). For instance, our guarantee γ_5 (Section 4.1) may be viewed as something between syntactic and semantic equivalence, which may be replaced with a weaker or a stronger variant according to the level of *prop2* to achieve. This "flexibility" strengthens the rationale of using contracts, especially that in the case where only simple guarantees are needed, the cost of formalizing the input and output model may be drastically reduced (since such cost depends on the properties to verify, see §"*Other input/output models*" above). Yet, using our approach for heavy semantical checks (e.g. bisimulation) will need further reflection (Section 8).

Discarding prop1/prop2. Finally, if the engineer needs to verify only *prop1* or *prop2*, they may do so easily by simply encoding their assumptions (resp. guarantees) as tautologies. This aspect, stemming from the flexibility aspect described in the previous paragraph, gives the practitioner a full control over which aspect of correctness they are interested in.

7 RELATED WORK

There is a large corpus of works on model transformation verification (see e.g. reviews in [7, 25]). In this section, we focus on contract-based works. Contracts, having their roots in the Floyd-Hoare logic [12, 18], have been proposed to support the design and analysis of software in various domains [10, 22, 27, 28]. Due to their assumption/guarantee structure, contracts have naturally been integrated within transformation frameworks. In particular, Cariou et al. [8, 9] studied model transformation contracts written in OCL, where transformation operations are associated with contracts. Part of these contracts 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 conforms to a set of contracts with no need to execute the transformation. In a quite different scope, Guerra et al. [17] proposed 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. Then, the model transformation is 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). In general, besides applicability limitations (e.g. contracts in [8, 9] are only

applicable to endogenous model transformation, *i.e.* models with the same input and output metamodels), contract-based verification efforts presented above lack formal environments that define correctness properties and explain how they are verified.

The closest approach to ours is DSLTrans [3, 29], a transformation language featuring a formal contract-based prover. The spirit of DSLTrans is similar to that of our approach as contracts are verified in a formal way. However, whereas our approach relies on verifying the correctness of a transformation at runtime given an input model instance, DSLTrans is more generic as it targets proving correctness of a given transformation for any input model. Yet, since in DSLTrans the existence of a transformation (on which the prover runs) is assumed, such transformation (i) is not executed, which makes it impossible to verify *prop1* and (ii) must be encoded beforehand into DSLTrans (e.g. GM-2-AUTOSAR in [29]), which raises concerns on the correctness of such encoding.

8 CONCLUSION AND PERSPECTIVES

Reasoning on model transformations correctness in MDE is a challenging task. Formal methods may be employed to rigorously verify such correctness, but are out of software engineers expertise. In this paper, we proposed a formal, engineer-friendly and flexible approach to verify both *prop1* and *prop2* (Section 1) of transformation correctness. Flexibility refers to the possibility to tailor such verification to the engineer needs by e.g. increasing the level of *prop2* or discarding *prop1*. The "formal" aspect enables rigorous verification of correctness on a formal model. Finally, engineer friendliness is achieved through contracts, supported with a well-known programming language. Our approach is applied to the AADL to Cheddar model transformation and validated on a real case study.

We give two examples of future work directions. First, we will investigate extending our approach to other model-based frameworks, other application domains and other levels of correctness (Section 6). We believe that our approach may be suitable for e.g. component-based robotic frameworks (e.g. MAUVE [16]), where systematic mapping to verification frameworks through model transformation is of high importance. A major challenge will be to account for heavy yet necessary semantical guarantees knowing that the underlying robotic models feature rich behavioral models, hard real-time constraints and complex interaction mechanisms. Second, locating the source of errors (in case an assumption or a guarantee fails) based on errors raised by Ada may be inconvenient for large applications or transformations with a significant number of contracts. In future work, we will investigate implementing our framework in BIP [4]. Being both based on transition systems and able to embed blackbox functions, the BIP model will allow to find at which "state" the transformation failed without the need to worry about the Ada code or read the runtime errors.

REFERENCES

[1] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics*, 8(1):3–16, 2017.

[2] John Barnes. Rationale for Ada 2012: 1 Contracts and Aspects. *Ada User Journal*, 32(4):247, 2011.

[3] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *International Conference on Software Language Engineering*, pages 296–305. Springer, 2010.

[4] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE software*, 28(3):41–48, 2011.

[5] Bernard Berthomieu, Pierre-Olivier Ribet, and François Vernadat. The Tool TINA: Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[6] Björn B Brandenburg. Scheduling and Locking in Multiprocessor Real-Time Operating Systems. *PhD Thesis, University of North Carolina at Chapel Hill*, 2011.

[7] Daniel Calegari and Nora Szasz. Verification of Model Transformations: A Survey of the State-of-the-Art. *Electronic Notes in Theoretical Computer Science*, 292:5–25, 2013.

[8] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL Contracts for the Verification of Model Transformations. *Electronic Communications of the EASST*, 24, 2009.

[9] Eric Cariou, Raphël Marvie, Lionel Seinturier, and Laurence Duchien. Model Transformation Contracts and their Definition in UML and OCL. Technical Report 2004-08, LIFL, 2004.

[10] Patricia Derler, Edward A. Lee, Stavros Tripakis, and Martin Törngren. Cyber-Physical System Design Contracts. In *International Conference on Cyber-Physical Systems*, pages 109–118. ACM/IEEE, 2013.

[11] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison-Wesley, 2012.

[12] Robert W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[13] Mohammed Foughali. Formal Verification of the Functional Layer of Robotic and Autonomous Systems. *PhD Thesis, INSA Toulouse*, 2018.

[14] Mohammed Foughali, Silvano Dal Zilio, and Félix Ingrand. On the Semantics of the GenoM3 Framework. Technical Report 19036, LAAS-CNRS, 2019.

[15] Mohammed Foughali, Félix Ingrand, and Cristina Cerschi Seceleanu. Statistical Model Checking of Complex Robotic Systems. In *International Symposium on Model Checking of Software*, pages 114–134, 2019.

[16] Nicolas Gobillot, Charles Lesire, and David Doose. A Design and Analysis Methodology for Component-Based Real-Time Architectures of Autonomous Systems. *Journal of Intelligent & Robotic Systems*, 96:123–138, 2019.

[17] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated Verification of Model Transformations Based on Visual Contracts. *Automated Software Engineering*, 20(1):5–46, 2013.

[18] Charles A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[19] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Ocarina : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In *International Conference on Reliable Software Technologies (Ada-Europe)*, pages 237–250. Springer, 2009.

[20] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan MK Selim, Eugene Syriani, and Manuel Wimmer. Model Transformation Intents and Their Properties. *Software & systems modeling*, 15(3):647–684, 2016.

[21] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[22] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[23] Anantha Narayanan and Gabor Karsai. Specifying the Correctness Properties of Model Transformations. In *International Workshop on Graph and Model Transformations*, pages 45–52, 2008.

[24] Anantha Narayanan and Gabor Karsai. Verifying Model Transformations by Structural Correspondence. *Electronic Communications of the EASST*, 10, 2008.

[25] Lukman Ab Rahim and Jon Whittle. A Survey of Approaches for Verifying Model Transformations. *Software & Systems Modeling*, 14(2):1003–1028, 2015.

[26] Christophe Reymann, Mohammed Foughali, and Simon Lacroix. Repeatable Decentralized Simulations for Cyber-Physical Systems. In *International Conference on Software Quality, Reliability and Security (QRS)*, pages 240–247. IEEE, 2019.

[27] Ivan Ruchkin, Dionisio De Niz, Sagar Chaki, and David Garlan. Contract-Based Integration of Cyber-Physical Analyses. In *International Conference on Embedded Software (EMSOFT)*, page 23. ACM, 2014.

[28] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control*, 18(3):217–238, 2012.

[29] Gehan MK Selim, Levi Lúcio, James R Cordy, Juergen Dingel, and Bentley J Oakes. Specification and Verification of Graph-Based Model Transformation Properties. In *International Conference on Graph Transformation*, pages 113–129, 2014.

[30] Shane Sendall and Wojtek Kozaczynski. Model Transformation: the Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[31] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a Flexible Real Time Scheduling Framework. In *SIGAda International Conference (SIGAda)*, pages 1–8, 2004.