A proof-based ground algebraic meta-model for reasoning on ASTD in Event-B

Christophe Chen, Peter Riviere, Neeraj Kumar Singh, Guillaume Dupont, Yamine Ait Ameur, Marc Frappier

A proof-based ground algebraic meta-model for reasoning on ASTD in Event-B

Christophe Chen[1,2], Peter Rivière[1,3], Neeraj Kumar Singh[1], Guillaume Dupont[1], Yamine Ait Ameur[1], Marc Frappier[2]

[1]*IRIT/Toulouse INP-ENSEEIHT, CNRS, University of Toulouse, France*
[2]*Université de Sherbrooke, Sherbrooke, QC, Canada*
[3]*JAIST - Japan Advanced Institute of Science and Technology, Nomi, Ishikawa, Japan*
{christophe.chen, peter.riviere, nsingh, guillaume.dupont, yamine}@enseeiht.fr,
marc.frappier@usherbrooke.ca

***Abstract*—Algebraic State Transition Diagram (ASTD) is a formal, graphical, state-based modeling language for the design of complex critical systems. It offers a set of process algebra operators to compose hierarchical state machines, streamlining modularity in system design. Despite advances in incorporating features such as local state variables and real time, ASTD tool support has primarily focused on design and testing. Recently, proof obligations for state invariant preservation were introduced, but the validity of these proof obligations was informally justified. To address this issue, this paper introduces an Event-B-based algebraic data-type, called EB[ASTD], formalising the operational semantics of ASTDs. Relying on a deep modelling strategy, this framework defines a ground model for the formal operational semantics of ASTDs and provides a proof-based mechanism allowing for reasoning on specific ASTDs defined as instances of this meta-model. It enables explicit manipulation of ASTD concepts and improves formal reasoning by integrating their syntax and semantics as Event-B algebraic theories. The designed tool relies on the Rodin platform to provide automated proof obligation generation, automatic and interactive verification, graphical animation, and model checking of ASTDs. In addition, it allows us to prove the validity of the generated ASTD proof obligations for state invariants, which reveals the core properties and theorems that ensure its reliability. Overall, EB[ASTD] provides a sound foundation for proving properties about ASTDs and operates effectively within the Rodin platform, designed for managing ASTD models and proofs.**

I. Introduction

Context. Algebraic State Transition Diagram (ASTD) [1] is a formal, graphical, state-based modeling language for the design of complex systems [2], [3]. It offers a set of process algebra operators, drawn from CSP, to compose hierarchical state machines, streamlining modularity in system design. Its operational semantics [1] specifies transition rules for each ASTD operator. The expressiveness of ASTD is defined by an extensive range of operators, such as *Automaton, Sequence, Guard, Closure, Choice, QChoice, Synchronisation, QSynchronisation, Flow, and QFlow*, which facilitates the design of complex systems using the composition operators.

Motivation. Over time, ASTD has been extended with various perspectives. Despite advances in incorporating features such as local variables [4], time [5] and state invariants [6], ASTD tools have primarily focused on design and testing, with no tool support for formal proof of properties. These developments have been conducted empirically relying on ad hoc model transformations and testing. Indeed, the attempts for mechanising the semantics of ASTDs and its associated verification process involved shallow embeddings of ASTDs, i.e., interpreting ASTD specifications in another language being either a formal modelling language like B and Event-B machines [7], [8] or a programming language like C++ [4], [9]. Reasoning on ASTDs is performed on the target modelling or programming languages, using their own semantics, verification processes and tools. As a consequence, the exploitation of the expressive power, hierarchical and compositional structure of ASTDs are lost after being embedded in the target language. The identified specification errors are expressed in terms of the target modeling or programming language; tracing them back to the original ASTD is difficult. This lack of traceability complicates validation and decreases confidence in the overall system. To the best of our knowledge, there is no convincing mechanism for representing ASTD semantics and formalising them in order to do formal reasoning on ASTD models, including defined properties.

Our contribution. To address this issue, we introduce the EB[ASTD] framework, that formalises the ASTD language and its semantics, including state-transitions systems and CSP composition operators. We use Event-B [10] and its extension for defining algebraic theories [11] to express a meta-model. Relying on a deep embedding, an ASTD specification is formalised as instances of algebraic data-types representing the various ASTD operators. This meta-model formalises the semantics of ASTDs, their properties (e.g., invariants) as well as the proof system to discharge such properties.

It is worth noting that, unlike other defined embeddings, the proposed deep embedding preserves the structure of the ASTD descriptions, making it possible to trace back identified errors on the ASTD itself. Moreover, we highlight that additional proof obligations can be added to the EB[ASTD] framework by extending the core meta-model. The consistency of the approach is formally established. Lastly, an example that

Part of this work is supported by the ANR project EBRP:EventB-Rodin-Plus under grant no. ANR-19-CE25-0010 and by NSERC.

illustrates the application of our proposed framework through meta-model instantiation is presented.

ASTD Mechanisation. The entire EB[ASTD] framework is supported by the Rodin platform [12], which facilitates the handling of Event-B models and proofs. The Rodin platform has been set up to provide tool support, to facilitate formal verification of ASTDs, and to automatically generate the corresponding ASTD proof obligations (POs). Furthermore, the ProB model checker [13] associated to Rodin facilitates both the animation and model checking of ASTD models, enhancing the validation process. Additionally, we use VisB [14] to provide a graphical representation of ASTD models, which aids in animation and further improves the overall validation process. All these features define the components of a complete framework allowing to formally engineer ASTD models.

Organisation of this paper. Section II summarises the various developments used to study ASTD models, including the use of proof assistants and embedding mechanisms. Section III provides an overview of the features of ASTD and Event-B used in this paper. Sections IV and V present respectively the EB[ASTD] framework we designed and show how to use it to develop specific ASTD models. The definition and generation of ASTD proof obligation is addressed in Section VI. Finally, conclusion is presented along with future work in Section VII.

II. Related work

There were several attempts to handle ASTD for validation and verification purposes. In general, each of them manage to offer partial solutions but come with significant limitation.

eASTD and cASTD [15], [16]: eASTD is a graphical editor for ASTD specifications. cASTD is a compiler that generates C++ code implementing an ASTD specification. Generated programs consider events from the environment and execute them on the ASTD specification. ASTD actions are triggered by events; they are written in C++ and they can use any C++ libraries to execute code necessary in a given application context, making ASTD very effective for constructing industrial-strength control systems. The generated code can be used as an efficient implementation of the specification, with excellent response times. cASTD can also be used to execute basic scenarios to validate a specification. However, it does not offer the same rich animation features provided by ProB [13] to explore various scenarios, like path search that leads to a desired state, show events enabled in a given state, backtrack the execution to explore a different execution branch, or support of graphical animation of specification.

Translation of ASTD to Event-B [17]. This approach defines a model-to-text transformation of ASTD into Event-B by associating each composition operator to a pattern in Event-B. This transformation relies on control variables to handle ASTD operators. Since all variables in Event-B are global, the translation results in a monolithic and flat Event-B model where state invariants are hard to prove. Part of this translation was verified in Coq [18], using a simulation relation.

pASTD [6]: pASTD enables the definition of invariants on states on any ASTD. Thus both global invariants and local invariants can be defined. pASTD generates proof obligations in the form of theorems in an Event-B context; it is implemented as a Java plugin in Rodin. Then, Event-B provers are used to discharge the proof obligations associated to the generated context. Here, modifications of an ASTD are not reflected back in the Event-B context initially generated; it must be re-generated to obtain the new proof obligations, and proofs completed on the initial version of the context are lost and must be redone. The rules used by pASTD to generate the proof obligations were informally justified. The correctness proof of its Java code would be a tremendous effort.

The limitations of these approaches are mostly due to their pragmatic conceptualisation. Each technology provides a custom tool support for ASTD, which can introduce new errors and makes it difficult to integrate several tools on the same model. Moreover, the transformations are validated empirically and verification is a posteriori i.e. on the result of the transformation leading to a loss of the rich expressivity provided by ASTDs. In addition, as they manipulate ASTD instances, these approaches lack formal reasoning on the ASTD modelling language itslef (no meta-model).

Actual deep embedding, promoted in our work, of modelling languages into another is an effective technique used to perform reasoning and model analysis. It helps to ensure the foundational soundness of both core syntax and semantics of the embedded languages. By doing so, system designers can leverage the benefits of formal specifications and reasoning mechanisms essential for designing complex systems. Here are several efforts in this direction adopted in meta-modelling and programming [19], DSL [20], reflexion [21], using PVS [22], Coq [23], Isabelle/HOL [24] and Event-B [10].

For instance, in [25], the authors introduced a framework within the MetaCoq project to specify and encode the semantics of Coq, facilitating the development of a certified meta-programming environment. This framework was successfully implemented in the development of CertiCoq [26], a certified compiler for Coq. Similarly, Isabelle/HOL [27] was used to define HOL models and the reasoning mechanisms necessary for describing complex systems with self-replacement functionality. In [28], a construction and formal proofs of equivalence between the operational and denotational failure-divergence semantics in Isabelle/HOL-CSP [29], using a shallow embedding, has been proposed. For the B method [30], the authors of [31] presented a PVS formalisation addressing the key concepts and core functionalities of B operators. Additionnally, the context formalisation proposed in [32] aims to provide explicitly defined theorems used to express advanced reasoning capabilities on Event-B models. The embedding approach is also used in [33] to formalise the core semantics of Event-B and its refinement in CSP [34], [35]. In [36] the EB4EB reflexive framework uses Event-B to model Event-B. The semantics of Event-B is expressed as a machine with events that express state transformations. It enables the modelling of generic properties of Event-B models and sufficient conditions to prove them. This framework is extended in [37] to handle temporal properties and associated

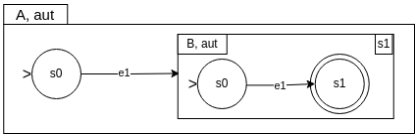

Figure 1: A simple Automaton

reasoning. We rely on this approach for defining a meta-model integrating both syntax and semantics of ASTDs as Event-B algebraic theories.

In the same spirit of [36], [37], the proposed framework *invites* the semantics of ASTDs in Event-B with its well-established tool support Rodin. Event-B and its proof mechanism are reused for free. No additional tool or code has to be written for proof obligation generation as it is the case for pASTD. More importantly, EB[ASTD] allows us to formally prove the correctness of pASTD proof obligation rules. The ProB model checker has been applied for ASTD animation remaining at a higher abstraction level, contrary to cASTD.

III. Background

In this section, we provide an overview of ASTD and Event-B that is required for understanding the meta-modeling and reasoning concepts proposed in the rest of the paper.

A. Algebraic State Transition Diagram (ASTD)

Algebraic State Transition Diagrams (ASTD) [1] is a graphical, state-based formalism designed to overcome the limitations and lacks of languages such as state charts and UML activity diagrams. They rely on automata-like diagrams that may be composed using various operators borrowed from CSP, and whose states may include ASTDs themselves, making it possible to create complex hierarchies.

1) Automata and composition operators: **Automaton ASTD.** An automaton ASTD $(aut, \Sigma, Q, \delta, F, q_0, \nu)$ is similar to a traditional automaton, with the distinction that a state can be complex and can itself include an ASTD, similar to hierarchical states in Statecharts. In this structure, Σ denotes the set of event labels (alphabet) of the automaton, while Q represents the set of states. $F \subseteq Q$ is the set of final states. $q_0 \in Q$ is the initial state. $\nu \in Q \rightarrow ASTD$ maps each automaton state to a nested ASTD, with the ASTD type $Elementary$ being used when the state is atomic (i.e., not hierarchical).

For a given ASTD, the transition relation is defined by $\delta \subseteq \langle \eta, \sigma, \phi, final? \rangle$, where η denotes a transition arrow $\langle n_1, n_2 \rangle$, with n_1, n_2 respectively denoting the source and destination states, $\sigma \in \Sigma$ is an event, ϕ is a guard (predicate) for the transition and $final?$ is a Boolean to indicate that the source state n_1 must be final in order to trigger the transition.

Example. Fig 1 depicts an example of an Automaton A where the states $s0$ and $s1$ of ASTD A are respectively mapped to an elementary ASTD $(A.\nu(s0) = Elementary)$ and another automaton B $(A.\nu(s_1) = B)$.

Sequence ASTD $(\Rightarrow)$**.** A sequence ASTD, denoted with $\langle \Rightarrow, A_1, A_2 \rangle$, represents the sequential composition of two ASTDs. The composed ASTD starts execution in ASTD A_1; when A_1 has reached its final state, ASTD A_2 execution starts.

Closure ASTD $(\star)$**.** Closure ASTD $\langle \star, A_1 \rangle$ allows ASTD A_1 to be executed an arbitrary number of times. A_1 can restart from its initial state when it reaches a final state.

Guard ASTD $(\Rightarrow)$**.** Guard ASTD $\langle \Rightarrow, g, A_1 \rangle$ checks that the guard g is satisfied before executing the first transition of A_1. Note: g is not checked for the subsequent transitions of A_1.

2) ASTD states: The transition relation of the operational semantics is defined on the notion of ASTD state.

An Automaton state is a triplet $\langle aut_\circ, q, s \rangle$ where $aut_\circ$ is the constructor for automaton state, $q \in Q$ is the state name and s the state of sub-ASTD $\nu(q)$. For simplicity we note S_a as the set of all ASTD states of a. Hence, $s \in S_{a.\nu(q)}$.

A Sequence state is represented by $(\Rightarrow_\circ, i, s)$, where $\Rightarrow_\circ$ is the constructor for sequence state, $i \in \{1, 2\}$ is the index of the running sub-ASTD, and $s \in S_{A_i}$ represents the ASTD state of the running sub-ASTD.

A Closure state is represented as $\langle \star_\circ, i, s \rangle$, where $\star_\circ$ is the constructor for closure state, $i \in \{0, 1\}$ indicates whether the closure has started $(i = 1)$ or not $(i = 0)$, and $s \in S_{A1}$ denotes the current state of the sub-ASTD A_1.

A Guard state is defined as $\langle \Rightarrow_\circ, i, s \rangle$, where $\Rightarrow_\circ$ is the constructor for guard state, $i \in \{0, 1\}$ precises if the guard is evaluated, and $s \in S_{A_1}$ is the state of the sub-ASTD A_1.

The other ASTD operators are omitted for brevity; the reader may refer to [38], [39] for more details.

Example. Fig 2 shows an ASTD drawn from [6] serving as an illustrative example throughout the paper. ASTD `A` is a closure of ASTD `B`, where ASTD `B` consists of a sequence that combines ASTD `C` and ASTD `E`. ASTD `C` is a closure of ASTD `D`, while ASTD `E` is a guard for ASTD `F`. Industrial scale ASTD examples can be found in [2], [3], [40].

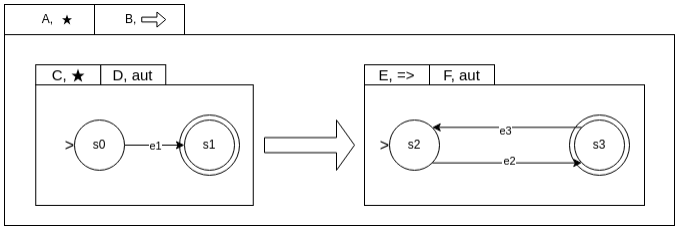

Figure 2: Illustrative example

3) Extending ASTD with state variables: Each non elementary ASTD is extended with the attributes $\langle V, A_{init}, Inv \rangle$ where V is a set of local variables with read/write access for its sub ASTDs, A_{init} is the initialisation action for its local variables V and Inv is the local invariant about local variables and variables from enclosing ASTDs above in the hierarchy. Elementary ASTD can only have invariant. State variables are also called *attributes*.

State Variable update. In order to allow variable update during execution, the automata transition relation $\delta \subseteq \langle \eta, \sigma, A_{tr}, \phi, final? \rangle$ is also extended to trigger an action A_{tr} upon accepting an event σ from the environment. The

ASTD Type	ASTD State	Description
(elem, Inv)	$\text{elem}_\circ$	
$(\text{aut}, \Sigma, Q, \delta, F, q_0, \nu, V, .., Inv)$	$(\text{aut}_\circ, v, q, s)$	$q \in Q, s \in S_{a.\nu(q)}$
$(\diamond\!\!\rightarrow, A_1, A_2, V, A_{init}, Inv)$	$(\diamond\!\!\rightarrow_\circ, v, i, s)$	$i \in 1..2 \wedge s \in S_{A_i}$
$(\star, A_1, V, A_{init}, Inv)$	$(\star_\circ, v, i, s)$	$i \in 0..1 \wedge s \in S_{A_1}$
$(\Rightarrow, g, A_1, V, A_{init}, Inv)$	$(\Rightarrow_\circ, v, s)$	$s \in S_{A_1}$

Table I: ASTD syntax and ASTD state

ASTD	Attributes	Initialisation	Invariant
A	x_A	$x_A \coloneqq 0$	$x_A \geq 0$
B	x_B	$x_B \coloneqq x_A + 1$	$x_B > 0$
C	x_C	$x_C \coloneqq 0$	$x_B > x_A$
D	x_D	$x_D \coloneqq x_C + 1$	$x_D \geq 0$
E	x_E	$x_E \coloneqq x_A$	$x_E \geq 0 \wedge x_E < x_B \wedge x_E \leq x_A$
F	x_F	$x_F \coloneqq 0$	$x_F \geq 0$
S0			$x_D > x_C \wedge x_C \geq 0$
S1			$x_C \geq x_D \wedge x_A > 0$
S2			$x_F = 0 \vee x_A > x_E$
S3			$x_F > 0 \wedge x_A > 2 * x_E$

Table II: Attributes, Initialisations, and Invariants for ASTDs

initialisation action is triggered when the ASTD starts its execution. Actions are abstracted as a relation between the before and after values of the variables; the description of an action language is omitted. In general, the initialisation is done top-down, while transition actions are executed bottom-up.

Semantics of State Variables. In order to record the evolution of variables, ASTD states are also extended to take into account the valuation of variables. Each non elementary ASTD state is associated to a field $v \in Var \nrightarrow T$ that records the current value of local variables. A summary of ASTD syntax and ASTD states is given in Table I.

The illustrative example of Fig 2 is extended with variables, actions and invariants (see Tables II and III). Note that the guard for ASTD E is $x_B > x_A + 4$, and the guard for event e_3 is $x_A < 10000$. The initial state for the same example is given in Table IV. For simplicity, all variables are integers.

B. Event-B

Event-B [10] is a state-based formal method based on set theory and first-order logic (FOL) for developing complex systems using a *correct-by-construction* approach based on refinement. State transitions are triggered by events. Event-B has been extended with several features, like the ability to define new types using an algebraic approach. Figure V shows an overview of the global architecture of the modelling framework.

Event	Action
e_1	$x_C \coloneqq x_C + x_D;\ x_B \coloneqq x_B + x_C;\ x_A \coloneqq x_A + 1$
e_2	$x_A \coloneqq x_A + x_B;\ x_F \coloneqq x_F + 1$
e_3	$x_A \coloneqq x_A - x_E$

Table III: Events and their corresponding actions

$$(\star_\circ, \{xA \mapsto 0\}, 0,\ (\diamond\!\!\rightarrow_\circ, \{xB \mapsto 1\}, 1,\ (\star_\circ, \{xC \mapsto 0\}, 0,\ (aut_\circ, \{xD \mapsto 1\},\ (s_0 \mapsto elem_\circ)))))$$

Table IV: Initial state of ASTD A of Fig. 2

Context	Machine	Theory
CONTEXT Ctx **SETS** s **CONSTANTS** c **AXIOMS** A **THEOREMS** T_{ctx} **END**	**MACHINE** M **SEES** Ctx **VARIABLES** x **INVARIANTS** $I(x)$ **THEOREMS** $T_{mch}(x)$ **VARIANT** $V(x)$ **EVENTS** **EVENT** evt **ANY** α **WHERE** $G_i(x, \alpha)$ **THEN** $x :\mid BAP(\alpha, x, x')$ **END** ... **END**	**THEORY** Th **IMPORT** Th1, ... **TYPE PARAMETERS** E, F, ... **DATA TYPES** **Type1**(E, ...) **constructors** **cstr1**(p_1: T_1, ...) **OPERATORS** **Op1** <nature> (p_1: T_1, ...) **well–definedness** $WD(p_1, \ldots)$ **direct definition** D_1 **AXIOMATIC DEFINITIONS** **TYPES** A_1, ... **OPERATORS** **AOp2** <nature> (p_1: T_1, ...): T_r **well–definedness** $WD(p_1, \ldots)$ **AXIOMS** A_1, ... **THEOREMS** T_1, ... **PROOF RULES** R_1, ... **END**
(a)	(b)	(c)

Table V: Structure of Event-B Contexts, Machines and Theories

(1.1)	Theorems (THM)	$A \Rightarrow T_{ctx}$
(1.2)	Theorems (THM)	$A \wedge I^A(x^A) \Rightarrow T_{mch}(x^A)$
(2)	Initialisation (INIT)	$A \wedge AP^A(\alpha^A, x^{A\prime}) \Rightarrow I^A(x^{A\prime})$
(3)	Invariant preservation (INV)	$A \wedge I_A(x^A) \wedge G_A(x^A, \alpha^A) \wedge BAP^A(x^A, \alpha^A, x^{A\prime}) \Rightarrow I^A(x^{A\prime})$
(4)	Event feasibility (FIS)	$A \wedge I_A(x^A) \wedge G^A(x^A, \alpha^A) \Rightarrow \exists x^{A\prime} \cdot BAP^A(x^A, \alpha^A, x^{A\prime})$
(5)	Variant progress (VAR)	$A \wedge I^A(x^A) \wedge G^A(x^A, \alpha^A) \wedge BAP^A(x^A, \alpha^A, x^{A\prime}) \Rightarrow\ V(x^{A\prime}) < V(x^A)$

Table VI: Machine Proof obligations

1) Event-B Contexts and Machines: Table V(a) shows the syntax of Event-B `Context`, which defines the static elements of a model using elementary components such as sets s, constants c, *axioms* A. Table V(b) shows the syntax of Event-B `Machine`, which represents the dynamic behaviour using *variables* x, *invariants* $I(x)$, theorems T_{mch}, *variants* $V(x)$ and *events* evt. Events use a Before-After Predicates (BAP) to modify declared variables. Defined invariants and theorems ensure safety properties, while defined variants are useful to express convergence of events.

2) Refinement: is a powerful Event-B feature, not used in this paper, allowing machines to be built gradually at various abstraction levels while maintaining proved properties.

3) Proof Obligations (POs): Table VI outlines proof obligations linked to Event-B models. They are generated automatically by Rodin and must be discharged to ensure the correctness of the model. In this paper, the `THM` proof obligation associated to contexts is extensively used.

4) Event-B extension with Theories: The expressiveness of Event-B, which is based on set theory and first-order logic (FOL), is adequate for modelling, reasoning, and simulating system behaviours. However, it lacks to define new types, operators, and proof mechanisms required for designing complex systems and scaling them for future system development. To address this limitation, the *Theory Plug-in* has been introduced [11]. Table V(c) depicts the overall structure and modelling components for developing new theories. This fea-

ture enables the development of custom data-types, operators, definitions, and axioms. Additionally, theories facilitate polymorphic types, well-defined (WD) conditions for operators, and reusable theorems. WD conditions act as preconditions to ensure that the partially defined operators of a theory are used correctly. WD proof obligations are generated when an operator is applied. Multiple theories, including `Real`, `DiffEq` [41], `Ontology` [42],..., have been developed to address various aspects of modelling concepts.

5) Rodin IDE: Rodin [12] is an open-source Eclipse-based Integrated Development Environment designed for developing Event-B models and theories. It offers a range of features including project management, model editing, model animation, refinement, proof management, and code generation. ProB [13] and VisB [14] can be used within the Rodin IDE for model checking and animating the developed specifications, allowing to identify potential errors and deadlock checking. Additionally, Rodin is equipped with advanced automated proving mechanisms, such as SMT solvers and predicate provers, which support both automated and interactive proof reasoning. The theory plugins enable users to develop complex theories that enhance Event-B modeling concepts. These developed theories are seamlessly integrated into the modeling and proof processes, making them available for use during system development.

IV. The EB[ASTD] Framework

The EB[ASTD] framework is grounded on Event-B and its algebraic theories that allows reasoning on them. It is inspired from the reflexive meta-modelling EB4EB framework [36] defined for Event-B. The complete models are available at https://www.irit.fr/EBRP/software/.

Set-based notation. In order to handle predicates as first-class citizens in Event-B, we extensively use the *axiom of comprehension* together with set-theoretical notations. Typically, a predicate $P(x)$ is modelled as $\widetilde{P} = \{x \mid P(x)\}$, and the truth value of P on a given x is represented as $x \in \widetilde{P}$. In addition, set-theoretical operators (e.g., $\subseteq$, $\cap$) are used to encode more conveniently predicate connectors (e.g., respectively, $\Rightarrow$, $\wedge$).

A. Architecture of EB[ASTD]

As illustrated in Fig. 3, EB[ASTD] is based on a set of algebraic data-types that reuse basic theories for natural numbers and arrays. The theories for the core concepts of ASTD are represented in the *ASTD Core* section of Fig. 3. The `ASTDStruct` algebraic theory (presented in Sections IV-B and IV-C) introduces the syntactic constructs of ASTDs and their static semantics properties. A trace-based operational semantics is described by the `ASTDBehaviour` theory (see Section IV-D). The proof obligations together with their automatic generation are formalised in the `ASTDPO` Event-B theory (detailed in Section VI-C). All of these theories together define a denotational semantics for ASTD.

The *ASTD PO Correctness* area of Fig. 3 depicts the framework's architecture for consistency and soundness. The `ASTDCorrectness` theory (Section VI-B) ensures the consistency of these proof obligations using the trace-based semantics described in the `ASTDTraces` theory.

Last, the *Basic Theory* of Fig. 3 are the basic mathematical building blocks grounding the whole framework (Peano, associative arrays). They are not discussed in this paper.

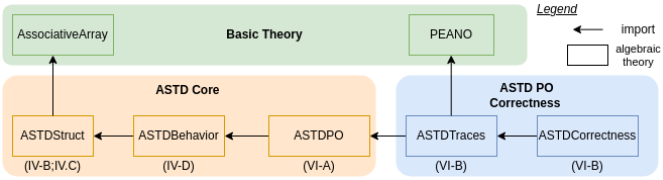

Figure 3: Architecture of the meta-theory

B. Data types and constructors for ASTDs

Listing 1 presents the `ASTDStruct` Event-B theory. It is central to the framework as it describes ASTD types and ASTD states in a denotational style. It introduces three polymorphic parameters: `St` for state names, `Ev` for events (transition labels) and `Var` for state variables.

Note. For every development presented in this paper, variables are associated to the hierarchical level (number) of decomposition of the ASTD, using the mapping $Var \nrightarrow \mathbb{Z}$.

```
THEORY ASTDStruct
TYPE PARAMETERS St , Ev , Var
DATA TYPES
ASTD( St , Ev , Var )
constructors
  Elementary ( ElemInv : ℙ(ℙ( Var × ℤ)))          // invariant
  Automaton (                                      //(aut, Σ, Q, δ, F, q0, ν)
    InitialState : St ,                            // q0
    FinalState : ℙ(St) ,                           // F
    States : ℙ(St) ,                               // Q
    Event : ℙ(Ev) ,                                // Σ
    Transitions : ℙ(Ev × (St × St)) ,              // δ
    ...,
    AutAttr : ℙ( Var) ,                            // V
    AutInitAttr : ℙ( Var × ℤ) ↔ ℙ( Var × ℤ) ,      // A_init
    AutInv : ℙ(ℙ( Var × ℤ)) ,                      // invariant
    Mapping : ℙ(St × ASTD(St, Ev, Var))            // ν
    )
  Sequence (                                       // ⟨⇨, A1, A2⟩
    SeqFirst : ASTD(St, Ev, Var) ,
    SeqSnd : ASTD(St, Ev, Var) ,
    SeqAttr : ℙ( Var) ,
    SeqInitAttr : ℙ( Var × ℤ) ↔ ℙ( Var × ℤ) ,
    SeqInv : ℙ(ℙ( Var × ℤ))
    )
  Closure ( ... )
  Guard ( ... )

ASTDState ( St , Var )
constructors
  elem∘ ( )
  aut∘ (
    AutCache : ℙ( Var × ℤ) , // store the values of variables
    StateName : St ,
    AutSubState : ASTDState(St, Var)
  )
  seq∘ (
    SeqCache : ℙ( Var × ℤ) , // store the values of variables
    Pos : ℤ , // index of current ASTD
    SeqSubState : ASTDState(St, Var)
  )
  cls∘ ( ... )
  grd∘ ( ... )
```

Listing 1: Data Type for ASTD

ASTD structure and states are formalised with the two data-types, `ASTD(St, Ev, Var)` and `ASTDState(St, Var)`, that make use of the type parameters of the theory. The `ASTD(St, Ev, Var)` data-type is equipped with five constructors:

- `Elementary` represents a basic state with an invariant.
- `Automaton` models a state transition system with all the components defining an ASTD automaton as defined in Section III-A1 (states, transitions, invariants, etc.). Transitions are themselves associated to guards and actions with before and after state variables relations $\mathbb{P}(Var \times \mathbb{Z}) \leftrightarrow \mathbb{P}(Var \times \mathbb{Z})$. To model ASTD hierarchy, $Mapping$ introduces the decomposition of a given automaton state into another ASTD.
- `Sequence`, `Closure` and `Guard` operators define ASTD composition operators, following the definitions of Section III-A1.

The `ASTDState(St, Var)` data-type defines the complex notion of state in relation to the ASTD construct. It is composed of:

- $elem_\circ$, represents an elementary (non-hierarchical) state.
- $aut_\circ$, represents the state of an automaton, consisting of the values of state variables `AutCache` of that automaton, a state label `StateName` and potential sub-states in the case of hierarchical ASTD states.
- $seq_\circ$, $cls_\circ$ and $grd_\circ$ define the state resulting from composed ASTDs. For example, in the case of `Sequence`, the state is described by the current state variable value `SeqCache`, the current position `Pos` in the sequence (first or second ASTD) and its sub-state.

C. Well-defined ASTDs

The data-types previously defined, as well as constructors and destructors, only contain typing information, which may result in ill-defined instances. The second part of the `ASTDStruct` theory provides necessary extra conditions, in the form of predicates, for consistently defining ASTD.Such conditions relate to the static semantics of ASTD. They are formalised in the `ASTDStruct` theory through predicate operators. Then, these predicates are used as well-definedness (WD) conditions and thus generate associated proof obligations each time an ASTD or an ASTD state is manipulated.

This part describes, in Listings 2, 3 and 4 a set of well-constructed operators to enhance the static semantics of ASTDs. They are derived from the core ASTD modelling language as well as from necessary hypotheses needed to establish the well-definedness (WD) of the defined operators.

Listing 2 presents a list of operators with WD conditions related to the actions associated to the transitions.

```
InitAction_WellCons predicate (
    attr : ℙ(Var),
    act : ℙ(Var × ℤ) ↔ ℙ(Var × ℤ))
  direct definition
    (∀env · env ∈ dom(act) ⇒
      (env ∪ (attr ◁ act(env)) = act(env)))

InitActionS_WellCons predicate (a : ASTD(St, Ev, Var))
    well-definedness condition ASTD_WellTyped(a)
  recursive definition
  case a:
    Elementary(inv) => ⊤
    Automaton(i, ..., attr, initAttr, inv, mapping) =>
      InitAction_WellCons(attr, initAttr)
      ∧(∀s · s ∈ allstate ⇒
        InitActionS_WellCons(mapping(s)))
    Sequence(fst, snd, attr, initAttr, inv) =>
      InitAction_WellCons(attr, initAttr)
      ∧InitActionS_WellCons(fst)
      ∧InitActionS_WellCons(snd)
    Closure(...) => ...
    Guard(...) => ...
```

Listing 2: Operator InitActionS_WellCons

The `InitAction_WellCons` operator, is declared with two arguments: *attr*, a set of state variables, and an action *act* which asserts that the action *act* modifies only the variables in the scope of the action ($\lhd$ is a domain restriction operator).

In the same listing, the predicate operator `InitActionS_WellCons` recursively applies predicate `InitAction_WellCons` on each type of ASTD (elementary, automaton, and the composition operators). This definition is inductive and entails inductive case-based reasoning when the proof obligations need to be discharged.

This kind of property definition and its replication on the ASTD structure is extensively used in our models.

```
Automaton_WellCons predicate (astd : ASTD(St, Ev, Var))
  recursive definition ...

ASTD_WellTyped predicate (a : ASTD(St, Ev, Var))
  well-definedness condition ...
  recursive definition ...

Scope_WellCons predicate (astd : ASTD(St, Ev, Var),
    accVar : ℙ(Var)) //accVar is empty set for root ASTD
      well-definedness condition ...
      recursive definition

InvariantS_WellCons predicate (astd : ASTD(St, Ev, Var),
    accVar : ℙ(Var)) //accVar is empty set for root ASTD
  well-definedness condition ....
  recursive definition ..
```

Listing 3: Other WellCons rules

Additional predicate operators formalise relevant properties of ASTDs (Listing 3), ensuring that the structure of an automaton is well defined (`Automaton_WellCons`), the typing constraints are fulfilled (`ASTD_WellTyped`), state variables of an ASTD hierarchy are not already declared in its parent ASTDs (`Scope_WellCons`) and the local invariants of an ASTD are defined correctly on the variables in its scope (`InvariantS_WellCons`).

```
ASTD_WellCons predicate
(a : ASTD(St, Ev, Var), accVar : ℙ(Var))
  direct definition
    Automaton_WellCons(a)
    ∧ASTD_WellTyped(a)
    ∧Scope_WellCons(a, accVar)
    ∧InitActionS_WellCons(a)
    ∧InvariantS_WellCons(a, accVar)
```

Listing 4: Definition of operator ASTD well-constructed

Last, in Listing 4, all of the well-constructed properties defining static semantics constraints are wrapped in a single well-defined operator `ASTD_WellCons`. This operator

includes all of the well construction rules, representing the necessary well-defined conditions associated with an ASTD.

The `ASTD_WellCons` operator is a conjunctive predicate. When used in an Event-B model, it generates all of the well-definedness proof obligations of *all* the operators that are involved in its definition. Then, its proof uses the discharged well-definedness conditions as hypotheses and its first applied proof rule is the $\wedge$-elimination rule. When discharged, the associated WD POs are used as hypotheses for all remaining proofs. In particular, their definition proved sufficient to establish the consistency of the defined semantics (see Section VI).

D. Operational semantics for ASTD

The next step consists in describing the behavioural semantics of ASTDs as they define state transitions systems. An operational semantics is described as a set of rules in the `ASTDBehaviour` Event-B theory as shown in Listings 5 and 6. Following an ASTD structure-based definition, two categories of rules are inductively defined for each ASTD construct: one for the initialisation provided in Listing 5 and the second for the induction step in Listing 6.

The algebraic theory `ASTDBehaviour` imports `ASTDStruct` describing the structure of ASTDs and similarly, it uses three type parameters `St`, `Ev`, and `Var`.

1) The initialisation rule (Listing 5): is an expressionwhere its *astd* parameter is well defined. It returns the basic ASTD state for `elementary`, builds, from the initial values of the state variables, an ASTD state from an automaton $aut_\circ$, the first ASTD state of a sequence, etc. The same definition is provided for the other composition operators.

Note that in the case of operators other than *elementary*, their definition collects recursively the initial states of the potential ASTD hierarchy. This recursion stops when an `elementary` ASTD is reached.

```
THEORY ASTDBehaviour
IMPORT THEORY ASTDStruct
TYPE PARAMETERS St, Ev, Var
OPERATORS
    Init expression
    (astd : ASTD(St, Ev, Var), env : Var ↠ ℤ)
      well-definedness condition
      Automaton_WellCons(astd), ...
      recursive definition
        case astd:
          Elementary(...) =>
            elem∘ : ASTDState(St, Var)
          Automaton(i, ..., attr, initAttr, inv, mapping) =>
            aut∘(
                attr ◁ initAttr(env), // initialize local attr
                i,                    // the initial state name
                Init(mapping(i), initAttr(env)) // recursive call
            )
          Sequence(fst, snd, attr, initAttr, inv) =>
            seq∘(
              attr ◁ initAttr(env),
              1,                      // position fst
              Init(fst, initAttr(env))
            )
          Closure(..) => ...
          Guard(...) => ...
```

Listing 5: Operator Init from ASTD

2) The next state (progress) rule (Listing 6): It is defined case by case, similarly to the initialisation. The `NextState` operator of Listing 6, parametrised by an ASTD automaton *astd*, event σ, current state *curr*, and environment E_e, returns a pair of elements. The first element is the potential next state derived from the current state *curr* after the occurrence of the event σ according to the environment *env*. The second one is the updated set of state variables after the actions are executed.

Here again, the definition collects recursively the next states of the potential ASTD hierarchy (except for *elementary*). This recursion stops when an `elementary` ASTD is reached.

```
NextState expression (astd : ASTD(St, Ev, Var), σ : Ev,
      curr : ASTDState(St, Var), E_e : Var ↠ ℤ)
  well-definedness condition ...
  recursive definition
    case astd:
        Elementary(inv) => ...
        Automaton(i, f, ..., inv, mapping) =>
            aut_1 ∪ aut_2 ∪ aut_3
        Sequence(fst, snd, attr, initAttr, inv) => ....
        Closure(..) => ...
        Guard(...) => ...
```

Listing 6: Transitions rules from ASTD

To illustrate a transition rule of the operational semantics, we have chosen to unfold the definition of the set aut_2 in the non-deterministic transition associated to the `Automaton` constructor defined in [39]. The aut_2 rule below defines the semantic inference rule that produces the next ASTD state $(\mathsf{aut}_\circ, n, E', s')$ and its environment from an internal transition $s \xrightarrow{\sigma, E_g, E'_g}_{a.\nu(n)} s'$ in the automaton $a.\nu(n)$ under a set of hypotheses H.

$$\mathsf{aut}_2 \ \frac{s \xrightarrow{\sigma, E_g, E'_g}_{a.\nu(n)} s' \quad H}{(\mathsf{aut}_\circ, n, E, s) \xrightarrow{\sigma, E_e, E'_e}_{a} (\mathsf{aut}_\circ, n, E', s')}$$

In Listing 7, this inference rule is translated into Event-B as a set of pairs composed of an ASTD state and an environment collecting the state variables. Lines 1-4 provide definitions, Line 5 corresponds to the antecedent of the aut_2 rule. Finally, Lines 6-8 define the assumed constraints and restrictions on the environment state variables using overriding $\lhd\!\!\!-$, domain subtraction $\lhd$ and restriction $\lhd$ operators.

```
{aut∘(E', n, s') ↦ E'_e | n, E, s, s', E'_e, E_g, E'_g, E' ·
  1. E'_g ∈ Var ↠ ℤ
  2. ∧n = StateName(current)
  3. ∧s = AutSubState(current)
  4. ∧E = AutCache(current)
  5. ∧s' ↦ E'_g ∈ NextState(mapping(n), σ, s, E_g)
  6. ∧E_g = E_e ⩤ E  // The symbol "⩤" means override by
  7. ∧E'_e = E_e ⩤ attr ⩤ E'_g  // "⩤" means domain subtraction
  8. ∧E' = attr ◁ E'_g  // "◁" means domain restriction
}
```

Listing 7: Event-B definition of the aut_2 inference rule

Each inference rule of the ASTD operational semantics of [39] is expressed similarly in theory `ASTDBehaviour`.

V. Modelling specific ASTDs in EB[ASTD]

Thanks to the ASTD meta-model defined in Section IV, it is possible to define specific ASTDs (instances). Following

the EB4EB framework, two instantiation mechanisms are possible. The first one, *deep instantiation*, is based on a direct instantiation of the type parameters and on the application of the defined composition operators. The second one, *shallow instantiation* relies on a generic Event-B machine using the specific operational semantics inference rules associated to each ASTD constructor. It is used for animation purposes.

A. Deep Instantiation

This instantiation mechanism makes it possible to define ASTD instances in an Event-B context, as concrete elements of the ASTD data-types. A generic template of a context corresponding to this instantiation mechanism is provided in Listing 8. The generic type parameters of the ASTD theories are instantiated with enumerated sets describing ASTD states, events and variables.

```
CONTEXT ASTD_Ctx
SETS St, Ev, Var
CONSTANTS
  s0, s1,..,                    // a collection of statenames
  e1,e2...,                     // a collection of events
  v1,v2...,                     // a collection of variables
  root, B, C, D...,             // some ASTDs
AXIOMS
  axm1: partition(St,{s0},{s1},..)
  axm2: partition(Ev,{e1},{e2},...)
  axm3: partition(Var,{v1},{v2},...)
  //Define each ASTD by composition
  axm4: C = Operator_1(...)
  axm5: B = Operator_2(C,...)
  axm6: root = Operator_3(B,...)
  //where Operator_i ∈ {Automaton, Sequence, Closure, Guard}
THEOREMS
  thm_of_welldefinedness: ASTD_WellCons(root, ∅)
END
```

Listing 8: Squeleton of deep instantiation

All core components of ASTDs, described in the `ASTD-Struct` theory such as ASTD automaton, sequence and operators, are provided in the axioms.

Theorem `thm_of_welldefinedness` uses the operator `ASTD_WellCons` to ensure the consistency of the defined ASTD. The generated THM PO must be discharged to ensure the correctness of the instantiated model. In addition, other POs related to the well-definedness conditions of the operators borrowed from the `ASTDStruct` theory are also generated.

Application to the illustrative example of Fig 2. Listing 9 presents the Event-B context corresponding to the ASTD of Fig 2. Enumerated sets are defined for *Ev*, *St* and *Var* in *axm1-3*. The ASTD automaton `root` is defined in *axm4*. In addition, a collection of axioms (*axm12-17*) is presented to relate the `elementary`, `automaton`, and other composition operators according to the example of Fig 2.

```
CONTEXT IllustrativeExample
CONSTANTS
  e1, e2, e3,
  s0, s1, s2, s3,
  astd, B, C, D, E, F,
  xA, xB, xC, xD, xE, xF
AXIOMS
axm1: partition(Ev,{e1},{e2},{e3})
axm2: partition(St,{s0},{s1},{s2},{s3})
axm3: partition(Var,{xA},...,{xF})
axm4: root ∈ ASTD(St, Ev, Var)
...
axm12:
  F = Automaton(
      s2, //initial state
      {s3}, //final states F
      {s2,s3}, //all state names
      {e2,e3},//events
      {e2 ↦ (s2 ↦ s3), e3 ↦ (s3 ↦ s2)},//transitions
      ...
      {xF}, // attributes
      {...}, // initialisation
      {env · (env ∈ Var ⇸ ℤ
       ∧xF ∈ dom(env) ∧ env(xF) ≥ 0) | env}, //invariant
      {...} //hierarchy function
      )
axm13: E = Guard({...}, F, {xE}, ...})
axm14: D = Automaton(...)
axm15: C = Closure(D, ...)
axm16: B = Sequence(C, E, ...)
axm17: root = Closure(B, ...)
THEOREMS
thm_of_welldefinedness: ASTD_WellCons(root, ∅)
END
```

Listing 9: A deep instance of the ASTD example

Finally, in the theorem clause, the `ASTD_WellCons` operator is invoked to check the well-definedness of the ASTD, as well as the correct use of ASTD compositional operators.

B. Shallow Instantiation for model animation

The availability of the Rodin tool suite represents one of the benefits of using Event-B as the base formal modelling language. Deep instantiation is useful in a proof perspective, as well-definedness and theorem POs associated to an Event-B context need to be discharged.

The second instantiation mechanism (so-called *shallow* instantiation) available in Event-B consists in exploiting the operational semantics introduced in Section IV-D. It relies on the definition of a generic Event-B machine (see Listing 10) with two events: an `initialisation` and a `progress` event that respectively refer to the `Init` (see Listing 5) and `NextState` (see Listing 6) operators corresponding to the inference rules of the operational semantics. In this machine, `initialisation` initialises the ASTD state with $current := Init(root, \emptyset)$, while `progress` applies an inference rule and determines the next state as the projection $current := prj1(nxt)$ of the ASTD state returned by the $NextState$ operator.

```
MACHINE ShallowGenAnim
SEES ASTD_Ctx
VARIABLES current
INVARIANTS
  inv1: current ∈ ASTDState(St, Var)
EVENTS
  INITIALISATION
  THEN
    act1: current := Init(root, ∅)
  END
  progress
  ANY evnt, nxt
  WHERE
    grd1: evnt ∈ Ev
    grd2: nxt ∈ NextState(root, evnt, current, ∅)
    grd3: NextState(root, evnt, current, ∅) ≠ ∅
  THEN
    act1: current := prj1(nxt)
  END
END
```

Listing 10: Shallow generic machine for ASTD

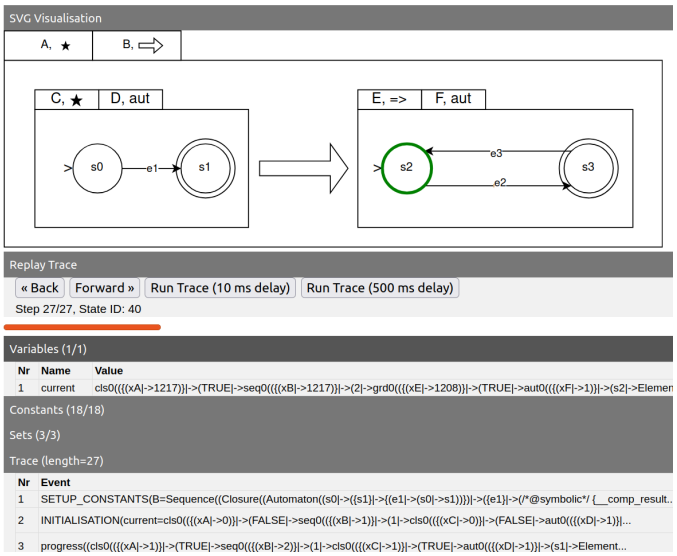

Figure 4: VisB animation applied to ASTD

Animating this machine using the ProB [13] model checker leads to a concrete trace of the ASTD (see bottom part of Fig 4). Moreover, VisB [14] is used to perform visual animation of ASTD operations. A visual animation (see Fig 4) of the illustrative example is available at[1]. This animation is used to validate the execution of ASTDs and to identify potential flaws in the ASTD model.

Last, note that the Event-B machine of Listing 10 can be refined. Indeed, the Progress event may be refined with the specific transitions of an ASTD, leading to another mechanism for instantiation. Here the interest is to check machine consistency using the Event-B method itself (shallow embedding). This approach is not presented in this paper.

VI. Proof Obligations for ASTDs

The last building block of our framework enables the definition of proof obligations, checking their soundness and generating the proof obligations for a given ASTD model. We show how our framework exploits the Rodin Event-B IDE, in particular its proof capabilities, to encode POs. To illustrate our approach, we demonstrate the case of the state invariant POs defined in *pASTD* [6].

A. Definition of proof obligations in Event-B

Proof obligations are defined as predicates modelling a property on ASTDs. Listing 11 shows the Event-B template we use to define a PO.

The Event-B theory `ASTDPO`, importing `ASTDBehaviour` defined in Section IV-D, introduces two predicate operators to model the inductive invariant PO: one for the initialisation and the second one for the inductive case. These two predicate operators invoke the operational semantics operators $init$ and $NextState$ defined in the imported theory.

[1] https://www.irit.fr/EBRP/software/

```
THEORY ASTDPO
IMPORT THEORY ASTDBehaviour
TYPE PARAMETERS St, Ev, Var
OPERATORS
  POi predicate (a : ASTD(St, Ev, Var), ...)
    well-definedness condition ....
    recursive definition ...

  POtr predicate (a : ASTD(St, Ev, Var), ...)
    well-definedness condition ....
    recursive definition ...
```

Listing 11: Deep modeling of POs

Proceeding this way, we obtain a denotational-based definition. Note that other predicate operators may be defined to model other properties or proof obligations.

B. Proof obligation consistency

Once a PO is defined, it is important to check that it is consistent. For this, we rely on the defined operational semantics, and have introduced trace-based semantics for ASTDs. The objective is to check that the POs associated to the denotational semantics entail their specification on the traces.

Listing 12 lays out the theory `ASTDTraces`, describing the notion of ASTD traces adopted from [43]. It imports the `ASTDPO` theory, defining proof obligations.

```
THEORY ASTDTraces
IMPORT THEORY ASTDPO
TYPE PARAMETERS St , Ev , Var
OPERATORS
    IsANextState predicate (a : ASTD(St, Ev, Var) ,
                            s, sp : ASTDState(St, Var))
        well-definedness condition ASTD_WellCons(a, ∅)
        direct definition
          (∃e · e ∈ Ev ⇒ sp ∈ dom(NextState(a, e, s, ∅)))

    IsATrace predicate (a : ASTD(St, Ev, Var) ,
          tr : ℕ ⇸ ASTDState(St, Var))
  well-definedness condition ASTD_WellCons(a, ∅)
  direct definition
    1-    tr(0) = Init(a, ∅)
    2- ∧ (∀i, j · i ∈ dom(tr) ∧ j ∈ dom(tr) ∧ j = i + 1 ⇒
          IsANextState(a, tr(i), tr(j)))
    3- ∧ (tr ∈ ℕ → ASTDState(St, Var)) ∨ // infinite trace
           (∃n · n ∈ ℕ ∧ tr ∈ 0..n → ASTDState(St, Var)
             ∧ (¬CanProgress(a, tr(n))))) //finite trace
```

Listing 12: Trace-based semantic for ASTD

In this theory, the $IsATrace$ predicate operator is parameterised by an ASTD and a trace $tr : \mathbb{N} \pfun ASTDState(St, Var)$, which is a sequence $(s_0 \mapsto s_1 \mapsto ... \mapsto s_n)$ of ASTD states. $IsATrace$ uses the inference rules of the operational semantics previously defined in Section IV-D, and asserts that:

1) the first state of an ASTD trace $(tr(0))$ corresponds to the initialisation of the *upper* ASTD, i.e. $s_0 = Init(a, \emptyset)$.
2) for each consecutive state $s_i \mapsto s_{i+1}$, s_{i+1} is the next state of s_i after an event is triggered. It uses the $IsANextState$ predicate operator, defined in the same theory, to check that $tr(i)$ and $tr(i+1)$ are correct consecutive ASTD states.
3) traces can either be infinite and deadlock-free, or finite and deadlocking.

By leveraging this definition of traces, we may encode the *specification* of proof obligations in terms of properties on the trace. In the case of the ASTD state invariant PO, the associated specification is that the invariant of the machine holds on every state of the trace. It is formalised by the $InvSpecOnState$ predicate operator defined for an ASTD state (see Listing 13).

```
THEORY ASTDCorrectness
IMPORT THEORY ASTDTraces
TYPE PARAMETERS St, Ev, Var
OPERATORS
  InvSpecOnState predicate
      (a : ASTD(St, Ev, Var), s : ASTDState(St, Var))
    well-definedness condition ...
    recursive definition
      case a:
        Elementary(...) => ...
        Automaton(...) => ...
        Sequence(...) => ...
        Closure(..) => ...
        Guard(...) => ...
```

Listing 13: Definition of satisfying invariants

```
thm_of_PO_Correctness:
∀astd, tr·

astd ∈ ASTD(St, Ev, Var) //                    for all astd
∧ASTD_WellCons(astd, ∅) //                        well cons

∧tr ∈ ℕ ⇸ ASTDState(St, Var) //                 and trace
∧IsATrace(astd, tr) //                            of the astd

∧POtr(astd, Var ⇸ ℤ, ∅) //                     when POs hold
∧POi(astd, Var ⇸ ℤ, Var ⇸ ℤ, (Var ⇸ ℤ) ◁ id)
⇒
(∀i · i ∈ dom(tr) ⇒ InvSpecOnState(astd, tr(i)))
// every state in the trace satisfies the invariant
```

Listing 14: Ultimate theorem of PO correctness

Listing 14 shows the consistency theorem. When the proof obligation denoted encoded by the PO_i and PO_{tr} predicates hold, it states that each state in tr satisfies the $InvSpecOnState$ property.

The proof of this theorem needed several intermediate lemmas to be proved. In particular it required the proof of properties related to the operational semantics that identified some errors in the manual specification [6], [38] which have been corrected and reported to the authors.

C. Adding the Generation of proof obligations

The process of discharging the proof obligations is straightforward. It requires to add the definition of the predicates that define the PO in a theorem clause of a context resulting from a deep instantiation.

```
CONTEXT   ASTD_Ctx_PO
EXTENDS   ASTD_Ctx
THEOREMS
  Generation_of_POi: POi(root, ...)
  Generation_of_POtr: POtr(root, ...)
END
```

Listing 15: Automatic generation of PO from ASTD

Listing 15 shows a context extending the deep instantiation context $ASTD\_Ctx\_PO$ of Listings 8 and 9 with two theorems. The Event-B THM PO(see Table VI) generated for theorems carries the invariant PO for ASTDs.

Note that the development shown above applies to all kinds of proof obligations defined for ASTDs.

VII. Conclusion and Future work

This paper presented embeddings of ASTD in Event-B using algebraic theories and Rodin, the Event-B IDE.

A deep embedding allows for the definition of the ASTD abstract syntax and its well-construction conditions, as well as its operational and trace semantics. An approach for defining proof obligations for ASTD properties has been defined. It is illustrated on the proof obligations for ASTD state invariants introduced in [6]. Their soundness has been formally proved and minor errors were found in the original definitions of the authors. It illustrates the benefits of such a formalisation.

An ASTD specification can be instantiated in the theory in the form of an Event-B context that uses the ASTD theory. Well-construction properties can be proved, to ensure the structural validity of the ASTD specification. State invariant proof obligations for an ASTD specification can be generated and discharged using the Event-B proof system.

The ASTD specification can also be animated using ProB, by executing a generic Event-B machine that contains two events, $init$ and $progress$, and which uses the context representing the ASTD specification. VisB can also be used to graphically represent the ASTD specification while it is animated using ProB, and display its current state and current values of its state variables. Rodin, ProB and VisB now provide a complete environment for specifying, verifying and validating ASTD specification, without having to write custom tools from scratch, avoiding implementation errors, and facilitating the traceability of errors found during verification and validation to their source in an ASTD specification.

We believe that the EB[ASTD] framework represents a significant improvement over existing ASTD tools, which consisted mainly of shallow embeddings in the form of model-to-text transformations. Moreover, in case of identified flaws in the ASTD models, this framework offers a trace back to the original ASTD models in terms of ASTD constructs. This facility avoids users to master target modelling languages in case of model-to-text transformations.

For simplicity, the current ASTD theory is defined on integer ASTD state variables. This is not a limitation as it could be overcome by the introduction of a generic axiomatisation of types. Such axiomatisation, already available in EB4EB, through universal types, could be reused for EB[ASTD].

Last, the availability of a meta-model for ASTD opens the path to defining model-to-model transformations and guaranteeing their correctness. For example, a transformation between the EB[ASTD] and the EB4EB meta-models could be defined, and its soundness checked using a (bi-)simulation relationship on the traces. The other path opened by this approach is the possibility to use first order logic as an exchange format. Indeed, in the case of deep embedding, an ASTD is defined in an Event-B context as a FOL term. Exporting such a representation in other proof assistants, to exploit their capabilities, thus becomes possible.

References

[1] M. Frappier, F. Gervais, R. Laleau, and J. Milhau, "Refinement patterns for ASTDs," *Formal Aspects Comput.*, vol. 26, no. 5, pp. 919–941, 2014. [Online]. Available: https://doi.org/10.1007/s00165-013-0286-3

[2] A. R. Ndouna and M. Frappier, "Modelling a mechanical lung ventilation system using TASTD," in *Rigorous State-Based Methods - 10th International Conference, ABZ 2024, Bergamo, Italy, June 25-28, 2024, Proceedings*, ser. Lecture Notes in Computer Science, S. Bonfanti, A. Gargantini, M. Leuschel, E. Riccobene, and P. Scandurra, Eds., vol. 14759. Springer, 2024, pp. 324–340. [Online]. Available: https://doi.org/10.1007/978-3-031-63790-2_26

[3] D. de Azevedo Oliveira and M. Frappier, "Modelling an automotive software system with TASTD," in *Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings*, ser. Lecture Notes in Computer Science, U. Glässer, J. C. Campos, D. Méry, and P. A. Palanque, Eds., vol. 14010. Springer, 2023, pp. 124–141. [Online]. Available: https://doi.org/10.1007/978-3-031-33163-3_10

[4] L. N. Tidjon, M. Frappier, M. Leuschel, and A. Mammar, "Extended algebraic state-transition diagrams," in *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*. IEEE Computer Society, 2018, pp. 146–155. [Online]. Available: https://doi.org/10.1109/ICECCS2018.2018.00023

[5] D. de Azevedo Oliveira and M. Frappier, "TASTD: A real-time extension for ASTD," in *Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings*, ser. Lecture Notes in Computer Science, U. Glässer, J. C. Campos, D. Méry, and P. A. Palanque, Eds., vol. 14010. Springer, 2023, pp. 142–159. [Online]. Available: https://doi.org/10.1007/978-3-031-33163-3_11

[6] Q. Cartellier, M. Frappier, and A. Mammar, "Proving local invariants in ASTDs," in *Formal Methods and Software Engineering - 24th International Conference on Formal Engineering Methods, ICFEM 2023, Brisbane, QLD, Australia, November 21-24, 2023, Proceedings*, ser. Lecture Notes in Computer Science, Y. Li and S. Tahar, Eds., vol. 14308. Springer, 2023, pp. 228–246. [Online]. Available: https://doi.org/10.1007/978-981-99-7584-6_14

[7] T. Fayolle, M. Frappier, R. Laleau, and F. Gervais, "Formal refinement of extended state machines," in *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015*, ser. EPTCS, J. Derrick, E. A. Boiten, and S. Reeves, Eds., vol. 209, 2015, pp. 1–16. [Online]. Available: https://doi.org/10.4204/EPTCS.209.1

[8] J. Milhau, M. Frappier, F. Gervais, and R. Laleau, "Systematic translation rules from ASTD to Event-B," in *Integrated Formal Methods - 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*, ser. Lecture Notes in Computer Science, D. Méry and S. Merz, Eds., vol. 6396. Springer, 2010, pp. 245–259. [Online]. Available: https://doi.org/10.1007/978-3-642-16265-7_18

[9] L. N. Tidjon, M. Frappier, and A. Mammar, "Intrusion detection using ASTDs," in *Advanced Information Networking and Applications - Proceedings of the 34th International Conference on Advanced Information Networking and Applications, AINA-2020, Caserta, Italy, 15-17 April*, ser. Advances in Intelligent Systems and Computing, L. Barolli, F. Amato, F. Moscato, T. Enokido, and M. Takizawa, Eds., vol. 1151. Springer, 2020, pp. 1397–1411. [Online]. Available: https://doi.org/10.1007/978-3-030-44041-1_118

[10] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[11] M. J. Butler and I. Maamria, "Practical theory extension in Event-B," in *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, ser. Lecture Notes in Computer Science, Z. Liu, J. Woodcock, and H. Zhu, Eds., vol. 8051. Springer, 2013, pp. 67–81. [Online]. Available: https://doi.org/10.1007/978-3-642-39698-4_5

[12] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[13] M. Leuschel and M. J. Butler, "Prob: A model checker for B," in *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, ser. Lecture Notes in Computer Science, K. Araki, S. Gnesi, and D. Mandrioli, Eds., vol. 2805. Springer, 2003, pp. 855–874. [Online]. Available: https://doi.org/10.1007/978-3-540-45236-2_46

[14] M. Werth and M. Leuschel, "Visb: A lightweight tool to visualize formal models with SVG graphics," in *Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings*, ser. Lecture Notes in Computer Science, A. Raschke, D. Méry, and F. Houdek, Eds., vol. 12071. Springer, 2020, pp. 260–265. [Online]. Available: https://doi.org/10.1007/978-3-030-48077-6_21

[15] L. Nganyewou Tidjon, "Formal modeling of intrusion detection systems," Theses, Institut Polytechnique de Paris ; Université de Sherbrooke (Québec, Canada), Nov. 2020. [Online]. Available: https://theses.hal.science/tel-03137661

[16] ASTD Team. cASTD and eASTD. [Online]. Available: https://github.com/DiegoOliveiraUDES/ASTD-tools

[17] J. Milhau, "Un processus formel d'intégration de politiques de contrôle d'accès dans les systèmes d'information," Theses, Université Paris-Est ; Université de Sherbrooke (Québec, Canada), Dec. 2011. [Online]. Available: https://theses.hal.science/tel-00674865

[18] T. Fayolle, "Combinaison de méthodes formelles pour la spécification de systèmes industriels," Theses, Université Paris-Est ; Université de Sherbrooke (Québec, Canada), Jun. 2017. [Online]. Available: https://theses.hal.science/tel-01743832

[19] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura, "A metaprogramming framework for formal verification," vol. 1, no. ICFP, 2017.

[20] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, p. 316–344, Dec. 2005. [Online]. Available: https://doi.org/10.1145/1118890.1118892

[21] K.-D. Schewe, F. Ferrarotti, and S. González, "A logic for reflective asms," *Science of Computer Programming*, vol. 210, p. 102691, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000848

[22] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction - CADE*, ser. LNCS, D. Kapur, Ed., vol. 607. Springer, 1992, pp. 748–752.

[23] Y. Bertot and P. Castran, *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 2010.

[24] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[25] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter, "The MetaCoq Project," *J. Autom. Reason.*, vol. 64, no. 5, pp. 947–999, 2020.

[26] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver, "Certicoq: A verified compiler for Coq," in *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[27] B. Fallenstein and R. Kumar, "Proof-producing reflection for HOL - with an application to model polymorphism," in *Interactive Theorem Proving - 6th International Conference, ITP*, ser. LNCS, C. Urban and X. Zhang, Eds., vol. 9236. Springer, 2015, pp. 170–186.

[28] B. Ballenghien and B. Wolff, "An Operational Semantics in Isabelle/HOL-CSP," in *15th International Conference on Interactive Theorem Proving (ITP 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), Y. Bertot, T. Kutsia, and M. Norrish, Eds., vol. 309. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 7:1–7:18. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2024.7

[29] S. Taha, L. Ye, and B. Wolff, "Hol-csp version 2.0," *Archive of Formal Proofs*, April 2019, https://isa-afp.org/entries/HOL-CSP.html, Formal proof development.

[30] J.-R. Abrial, *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[31] C. Muñoz and J. Rushby, "Structural embeddings: Mechanization with method," in *International Symposium on Formal Methods*. Springer, 1999, pp. 452–471.

[32] J.-P. Bodeveix and M. Filali, "Event-b formalization of Event-B contexts," in *Rigorous State-Based Methods*, A. Raschke and D. Méry, Eds. Cham: Springer International Publishing, 2021, pp. 66–80.

[33] S. A. Schneider, H. Treharne, and H. Wehrheim, "A CSP account of Event-B refinement," in *15th International Refinement Workshop,*

Refine@FM, ser. EPTCS, J. Derrick, E. A. Boiten, and S. Reeves, Eds., vol. 55, 2011, pp. 139–154.

[34] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[35] S. A. Schneider, H. Treharne, and H. Wehrheim, "A CSP approach to control in Event-B," in *Integrated Formal Methods - 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*, ser. Lecture Notes in Computer Science, D. Méry and S. Merz, Eds., vol. 6396. Springer, 2010, pp. 260–274. [Online]. Available: https://doi.org/10.1007/978-3-642-16265-7_19

[36] P. Rivière, N. K. Singh, and Y. Aït-Ameur, "Reflexive Event-B: Semantics and correctness the EB4EB framework," *IEEE Trans. Reliab.*, vol. 73, no. 2, pp. 835–850, 2024. [Online]. Available: https://doi.org/10.1109/TR.2022.3219649

[37] P. Riviere, N. K. Singh, Y. Aït-Ameur, and G. Dupont, "Formalising Liveness Properties in Event-B with the Reflexive EB4EB Framework," 2023.

[38] D. de Azevedo Oliveira and M. Frappier, "TASTD: A real-time extension for ASTD," in *Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings*, ser. Lecture Notes in Computer Science, U. Glässer, J. C. Campos, D. Méry, and P. A. Palanque, Eds., vol. 14010. Springer, 2023, pp. 142–159. [Online]. Available: https://doi.org/10.1007/978-3-031-33163-3_11

[39] ——, "Technical report 27 - Extending ASTD with real-time," Université de Sherbrooke, Tech. Rep., 2024. [Online]. Available: https://marcfrappier.espaceweb.usherbrooke.ca/Papers/report-27.pdf

[40] C. E. Jabri, M. Frappier, T. Ecarot, and P. Tardif, "Development of monitoring systems for anomaly detection using ASTD specifications," in *Theoretical Aspects of Software Engineering - 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8-10, 2022, Proceedings*, ser. Lecture Notes in Computer Science, Y. Aït-Ameur and F. Craciun, Eds., vol. 13299. Springer, 2022, pp. 274–289. [Online]. Available: https://doi.org/10.1007/978-3-031-10363-6_19

[41] G. Dupont, Y. Aït-Ameur, N. K. Singh, and M. Pantel, "Event-B hybridation: A proof and refinement-based framework for modelling hybrid systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, pp. 35:1–35:37, 2021. [Online]. Available: https://doi.org/10.1145/3448270

[42] I. Mendil, Y. Aït-Ameur, N. K. Singh, G. Dupont, D. Méry, and P. A. Palanque, "Formal domain-driven system development in Event-B: Application to interactive critical systems," *J. Syst. Archit.*, vol. 135, p. 10pas98, 2023. [Online]. Available: https://doi.org/10.1016/j.sysarc.2022.102798

[43] P. Riviere, N. K. Singh, and Y. Aït-Ameur, "EB4EB: A framework for reflexive Event-B," in *26th International Conference on Engineering of Complex Computer Systems, ICECCS 2022, Hiroshima, Japan, March 26-30, 2022*, 2022, pp. 71–80. [Online]. Available: https://doi.org/10.1109/ICECCS54210.2022.00017