Tool Designed for translation from *Mediator* to Z3 and verification of its validity

Siyi Song and Zichen Tang

Peking University, No.5 Yiheyuan Road, Haidian District, Beijing, 100871, P.R.China

Abstract. As a specific language targeted at modeling and verification, *Mediator* is developed to describe properties in normalized files in the '.med' format, and to realize its functions through a set of tools. In this article, we focus on the design of a simple translation tool from *Mediator* to Z3 Python encodings suitable for SMT-based verification. With a basic understanding of the language *Mediator* itself, we try to analyze the structure and grammar in order to convert it into the more mature solver Z3 to verify the validity of given models. We also have some examples in this article as a preliminary attempt. ...

Keywords: Mediator, Z3, language converter, verification

1 Introduction

With the structure of hardware systems getting more and more complicated, formal verification is needed to ensure validity. Unlike conventional testbench method which only covers limited situations and may miss potential mistakes, domain-specific modeling languages
(DSL) such as *Mediator* enable compact, expressive descriptions of automaton and testbench scenarios.

Mediator is often used to describe hardware structure, signal interaction and testbench environmentm, while Z3 is a widely applied SMT solver for hardware property verification. Further introduction of *Mediator* and Z3 will be included in the next section.

Despite the fact that *Mediator* is convenient for modeling,there are currently limited support to translate *Mediator* to powerful SMT solvers such as Z3 so as to perform automated verification using , making it hard to directly apply *Mediator* models to formal solution.

In this article, we make an attempt to build a translation tool from *Mediator* to Z3, with two examples as tests of validity. A *Mediator* 'Automaton' or 'System' is taken, and flattened, and its structure is normalized. Then a z3 script which encodes a bounded unrolling of system behavior is emitted, enabling bounded model checking and other SMT-based verification techniques.

[0] Siyi Song is responsible for writing this article and provides some ideas for the realization of the project, while Zichen Tang mainly contributes to the specific programming part. Both take part in the analysis process.

This document is structured as follows. Section 2 covers background on Mediator and Z3. Section 3 reviews related work. Section 4 formalizes the translation and explains implementation details. Section 5 presents the Testbench case study and verification setup (with placeholders). Sections 6 discuss results, limitations, and future work.

2 Some Background Information

Now that we are trying to convert *Mediator* to z3, we must have an overall understanding of both languages. Here are some basic introduction of both languages, which is useful in our work.

2.1 The language *Mediator*

*Mediator*is a DSL for modeling reactive systems using entities like automata and systems, and is often used to specify interactions among components.

Types of terms *Primitive types* include common types similar to that we use in C++ and Python: integers(int), bounded integers(int l$\cdots$ r), real numbers(real), boolean values(bool), characters(char) and finite enumerations(enum $item_1, \cdots, item_n$).

Compsite types are more complex, and are at most times, combination of *Primitive types*. Examples include Tuple, Union(a combination of different types), Array and List(static and dynamic list of variables), map, struct, etc.

Parameter types can be any concrete type, and include abstract type and interface type.

Remark Subtyping rules indicate that certain order relations exist between different types, e.g. bounded integers can be assigned to the int type. This is an important aspect in verification.

For the purpose of translation, we asusume an operational semantics where at each global step one process executes a single transition (interleaving semantics). The translator optionally supports alternative semantics(e.g. synchronous steps) via a configuration parameter.

Functions, Automaton and System *Functions* in *Mediator* can be simplified into mathematical functions in that they involve no external variables. *Automaton* plays an important role in the work of formal verification. It contains local variables and typed ports.

What's unique about it is *transitions*, which are statements describing the update of variables being operated if certain conditions are satisfied. Such activating conditions are called guard, hence we denote the transition with guard -¿ statements. *Statement* in *automata* are divided into two categories. Firstly, we have assignment statements that update new values. We also have synchronizing statement when joining multiple automata. *System* is a collection of smaller components like *automata* to make it easier to use.

Something more about automaton Here we have a simple mathematical description of automaton.

Guards:$g : Conf(A) \to Bool$

Assignment Statements:$s_a : Conf(A) \to Conf(A)$

Standardize the expressions Since quite different statement combinations can actually share the same meaning, we need to introduce a standard form of transitions and automata.

Converting a hierarchical system to an automaton

1. Turn all entities into canonical automata
2. Rebuild the flattened automaton
3. Put the transitions together.

The execution of an automaton can be regarded as a discrete time sequence, where in each step we:

1. Evaluate the guard of all transitions, choose one transition whose guard is satisfied (if there is more than one, then we choose at random according to the semantics or let the scheduler decide)
2. Execute the statements of the transition and update variables
3. Move on to the next step

Remark We need to be aware of external transitions since they must synchronize with other transitions in other entities. However, through the application of the labeled transition system, the only difference between an internal transition and an external transition is that the latter may contain more than one assignments.[2]

2.2 Satisfiability Modulo Theories Solver Z3

Satisfiability Modulo Theories, or SMT, are basically a decision problem for logical formulas with respect to background theories like arithmetic, bit-vectors, arrays and uninterpreted functions. It is, in fact, an extension of SAT. Designed by Microsoft Research, Z3 is a mature, high-performance, state-of-the-art and widely used SMT Solver with specialized algorithms that supports various theories and languages like linear arithmetic, uninterpreted functions, arrays, bitvectors, etc. Here we use the Python API for execution and result parsing, which provides convenient constructors for creating Int, Real, Bool, and Array variables, building formulae and invoking solver queries. Under circumstances of bounded model checking, we often unroll system transitions for some finite bound. That is, listing out all the possible status at step k of the system, encoding the status and transition into SMT constraints and using Z3 to find counterexample (as long as status at one point violates the safety property), which implies the invalidity of the system, hence making it possible for Z3 to judge.[3][4][7]

Key functions of Z3 includes:

1. Variable Declaration: ¡type¿('variable_name')
2. Logic: And/Or/Not(...)
3. Condition: If(cond, then, else)
4. Solver control: Solver(),s.add(),s.check(),s.model()
5. Recall: s.push(),s.pop()

Common steps include:

1. Declaration of all variables
2. Add initial conditions
3. Add Or(And(guard,updates))of transition to each step
4. Add (Or(Not(P_0),...,Not(P_k)))→s.check()

If, however, further examination is required, push/pop can be applied to test different k or constraints using the same solver.

2.3 ANTLR and Parse Tree

ANTLR ANTLR, or Another Tool for Language Recognition, is a popular parser-generator. A grammar file in the format of .g4 is used to describe the semantics of the language, and ANTLR will generate a Lexer(which breaks the code into tokens) and a Parser(which resembles the tokens into a parse tree).

In this project, ANTLR is responsible for converting the source code in the format of .med to a parse tree before turning it into more friendly AST.

Parse Tree Parse tree is a tree-shaped structure generated directly from grammar, with each node a grammar rule or token. It contains the complete grammar information and is usually much longer than AST.

Parse tree is the mirror of the semantics that keeps all the details, while AST is a simplified and abstract structure that only contains the semantically important nodes.

In our project, we follow the following steps:

1. Read the source code .med
2. Create Lexer and Parser using ANTLR, and generate parse tree
3. Traverse the parse tree and map the grammar nodes to AST items
4. AST can be directly used by Generator/Simulator after semantic analysis.

3 Related Work

3.1 Hardware Verification using SMT Solvers

Contemporary hardware verification increasingly uses SMT solvers as tools for bounded and symbolic checks. Early and foundational work on bounded model checking(BMC) demonstrated how unrolling transition relations and solving the satisfiability problem can find bugs efficiently.[8] Toolchains such as the Yosys project [9] and its SMT based model checkers translate Verilog/RTL into SMT

formulas solvers like Z3 to carry out property check and bounded proofs. Such toolchains are mature and effective for standard hardware description languages (HDLs), but they are intended for Verilog/VHDL input, which is low-level semantics in the translation to SMT.

3.2 Domain-Specific Languages for Testbench or Hardware Modeling

Domain-specific languages and frameworks are widely applied in hardware verification to express testbenches, stimuli, and behavioral scenarios at higher abstraction levels than RTL. Industrial practice relies heavily on SystemVerilog together with the Universal Verification Methodology (UVM) for constructing rich, reusable testbench components and constrained random stimulus. SystemVerilog/UVM provide extensive libraries and a methodology tuned for large-scale verification [11]. Complementary commercial and legacy DSLs such as Specman/e and assertion languages like PSL (Property Specification Language) are also used to express monitors, properties and checkers in verification flows [12]. At a slightly different abstraction level, SystemC and Transaction-Level Modeling (TLM) support cycle-approximate or untimed modeling of communication and are commonly used for early-stage verification and architectural exploration [13].

In the formal methods research community, specialized modeling formalisms such as UPPAAL for timed automata and Promela/SPIN for asynchronous protocol checking provide domain-focused languages and toolchains tailored to particular classes of verification problems [14][15]. These DSLs emphasize behavioral modeling and property specification rather than gate-level implementation details.

Compared to these established languages, *Mediator* fits the category of a high-level modeling DSL that focuses on automata, guards, transitions and properties in a readable, structured way. Unlike Verilog/VHDL which are geared towards cycle-accurate hardware semantics, or SystemVerilog/UVM which target industrial testbench engineering, *Mediator* is more abstract and centered on behavior: it is intended to describe component interactions, test scenarios and properties at a level that is closer to system behavior than to synthesizable RTL. This difference motivates a dedicated translation and encoding strategy: instead of preserving cycle-accurate bit-level semantics (as many HDL→SMT pipelines do), a *Mediator*→SMT approach can exploit the language's higher-level structure (for example, by flattening structs to fields, encoding enums as named integer constants, and mapping properties directly to state-formula checks) and focus on verifying behavioral correctness and testbench properties.

3.3 Program/Model Translation to SMT

There have been various projects on translation of programming languages and models into SMT (or SAT) to enable automated verification. In software verification, Bounded Model Checking (BMC) and SMT-based symbolic checking have

been effectively applied to C and embedded C programs by translating program control-flow and data manipulations into quantifier-free formulas that are decided by modern solvers [16]. Intermediate verification languages such as Boogie separate front ends from back ends, serving as an Intermediate Representation that is well suited for subsequent translation to SMT and modular verification architectures [17].

In model-based verification, numerous projects translate state-based models (statecharts, UML state machines, timed automata) into target logics or tools (SMV, Promela, SMT encodings) to check reachability and temporal properties; the general recipe is to construct an intermediate representation of states and transitions, unroll them for a bounded depth when performing BMC, and hand final quantifier-free formulas to an SMT/SAT engine for decision and counterexample generation [18]. For hardware, toolchains such as Yosys[9] with its smtbmc/SymbiYosys frontends translate Verilog/RTL nets into SMT and use solvers to perform bounded checks. These systems illustrate how frontend semantics (e.g., bit-vector operations, synthesis transforms) shape SMT encodings and solver performance.

3.4 Bounded Model Checking (BMC)

Here we introduce the Bounded Model Checking in detail, as it is used in our program.

Bounded Model Checking, or in short, BMC, is a kind of verification method on an automatic model based on SAT/SMT. Usually it checks whether the system violates some certain properties (usually safety properties) by searching on a given number of steps.

Different from traditional searching method which uses BFS or DFS after building a graph, BMC flattens the "reachability of a bad state" problem to a boolean or logical formula under a fixed step of length k, and then gives it to SAT/SMT solve. If the conditions in the formula can be satisfied, the solver will return a counterexample with a length no more than k. [19]

Therefore, we will need the starting point I(s), transition T(s,s') and safety property P(s)(the bad state would then be ¬P). The formula we construct to test the step length k will be:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge (\neg P(s_0) \vee \neg P(s_1) \vee \ldots \vee \neg P(s_k)) \tag{1}$$

If the formula can be satisfied, then there exists a path with a length no more than k but violates P, and the solver will return a concrete sequence. Otherwise, there are no counterexamples. A `sat` result yields a counterexample trace, while `unsat` indicates no counterexample up to bound k.

It is worth noting that BMC cannot directly prove the security of infinite steps, and counterexamples too long will be stressful for the solver.

Our work follows the same high-level pattern — parse source models, form a compact intermediate representation, and emit bounded SMT encodings, but it targets the *Mediator* language specifically. *Mediator's* AST (with structs,

enums, unions and named properties) suggests concrete, pragmatic encoding choices (flattening composite types into atomic fields, representing enums as integer constants, using time-indexed variables v_t for unrolling, and encoding invariants as disjunctions of negated state formulas over time). These choices differ from those in general purpose software frontends and from HDL → SMT flows which emphasize bit-precise, cycle-accurate semantics. By focusing on Mediator's semantics and verification use cases, our generator aims at a practical and readable translation that helps verification engineers and researchers use Z3 directly on higher-level behavioral models.

4 Design of the *Mediator* to Z3 Translation Tool

4.1 Overview of the Translation Process

1. Input: *Mediator* source file
2. Analysis: Parse the *Mediator* source code into AST or intermediate expression
3. Semantic Processing: Analysis and reduction of the signal type, process and chronological order
4. Generate Z3 Python script
5. Call Z3 Solver: Check if there are counterexamples

4.2 Design principles

1. Soundness: The encoding must preserve semantics of the *Mediator* subset targeted i.e. any reachable state in the *Mediator* semantics corresponds to a model in the Z3 encoding.
2. Modularity: Support translation per-process to enable compositional encodings.
3. Simplicity and performance: Based on bounded model checking, we choose encodings that are easy to generate and reasonable for Z3 to handle.

4.3 Realization of the Tool: Z3 Generator for *Mediator*

We design a simple Z3 generator that emits a Z3 Python script performing bounded-safety check (k steps) with minimal support for automaton, integer and boolean local variables, simple guards and assignments.

In our Z3 Generator, we mainly realize the term generator, function generator and eventually, the generator of python code (which is not that important). Similar to the arduino generator, we add it to the *Mediator* project as plugin.

Our core idea is to compress the complex parallel system into a single automaton, which is composed of a series of local variables and transition guards. We flatten the local variables in a chronological sequence, and then translate the transition guards into constraint statements on each step $t \to t + 1$. Finally, we add ¬ property as a constraint to convert the test of properties to SAT problems. Hence by calling Z3 solver to find possible counterexamples within limited steps, the goal of verification can be achieved.

Type mapping *Mediator* is equipped with well-developed type system. In this project, we realize the most commonly used types —**int**,**bool** and **double**, which respectively correspond to three basic types in Z3—**int**,**bool** and **double**. Since the type of the variable must be ensured in the constraint declaration of each step, we design the **to_real** function to convert variables to real type before we conduct the division of integers.

For bounded model checking over k steps, we create time-indexed copies of each variable:

For each integer variable v and time t, create an Int('v_t')in Z3Py. For booleans, Bool('b_t').

For the enumeration type **enum**, we build a group of global variables to record the relationship between the enum constant and int values, so that the generator correctly understand related statements and directly use the corresponding int values of the enum constant.

Furthermore, for more complicated requests, we still adopt the flattening strategy ($p = \{x : int, y : int\} \rightarrow p_x : int, p_y : int$) Currently we have realized the support of the **struct** type.

Function generation The design of functions in *Mediator* allows them to be precisely expressed like mathematical formulae. To be more specific, the equations in *Mediator* can only influence local variables in the function itself. Hence the impact of the function on the external part all comes down to returning a single value.

Base on what is mentioned above, we might as well translate it directly into a function in python and consider it an item in the constraints.

Variable declaration The variables in *Mediator* change during the running process of the code. However, when we conduct Bounded Model Checking (which we have introduced above), we require that the variables we declare to be precise. While we carry out Bounded Model Checking with constraint programming, we flatten the running process of the model in the order of time. Hence, before everything starts, we must declare every variable under the time t i.e. <varname> _t. After we finish the variable declaration, we need to initialize the variables or Z3 solver may assign random values to the initial state of these variables, leading to the appearance of fake counterexamples.

In the design of *Mediator*, local variables are ensured to have their own initial values. However, one subtle point worth noting is that in principle, we do not want ti initialize the interface variables.

For example, assume we have a component a, with an input interface p whose value type is $p_{value} : int$, and the property we hope it would satisfy is that the value p gets is always less than 2. If we normally set the initial value of int to 0, a counterexample will come into being(unsat will never be satisfied), as long as the model itself is not contradictory, and we will always get that the result of not satisfied, which, apparently, is unreasonable.

Therefore, we consider using Z3 for a solution on the constraint set of property first to get initial value set of port value in correspondence with property constraints, which we use to intialize the interface variables. It should be noted that this will not influence the subsequent behavior of the automaton because the model itself has guaranteed that it will not be called after the initialization of the interface variables until it has been correctly assigned new values.

Transitions We use constraint statement to express transitions. A typical transition in *Mediator* consists of guard and statement. In an automaton that is generated by scheduling, all transitions are in a transition group, and every transition is canonical[2].

The logic of our work is quite simple: by taking advantage of grammatical parse tree, we traverse every transition in a scheduled automaton. For each transition, we obtain its guard and assignment statement, map the original *Mediator* item to the Z3 item under the corresponding time step, and then connect them with And().

When we deal with the assignment statements, we maintain a list in the realization to take a record of the variables in the transitions that have been assigned values and the corresponding assignment items.

This functions in two aspects. For one thing, when some variables did not change in this step, we need to let Z3 get aware of this. Thus, we need to add an extra unchanged variable constraint $x_{t+1} = x_t$. For another, in the realization, the left-hand side is the variable at time step t, while the right-hand side is the variable at time step t+1 by default.

However, if in the assignment statement sequence of the original model, one variable is first assigned a value on the left-hand side, and is then used to assign another value on the right-side hand of another equation, this will eventually lead to inconsistency in the semantics.

Therefore, we build a temporary mapping list storing the key-value pair from assignment items to assignment expressions. When translating one transition, we try to translate the process of each assignment statement in order —every time we meet an assignment statement, we first search the key set of this mapping list to decide whether the variables used for assignment have been assigned beforehand. If so, we simply swap it for its expression. After that, we get a new assignment statement, and we repeat the process. In this way, we can prove that ultimately, our constraint statement must be t+1 on the left-hand side and t on the right-hand side, and is correspondent to the original semantics.

After we finished the translation of transition one by one, we only need to connect them with Or() to get the transition constraints before applying it to each time step. With that, we get all constraints that the model itself contains.

Finally, from the aspect of semantics, there is no order for transitions in the transition group, i.e. considering the activated (where guard is satisfied) transition in the group in a certain time step, it is totally random which transition is carried out.

When we try to describe transitions with constraint conditions, such properties are naturally satisfied. On the one hand, the status transition of each variable is strictly defined under constraint statement. Hence, the transitions in Or() are mutually exclusive, i.e. only one can be executed at a time. Furthermore, Or() itself is not equipped with order, so if many transitions are activated, the property of being random can be guaranteed.

Note A strange thing here appears. According to [2], canonical transition is defined to be statement after guard without sync statement —single assignment statement with the form of $a, b, c, d, ... = t_1, t_2, t_3, t_4, ...$(here t_i is item/expression). But the output transition of the model during the actual running process of the schedule is not what it is supposed to be.

It is worth mentioning that on the one hand such way of assignment is not naturally supported. In the real parsing process, it will be interpreted as tuple assignment, yet currently we only support single variable assignment and struct assignment.(In fact, we assign values to the flattened struct, hence assigning values to multiple variables)

On the other hand, such way of assignment almost naturally reject the existence of the situation mentioned above, where a variable is assigned a value before it is used to assign another variable. This is its upside.

Property Verification Property verification is the essential target of the translation from *Mediator* to Z3 Python. We first expand schedule algorithm. While it still can compress and integrate the automata in the system, it collects properties defined under the automata and conduct the consistent variable rename process.

Currently we parse the source *Mediator* code in Z3 Generator, extract the properties and use regular expression and all kinds of operations to guess what the names of the variables are after the schedule. In this way,

4.4 Java Code Realization

Listing 1.1. Excerpt of our translation tool (Z3Generator.java))

```
private String renderZ3Script(Automaton autom, int k) throws ValidationException {
        StringBuilder sb = new StringBuilder();
        generatePreamble(sb);
        prepareVariables(autom);
        generateEnumDefinitions(sb);
        generateVariableDeclarations(sb, k);
        generateInitConstraints(sb, autom);
        generateTransitionConstraints(sb, autom, k);
        generatePropertyVerification(sb, autom, k);
        generateTypeSafetyVerification(sb, k);
        generateFooter(sb);
        return sb.toString();
}
```

Now we take a deeper look at the functions mentioned above.

generatePreamble(sb) This function serves as preparation for the project. "From Z3 import*" imports Z3 Python and what follows is using RealVal and ToReal to convert all integers and floats to real values that can be identified by Z3 Python.

prepareVariables(autom)

- Get the local variable declaration set vars.
- Clear or reset variables used for storage or map, and set the enumeration counter to 0
- Traverse every VariableDeclaration (that may contain several identifiers id):
  - For each id, analyze its type
  - Deal with StructType/EnumType/UnionType/other types respectively according to the real subtype of resolved(=resolveType(vt)).
  - Put the results into varFields (key= variable id, value= LinkedHashMap that stands for string→ type map).

generateEnumDefinitions(sb) This function maps enumerate items to texts and puts them together with StringBuilder. By traversing the enumItemConstMap and the Map it, we convert the name of the constant and the present serial number to a line of texts and append it to sb. Each time an enumeration is dealt with, an empty line is added as separation.

generateVariableDeclarations(sb, k) This function is used to generate atomic variable declaration texts in Z3 Python for each field of each variable at each time step according to varFields that have been collected and add them to sb.

For each step t(t=0,1,...,k)

- traverse every variable base and its field map b in varFields
- for each field
  - find the atomName: atomName= (field==base)?base:base+"_"+field.
  - get field type ftype=fields.get(field)
  - choose Z3 declaration type according to ftype.
  - generate the string "<atomName>_<t>=<Z3Type>('...')" and add to sb
- add an empty line to sb as separation.

generateInitConstraints(sb, autom) This funcion generates and combines initial constraint texts in Z3 Python, sets initial values for local variables of automata and add these constraints into Solver.

- "s=Solver()\n": create Z3 Solver.
- Traverse VariableDeclaration in the automaton and identifier:
  - get explicit initial value from variable declaration type

  - if init==null, then we provide a default initial value according to resolve type.
    - IntType→IntValue(0)
    - BoundedIntType$rightarrow$((BoundedIntType).getLowerBound())
    - DoubleType→ DoubleValue(0.0)
    - BoolType→BoolValue(false)
  - if we get init != null
    - if init is StructTerm, then we generate constraints for each field of the struct
    - else we turn init to Z3 expressions.
- add an empty line as separation.

generateTransitionConstraints(sb, autom, k) This function generate all possible transition Z3 constraints at each time step t (from 0 to k-1): for every time step we combine all the transitions with Or(...), each transition consisting of its guard and a next status equality of the variable (to be updated all stay unchanged).

- use flattenTransitions to get all single transitions
- for each time step t(t=0,1,...,k-1):
  - create stepCond starting with "Or()", meaning some transition must happen at this time.
  - traverse each transition ts:
    - calculate guard by calling termToZ3(ts.getGuard(),t)
    - initialize two maps
    - traverse the statement list of the transition
    - build an updates list
    - add all atoms that haven't been modified by nextStateValues to updates
    - join updates with comma ','
    - build logic transition expression and add it to stepCond
  - add s.add(stepCond) to sb.
- generate such constraints for each t.

generatePropertyVerification(sb, autom, k) This function converts properties defined in automaton autom into constraints of violating conditions that Z3 can check, combines all violating conditions with "Or(...)" and add them to given sb.

- return if pc, props==null
- write note in sb "#Properties verification" and create violations list
- traverse every item of props
  - get property name propName and property item prop.
  - get the formula pf from prop
  - convert the property formula to the string expression of violating the conditions, and then add it to the violation list.
- if violations != null, then add them after combining them with (Or(...))

generateTypeSafetyVerification(StringBuilder sb, int k) This function generates Z3 assertions that check whether any bounded-integer variable violates its declared lower or upper bound across time steps t=0,...,k.

generateFooter(sb) This function is used to add the ending script for Z3 Python to StringBuilder to check the satisfiability and print counterexamples when satisfied.

- print the results
- s.check()==sat to judge if it is satisfied. if sat, then we get access to the model and print it out.

Some Other Functions in this Program include functions that are used to deal with different types and their corresponding operator respectively (which is to some extent boring yet unavoidable)

String propertyToCheckExpr(StateFormulae formula, int k) where formula is the what we need to examine and k is the time upper bound.

- if formula is GloballyStateFormulae, then
  - we get inner hoping it can be AtomicPathFormulae
  - iterate t from 0 to k and get Z3 expression at time t of the atomic statement
  - return the string standing for the condition that formula is violated.
- if the formula itself is AtomicPathFormulae,then we directly generate the string.
- For other types that we do not currently support, it returns null.

5 Case Studies: Verification of *Mediator* Models

In this section we analyze two *Mediator* model cases using Z3 Solver to test the related properties.

5.1 Testbench Model

Model Description The testbench model encodes a small coordination scenario with:

- An output interface process (`s_OUT`) exposing `value`, `reqRead`, and `reqWrite` signals.
- An input interface process (`m_IN`) exposing `value`, `reqRead`, and `reqWrite`.
- A smoother module maintaining an integer counter `_smoother_0_count` and average `_smoother_0_avg`.
- Control locations encoded as integer variables `s_p_t`.

The translator unrolled the model within the bound of 40 (t=0,1,...,40). The generated Python file declares each variable for every step; initial conditions are asserted at step 0. For each timestep, the transition relation is a disjunction of six alternative guarded actions, each representing a potential *Mediator* transition. A fragment of the generated time-indexed declarations (for clarity) is given below.

Listing 1.2. Excerpt of generated declarations (testbench_z3_check.py)

```
s_p_0 = Int('s_p_0')
s_OUT_value_0 = Real('s_OUT_value_0')
s_OUT_reqRead_0 = Bool('s_OUT_reqRead_0')
...
_smoother_0_count_0 = Int('_smoother_0_count_0')
...
s.add(s_p_0 == 0)
s.add(s_OUT_value_0 == 0.0)
s.add(s_OUT_reqRead_0 == False)
...
# step 0 transition (Or of clauses)
s.add(Or(
And( And(Not(False), Not(s_OUT_reqRead_0 == s_OUT
_reqWrite_0)),
s_OUT_reqWrite_1 == s_OUT_reqRead_0,
s_p_1 == s_p_0, ...),
And( And(Not(False), m_IN_reqRead_0 == False),
m_IN_reqRead_1 == True, s_p_1 == s_p_0, ...),
...
))
```

Property under Verification The property (called `m_safe` in the repository) asserts that the input value remains below 2:

$$\forall t \in \{0, \ldots, 40\}.\ m\_IN\_value_t < 2$$

The negation is encoded as:

$$\bigvee_{t=0}^{40} \neg(m\_IN\_value_t < 2)$$

and passed to Z3. If Z3 returns `sat`, the model gives a violating timestep and concrete values for the variables.

Listing 1.3. Excerpt of the output (testbench_2.py)

```
--------------------
Step 20:
        s_p:5
        s_OUT_value:4
        s_OUT_reqRead:False
        s_OUT_reqWrite:False
```

```
        m_IN_value:0
        m_IN_reqRead:True
        m_IN_reqWrite:True
        _smoother_0_count:5
        _smoother_0_avg:10
--------------------
Step 21:
        s_p:5
        s_OUT_value:4
        s_OUT_reqRead:False
        s_OUT_reqWrite:False
        m_IN_value:2
        m_IN_reqRead:False
        m_IN_reqWrite:False
        _smoother_0_count:0
        _smoother_0_avg:0
--------------------
```

5.2 Medical System

Model Description This file is generated by our tool and flattened chronologically. It models the interaction among the output interface of the sensor, the input interface of the monitor and a safety controller. Each interface has three signals at each time step t: `value`,`reqRead` and `reqWrite`.

For each time step t→t+1, the translator generates an Or statement to describe the possible branches of guarded action. Each branch is constrained by several guards. Typical branches include:

- Empty/Maintaining branch: When nothing special happens, some signals are duplicated or switched at the next step.
- Monitor launching request to read branch: monitor_IN_reqRead=True
- Update sensor_OUT_reqRead according to has_data in SafetyController
- Control of monitor_IN_reqWrite
- Output/Collection branch (key branch): when sensor_OUT_reqWrite_t == True and satisfies a series of handshake conditions, SafetyController captures the current bpm of the sensor
- Monitor consumption behavior: when monitor_IN_reqWrite is triggered and the handshake conditions are satisfied, the monitor reads buffered_status and clears has_data.

The translator declares variables and add constraints one by one for each time step.

Listing 1.4. Excerpt of generated declarations (MedicalSystem_z3_check.py)

```
sensor_current_bpm_0 = Int('sensor_current_bpm_0')
sensor_OUT_value_0 = Int('sensor_OUT_value_0')
sensor_OUT_reqRead_0 = Bool('sensor_OUT_reqRead_0')
sensor_OUT_reqWrite_0 = Bool('sensor_OUT_reqWrite_0')
monitor_IN_value_0 = Int('monitor_IN_value_0')
monitor_IN_reqRead_0 = Bool('monitor_IN_reqRead_0')
```

```
monitor_IN_reqWrite_0 = Bool('monitor_IN_reqWrite_0')
_SafetyController_0_buffered_status_0 = Int('_SafetyController_0_buffered_status_0')
_SafetyController_0_has_data_0 = Int('_SafetyController_0_has_data_0')
```

Property under Verification The property is `monitor_IN_value==0`. Similarly, we list the excerpt of the verification output.

Listing 1.5. Excerpt of the output (MedicalSystem_z3_check.py)

```
--------------------
Step 59:
        sensor_current_bpm:110
        sensor_OUT_value:105
        sensor_OUT_reqRead:False
        sensor_OUT_reqWrite:False
        monitor_IN_value:0
        monitor_IN_reqRead:True
        monitor_IN_reqWrite:True
        _SafetyController_0_buffered_status:1
        _SafetyController_0_has_data:1
--------------------
Step 60:
        sensor_current_bpm:110
        sensor_OUT_value:105
        sensor_OUT_reqRead:False
        sensor_OUT_reqWrite:False
        monitor_IN_value:1
        monitor_IN_reqRead:False
        monitor_IN_reqWrite:False
        _SafetyController_0_buffered_status:1
        _SafetyController_0_has_data:0
--------------------
```

6 Results, Limitations and Future Work

6.1 Evaluation and Discussion

We evaluate the translation along three dimensions: *expressiveness*, *soundness of encoding* and *scalability/performance*.

Expressiveness This translator covers:

- Primitive types: Int, Real, Bool
- Per-process control locations (encoded as Int).
- Guards and assignments including arithmetic and boolean expressions.

However, unbounded queues, dynamic process creation, and higher-order data types such as maps with unbounded keys are not yet supported.

Soundness For the targeted *Mediator* subset the encoding preserves reachable-state semantics under the assumed interleaving semantics: every concrete *Mediator* execution up to k steps corresponds to a model of the Z3 encoding for that bound.

Scalability and Performance BMC unrolling grows linearly with the bound but formula size may blow up due to branching and frame axioms. Practical mitigations include:

- Generate frame axioms only for variables not updated in a clause.
- Use modular per-process encoding and compress identical clauses using macros or `If` constructs.
- Use bounded integers (BitVecs) when numneric ranges are known.
- Run Z3 with incremental solving (push/pop) to reuse contexts for successive bounds.

6.2 Limitations

Boundedness : The current bounded model checking focused implementation provides only bounded guarantees. Unbounded proofs require induction (k-induction), invariants, or PDR/IC3-style methods [20].

Coverage : The translator targets a core *Mediator* subset; constructs such as rich data types, parametric components, or asynchronous unbounded queues need further work.

Performance For larger models, unrolling causes formula growth, and may lead to timeout or memory exhaustion. Sophisticated techniques like abstraction and symbolic transition relations are not currently integrated, so some *Mediator* constructs like unbounded queues and dynamic topology are not supported.

6.3 Future Works

1. Extend support for inductive invariants and k-induction to obtain unbounded proofs.
2. Integrate counterexample-guided abstraction/refinement to handle larger models.
3. Add a translator front-end based on a formal grammar (ANTLR) with richer error reporting.
4. Provide a Docker image and CI pipeline to reproduce verification runs and automatically check example models on push.
5. Add a small property language to allow users to specify LTL-like properties and automatically lower them to BMC queries

7 Conclusion

We presented a translation tool from a practical subset of the *Mediator* language to Z3 Python encodings, enabling SMT-based verification of *Mediator* models. The tool supports state/time-indexed encodings for bounded model checking and was applied to two representative models to demonstrate feasibility. While bounded checks cannot prove unbounded properties in general, this approach provides a practical path for early bug finding and model assurance. Future work aims to broaden language coverage, improve scalability and add inductive proving capabilities.

Notes and Comments. This whole project is homework from Programming Techniques and Methodology taught by Professor Meng Sun from School of Mathematical Sciences, Peking University. We would like to express gratitude to Yi Li, Weidi Sun and Meng Sun for proposing the language *Mediator* so that components and systems can be modeled separately and precisely, and also to Professor Sun, for getting our focus on this topic.

The most essential documents of the project should be Z3Generator.java, a script which translate teh *Mediator* to Z3Python encoding; generated testbench: testbench_2.py, an unrolled transition system with initial conditions, transition disjunctions and a check of the safety property.

References

1. Our code can be accessed on https://github.com/tanzc628/mediator_z3pytranslation
2. Yi Li, Weidi Sun, Meng Sun: *Mediator*: A component-based modeling language for concurrent and distributed systems. Science of Computer Programming 192 (2020) 102438. https://doi.org/10.1016/j.scico.2020.102438
3. Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger: Programming Z3, Microsoft Research https://z3prover.github.io/papers/programmingz3.html
4. Nikolaj Bjørner, Clemens Eisenhofer, Arie Gurfinkel, Nuno P. Lopes, Leonardo de Moura, Lev Nachmanson, Christoph Wintersteiger https://z3prover.github.io/papers/z3internals.html
5. Yi Li and Meng Sun: Generating Arduino C Codes from *Mediator* in: It's All About Coordination: Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab, in: LNCS, vol. 10865, Springer, 2018, pp. 174-188. https://doi.org/10.1007/978-3-319-90089-6_12
6. A list of *Mediator* https://github.com/liyi-david/Mediator-Proposal
7. de Moura, L., Bjørner, N. (2008). Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science, vol 4963. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78800-3_24
8. Biere, A., Cimatti, A., Clarke, E., Zhu, Y. (1999). Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 1999. Lecture Notes in Computer Science, vol 1579. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-49059-0_14

9. The Yosys Project https://github.com/YosysHQ/yosys
10. DeLine R., Leino K.R.M., "BoogiePL: A typed procedural language for checking object-oriented programs", MSR Technical Report, 2005. https://www.microsoft.com/en-us/research/publication/boogiepl-a-typed-procedural-language-for-checking-object-oriented-programs/
11. "IEEE Draft Standard for Universal Verification Methodology Language Reference Manual," in IEEE Std P1800.2/D7, November 2016 , vol., no., pp.1-504, 1 Jan. 2016. https://ieeexplore.ieee.org/servlet/opac?punumber=7755719
12. "IEEE Standard for Property Specification Language (PSL) - Redline," in IEEE Std 1850-2010 (Revision of IEEE Std1850-2005) - Redline , vol., no., pp.1-188, 6 April 2010 https://doi.org/10.1109/IEEESTD.2010.5948310.
13. "IEEE Standard for Standard SystemC Language Reference Manual - Redline," in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) - Redline , vol., no., pp.1-1163, 9 Jan. 2012. https://ieeexplore.ieee.org/servlet/opac?punumber=6581814
14. Guijarro, L. (2004). Analysis of the Interoperability Frameworks in e-Government Initiatives. In: Traunmüller, R. (eds) Electronic Government. EGOV 2004. Lecture Notes in Computer Science, vol 3183. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30078-6_7
15. IEEE Transactions on Software Engineering https://spinroot.com/
16. Clarke, E., Kroening, D., Lerda, F. (2004). A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2004. Lecture Notes in Computer Science, vol 2988. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-24730-2_15
17. Barnett, M., Chang, BY.E., DeLine, R., Jacobs, B., Leino, K.R.M. (2006). Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, WP. (eds) Formal Methods for Components and Objects. FMCO 2005. Lecture Notes in Computer Science, vol 4111. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11804192_17
18. David Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming, Volume 8, Issue 3, 1987, Pages 231-274, ISSN 0167-6423, https://doi.org/10.1016/0167-6423(87)90035-9.
19. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, Roderick Bloem:Handbook of Model Checking https://doi.org/10.1007/978-3-319-10575-8
20. Bradley, A.R. (2011). SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds) Verification, Model Checking, and Abstract Interpretation. VMCAI 2011. Lecture Notes in Computer Science, vol 6538. Springer, Berlin, Heidelberg https://doi.org/10.1007/978-3-642-18275-4_7.