MedTiny: Enhanced Mediator Modeling Language for Scalable Parallel Algorithms

Xiangyu Li, Yihao Zhang, Xiaokun Luan, Xiaoyong Xue, and Meng Sun*
School of Mathematical Sciences, Peking University, Beijing, China
{xyli,zhangyihao}@stu.pku.edu.cn, {luanxiaokun,xuexy,sunm}@pku.edu.cn
*corresponding author

***Abstract*—This paper introduces MedTiny, a streamlined component-based modeling language, evolved from Mediator. Existing modeling languages struggle to provide convenience in expressing recurrent component configurations, which are common in modern large-scale parallel computation models in distributed environments. Furthermore, Mediator frequently conducts repetitive flag manipulations to synchronize between distributed entities. To address these issues, we propose syntactic enhancements, including component array with static foreach iterator and automatic synchronization flag manipulation. These enhancements significantly reduce code size and effectively address the aforementioned problems. After simplification, we successfully incorporate language feature enhancements into MedTiny without compromising its formality. Subsequently, we develop a prototype code generator that targets multi-threaded Python for code execution and PRISM model checker for verification. To demonstrate MedTiny's efficiency, we include a case study on a character counting MapReduce algorithm.**

***Index Terms*—Mediator, MedTiny, Modeling Language, Distributed System, Scalable Parallelism**

1. Introduction

Scalable parallel computing frameworks are everywhere, from powering Google search to managing the blockchain. As these frameworks have become more complex, it has become increasingly challenging to guarantee the correctness of the scalable algorithms that run on them. MapReduce [1], a well-recognized pioneering scalable computing framework, supports the generation of Google search results from vast amounts of raw internet data using a large number of computers. Cloud computing solution Amazon Web Services Elastic Compute Cloud [2] provides Auto Scaling and Elastic Load Balancing features to adjust allocations of computing resources on-demand with just a slider, a design specifically tailored for scalable parallel algorithms. Sharding has also been proposed by some blockchain networks to enhance the computational capabilities, notably in Ethereum 2.0 upgrade proposal [3]. A common characteristic of scalable algorithms used in these frameworks is the ability to define a template for computing instance generation, enabling batch instance creation and configuration. Formal verification is a mathematical technique for proving the correctness of a system with respect to a given specification, and the first step of applying formal verification is to precisely represent the system in a mathematical model. However, the absence of adequate grammar support in existing tools challenges formal method application on scalable parallel algorithms[4].

To the best of our knowledge, existing modeling languages do not have the ability to concisely repeat instantiations commonly found in scalable algorithms. For example, PRISM [5] a state-of-the-art probabilistic model checker, does not have support for the array type, which makes it difficult to model scalable algorithms that use arrays. As we will compare in details in the case study in Section 5, despite PRISM language providing formula and renaming features to reduce code duplication, its code size still grows with buffer sizes and numbers of computing instances. Other formal languages specific to certain tools that we have surveyed, for example, UML[6] and TLA+ [7] similarly lack support for array structures, making the manual creation of scalable models in these languages unrealistic. NuSMV [8], a symbolic model checker primarily used to verify the correctness of finite-state systems, supports the array data type but doesn't allow batch declarations of modules. UPPAAL [9], a tool for modeling and verifying real-time systems, allows collecting communication channels as arrays in additional to data arrays. However, UPPAAL's behavioral unit *process* can't be placed in arrays or created in batches. The model-oriented programming language Umple[10] uses Java code to initialize and execute models, but such Java code is not a part of the model. Umple tool doesn't generate any hint on how the models are connected in the modeling languages it supports. Mediator [11], a two-level hierarchical component-based modeling language, is capable of describing both high-level connected systems and low-level automata behaviors. Mediator aims to involve programmers in formal tools, and as a result, it borrows numerous ideas from programming languages, such as composite data types, imperative sequential statements, and component templates. Though Mediator still has room for improvements, its grammar offers a good foundation to address the instantiation repetition problem.

In this paper, we propose MedTiny, a simplified subset of the Mediator modeling language that allows addition of identifier arrays and static compile-time loops without losing formality. These features allow us to concisely and expressively model scalable parallel algorithms. We have to simplify Mediator first, because it is over-designed, featuring some ill-defined and ambiguous characteristics. We pinpoint some of the most significant design conflicts and limitations: a type system lacking sound and intuitive typing rules, an overly restrictive two-level composition mechanism, and a complicated but inexpressive approach to describe component connection topology. By exploring the implications of Mediator, we've managed to constraint the grammar to a sound, simplified, yet more expressive subset, which we name as MedTiny. Post-simplification, we successfully integrate component array and

2693-9371/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS-C60940.2023.00075

static compile-time loop into MedTiny.

MedTiny also includes an automatic synchronization enhancement that simplifies the expression of synchronized models. This enhancement is particularly beneficial when writing concise and reusable MedTiny libraries. MedTiny inherits Mediator's capacity for both synchronous and asynchronous communications but with augmented automated syntax sugar. Mediator utilizes a two-flag method for synchronization, and flag manipulations must be carefully orchestrated to achieve deterministic synchronization. This management of flags can become mundane, especially in large models. By exploring the semantics of these two flags, we have uncovered the full extent of this synchronization methodology's expressiveness. From the methodology, we have isolated the only two deterministic synchronization modes and incorporated them as automatic synchronization grammar or auto-sync. Auto-sync alleviates the complexity of expressing synchronized models, while still maintaining the capability to represent synchronization failure and nondeterministic synchronizations.

The remainder of the paper is structured as follows: Section 2 introduces the core grammar of MedTiny meticulously elucidating the aspects that significantly deviate from both Mediator and conventional standards. Section 3 details extension of the static iterator and its supporting features. Section 4 demonstrates the expressiveness and implication of MedTiny synchronization and introduces the automatic synchronization enhancement. Section 5 presents a case study. Section 6 identifies the future directions of MedTiny research. Section 7 concludes our work.

2. Core MedTiny

MedTiny is a multi-level, component-based modeling language featuring a type system that covers common programming types. MedTiny can describe both low level action units as automata and high-level connections with flexibility. MedTiny boasts programming language inspired features such as multi-file compilation, user-defined constants and types, and static templates to improve code conciseness and reusability.

In MedTiny, components are first-class citizens, and it's crucial to understand the semantics of the two types of components:

- *Function* components, which are stateless to compute;
- *Automaton* components, which are stateful to compute.

A *Function* component represents a series of assignment statements. By definition, *Function* is call-by-value and guaranteed to terminate, which corresponds to the mathematical concept of a total function. An *Automaton* component acts as an automaton in theoretical computer science, capable of modeling complex software control structures.

A component can contain other components as subcomponents without violating its definition: an *Automaton* may contain either *Function*s and/or *Automata*, while a *Function* can only contain other *Function*s.

MedTiny inherits most of Mediator's syntactical structure, and we make changes to lower language learning curve and to improve clarity. In the rest of this section, we introduce the syntax of MedTiny in Extended Backus-Naur Form (EBNF) where

- Terminal symbols are in `monospaced fonts`;
- Non-terminal productions are encapsulated in ⟨*angle brackets*⟩;
- We use question marks ($^?$) for "zero or one occurence", the star symbol (*) for "zero or more occurences" and the plus sign ($^+$) for "one or more occurences".

We use abbreviated *type_id*, *ent_id*, and so on in the syntax to indicate an identifier binding to a type, entity, etc. The plural form *ids* stands for a comma-separated list of identifiers with the same purpose. Terminating semi-colons are omitted for clarity.

2.1. Program and Dependency

A MedTiny *program* follows the syntax:

⟨*program*⟩ ::=(⟨*dependency*⟩ | ⟨*type_def*⟩ | ⟨*const_def*⟩ | ⟨*func_decl*⟩ | ⟨*autom_decl*⟩)*

Hierarchical static scoping is employed throughout MedTiny. This approach promotes modularity, aligning with the requirements of the component encapsulation principle, and facilitates the structured and predictable organization and management of variables and symbols within the language.

By design, MedTiny disallows use-before-definition and symbol redefinition, thereby eliminating the possibility of recursion. By prohibiting recursions, MedTiny ensures that symbols are specified and used in a well-defined order. This restriction is consistent with the stateless nature of *Function* and contributes to code clarity, while avoiding potential issues relating to undefined behaviors or circular dependencies.

dependency syntax is an `import` command that supports multi-file compilation. Each imported file should contain a program. The `import` command always creates a subscope based on the relative path to the current file, or it can be renamed with the `as` keyword. The syntax of *dependency* is defined as follows:

⟨*dependency*⟩ ::= `import` ⟨*dot_delimited_path*⟩ (`as` ⟨*prog_id*⟩)$^?$

type_def or *const_def* declare global aliases for a type or a constant. *func_decl* or *autom_decl* declare *Function* or *Automaton* templates. These elements will be discussed further in the following subsections.

2.2. Data Type

All MedTiny types can be resolved at compile time. MedTiny data types define directly accessible data slots or memory locations. Primitive data types hold a single value, while composite data types group multiple values. Immediate definitions for data types and type aliases are both considered valid use of types.

Automatic type coercion is permitted from a subtype to its supertype, allowing for seamless usage. However, explicit conversion is required when converting from a supertype to its subtype. This approach ensures type safety.

TABLE I
PRIMITIVE DATA TYPE

Category	Declaration	Value Examples	Domain
Integer	`int`	`-1,0,1,0xBEEF`	$\mathbb{Z}$
	`int` l_{const} `..` u_{const}	$l_{const}, u_{const}+1$	$\mathbb{Z} \cap [l_{const}, u_{const}]$
	`void`	`null`	0
	`bool`	`true, false`	$\mathbb{Z} \cap [0,1]$
	`char`	`'a', 'B', '@'`	$\mathbb{Z} \cap [0,127]$
Real	`real`	`0.1, 1E-3`	$\mathbb{R}$

TABLE II
COMPOSITE DATA TYPE *(T denotes an arbitrary data type)*

Category	Name	Declaration	Value Example
List	array	`T [length`$_{const}$`]`	[1,3,2*3]
	string	`char [length`$_{const}$`]`	"MedTiny!"
	tuple	`(`T_0`,...,`T_{n-1}`)`	(1,2.718,"Hi")
Record	struct	`{`var_id_0`:`T_0`, ..., `var_id_n`:`T_n`}`	{name:"Li",age:20}
	enum	`{`$const\_id_0$`, ..., `$const\_id_{n-1}$`}`	{Red,Green,Blue}
	in_port	`in T`	N/A
	out_port	`out T`	N/A

Primitive Data Type has two variants:

- *Integer* with arbitrary precision and optional bound annotations;
- *Real* number with arbitrary precision.

We choose arbitrary precision numbers to cover mathematical aspects of formal methods. The bit representation of numbers, such as the most commonly used two's compliments, can be considered a subset of arbitrary precision numbers. All bit-wise numerical behaviors can be simulated with *Function*s of *Integer*s and *Real*s.

Bounded *Integer* limits its domain to between lower bound and upper bound inclusively. However, bounds do not participate in type checking; instead they only serve as annotations to provide compilation hints.

`void`, `bool` and `char` are essentially built-in aliases of bounded *Integer*s, matching supported constant literals. A summary of primitive data types is shown in Table I.

Composite Data Type also has two variants, as shown in Table II:

- fixed-length mutable tuple type *List*;
- *Record* type pointing to a subscope.

An *array* carries a list of fields of finite length of the same type. A *string* is an *array* of `char`, matching string literal. A *tuple* creates a finite list of fields of different types. Fixed-length *array* is essentially a special case of *tuple* when they're both mutable as in MedTiny.

Record has a subscope for named members. A *struct* is a variable lookup, and an *enum* is a constant lookup. *enum* values are assigned starting from 0, counting up, thus *enum* type may be used interchangeably with a bounded *Integer* within `0..n-1`. To access its subscope, a *Record* value must have an *identifier*. *in_port* and *out_port* are built-in predefined data *Record*s, and their special semantics will be addressed when introducing automaton declaration.

These composite data types should cover the most commonly used data structures. MedTiny data types define only directly accessible data slots, and core built-in data types shall not have stateful behavioral implications. While dynamic-length *array* and *map* are widely used in modern programming languages, these types inherently carry implicit states. For example, a dynamic-length *array* has an implicit state associated with its current size, while a *map* has implicit states related to key existences. Additionally, dynamic-length *array* `add` and *map* `put` operations should also be encapsulated as component-specific actions, so those types fit into the definition of MedTiny *Automata* but not data types.

Type Alias may point to a data type or a component declaration. Retrospectively, both component and program are also treated as types with a subscope. *type_def* provides the ability to define a type alias:

⟨*type_def*⟩ ::= `typedef` ⟨*type*⟩ `as` ⟨*type_ids*⟩ $^+$

2.3. Expression and Assignment

Expression syntax and execution priorities are taken directly from ANSI C[12] for maximum programmer friendliness, but with the following modifications:

- separation of value assignment from expressions, and removal of inc(`++`), and dec(`--`) operators;
- support for immediate values of *List* and *Record* types;
- removal of pointer operators (`*` , `&`), and repurposing of arrow operator (`->`) found in later C standards;
- limitation of type casting to only allow from *Real* to *Integer*;
- addition of power (`**`) operator with one higher priority than multiplication (`*` , `/` , `%`), and one lower priority than type casting;
- restriction of true value and false value in a conditional expression to be of the same type.

Formal semantics of C expressions have been well-studied in Clight[13], and MedTiny uses the same corresponding semantics, except the numerical behavior is defined by Java BigInteger and BigDecimal library implementation[14].

Constant Expression is compile-time statically evaluable. If all operands in an *expression* are constant, the whole *expression* is constant too. In retrospect, lower and upper bounds in *Integer* type definition and length in *array* type definition can use a *constant expression* without violating full static typing rules.

A custom constant alias may be defined with *const_def*:

⟨*const_def*⟩ ::= `const` ⟨$expression_{const}$⟩ `as` ⟨*const_ids*⟩

Assignment statement has only the simple general form in C syntax:

⟨*assignment*⟩ ::= (⟨$expression_{var}$⟩ `=`)$^?$ ⟨*expression*⟩

The result of left-side *expression* should be a variable, and the right side can be any *expression* subtype of the left side. A single *expression* is also treated as an assignment, and such a partial statement has no behavioral effect but can serve as annotations or hints for code generation. The conditional operator provides the full ability for control flow branching,

so an if-then-else statement is not a necessity. Statements are order-sensitive to maintain readability and correctness of program generation, while the semantics of many model checkers do not require so.

Other statement-like syntaxes, specifically `return` and `sync`, are attached to the enclosing syntax and may not exist independently.

2.4. Component Declaration

A component has its own private local scope. All local variables are inaccessible outside its encapsulation, even to its subcomponents or parent components. Component declaration only specifies how a type of component is created, but it does not instantiate such a component. A component declaration without a body defines an interface, and interfaces are considered component stubs.

Function Declaration has the following syntax:

```
⟨func_decl⟩ ::= function ⟨func_id⟩ ⟨func_interface⟩ ⟨func_body⟩ ?
⟨func_interface⟩ ::= ( ⟨parameter⟩ * ) : ⟨data_type⟩
⟨func_body⟩ ::= {
                  ( temporaries { ⟨data_var_def⟩ * } )?
                  statements { ⟨assignment⟩ * ⟨return_stmt⟩ }
                }
⟨parameter⟩ ::= ⟨var_ids⟩ : ⟨data_type⟩
⟨data_var_def⟩ ::= ⟨var_ids⟩ : ⟨data_type⟩ = ⟨expression⟩
⟨return_stmt⟩ ::= return ⟨expression⟩
```

A *Function* declaration is used to support its semantics of a series of value assignments. Having local data `temporaries` won't violate *Function* semantics, because they can be eliminated using assignment collection methodology introduced in [15].

Additionally, variable initialization is enforced to prevent data use before value assignment, and this initialization is equivalent to the first few `statements`. One and only `return` substatement must be present at the end of `statements` section. In fact, each function has a predefined variable named `@retVal` of the return type, and the final return statement creates a value assignment to this variable. However, `@retVal` is not supposed to be accessed by user code for safety reasons.

Automaton Declaration is fairly complicated, and its structure can be summarized as follows:

```
⟨automaton_decl⟩ ::=( automaton | system ) ⟨autom_id⟩
                     ⟨autom_interface⟩ ⟨autom_body⟩ ?
⟨autom_interface⟩ ::= ( ⟨port⟩ * )
⟨autom_body⟩ ::= {
                   ( states { ⟨stateful_def⟩ * } )?
                   ( inits { ( ⟨assignment⟩ | ⟨connection⟩ )* } )?
                   transitions { ⟨transition⟩ * }
                 }
⟨stateful_def⟩ ::= ⟨data_var_def⟩ |( ⟨entity_ids⟩ : ⟨autom_type⟩ )
```

Automaton local variables are defined only in `states` section. An *Automaton* can have data as `states`. Because flattening a nested component combines subcomponent *Automata* state variables together, it is semantically safe to add the subcomponents to `states` to support internal automaton composition.

TABLE III
PORT'S ACCESS RESTRICTIONS (*`rw` for read-write, `ro` for read-only*)

Direction	value	readReady	writeReady
in	ro (only in read sync)	rw	ro
out	rw (only in write sync)	ro	rw

Data states must have initialization equivalent to the first few *assignment*s in `initializations` or `inits` like those in *function_declaration*. Subcomponent *Automata* are initialized by connecting their ports with *connection*s. From a programming perspective, `inits` only run once before executing any `transitions`, and *Automaton* `transitions` are in an infinite loop constantly trying to make possible state transfers.

Port and Connection serve to define interfaces and topology for *Automata* communication networks. Port and connection may be declared with the following syntax:

```
⟨port⟩ ::= ⟨port_ids⟩ : ( in | out ) ⟨data_type⟩
⟨connection⟩ ::= ⟨out_port_id⟩ | internal ->
                 ⟨in_port_id⟩ | internal
```

Ports are specialized `struct`s with additional access restrictions and special binding rules. All ports within an *Automaton* are publicly exposed for connection. Both *in_port* and *out_port* possess three fields: a `value` of an arbitrary data type, a boolean `readReady`, and another boolean `writeReady`. The direction of a port concerning its owner is indicated through an annotation. An *Automaton* can only read from(write to) its own *in_port*(*out_port*) or its subcomponents' *out_port*(*in_port*), with access restrictions shown in Table III.

In fact, `readReady` is a state variable of the automaton that holds the *in_port*, and `writeReady` is a state variable of the automaton that holds the *out_port*. These flags are initialized to `false`. However, `value` is a temporary variable used for passing value between data communicating transitions. Connecting two ports semantically equates to binding the common port fields to the same variable. The sides of a *connection* can either be a port of a subcomponent or `internal`, specifying that the port is internally connected to the host component. The type of an *in_port* must be a subtype of *out_port* type for a connection to be made.

The semantics of ports and connections will be discussed in greater detail in Section 4.

Transition represents the basic unit of stateful behavior, whether deterministic or not, found only in `transitions` section:

```
⟨transition⟩ ::= ⟨guard_statement⟩ | ND { ⟨guard_statement⟩ * }
⟨guard_statement⟩ ::= ⟨bool_expression⟩ ->
                      ( ⟨assignment⟩ | { ( ⟨assignment⟩ * ) } )
                    | ⟨read_sync⟩ | ⟨write_sync⟩ | ⟨rw_sync⟩
⟨read_sync⟩ ::= { sync ⟨port_ids⟩ ⟨assignment⟩ * }
⟨write_sync⟩ ::= { ⟨assignment⟩ + sync ⟨port_ids⟩ }
⟨rw_sync⟩ ::= { sync ⟨port_ids⟩ ⟨assignment⟩ * sync ⟨port_ids⟩ }
```

A deterministic transition is written in the form of a *guard_statement*, which starts with a boolean *expression* known as *guard*. When the *guard* evaluates to `true`, the *assignment*s it safeguards will be executed. Essentially, the assignments within a deterministic transition form a *Function*. The order of *guard_statement*s is important: the first satisfied *guard* takes precedence, skipping the evaluation of subsequent *guard*s. Once the *assignment*s in the first guard have been executed, a new round of guard evaluation begins from the start.

Synchronization between ports is labeled by `sync` substatement of a transition. A `sync` can occur before all *assignment*s, thereby creating a read transition, and/or after all *assignment*s thereby forming a write transition. If there is only one `sync` without any *assignment*s, the transition is still considered a read transition, but it discards all the values. Transitions without `sync` are internal transitions.

To introduce nondeterminism, multiple transitions may be grouped together using the keyword `nondeterministic` or `ND`. Within the `ND` group, all of the *guard*s are evaluated with equal importance, but only one of the satisfied *guard*s will have its actions executed. If no *guard* is satisfied, the entire `ND` group is overlooked and lower prioritized transitions are considered.

Template is also inherited from Mediator to improve code reusability. A component declaration may contain a list of template symbols, thereby generalizing component declarations. The syntax for templated declaration and the instantiation of a templated component is defined as follows:

```
⟨templated_decl⟩ ::= function ⟨template_ctx⟩ ? ⟨func_id⟩
                      ⟨func_interface⟩ ⟨func_body⟩ ?
                    |( automaton | system ) ⟨template_ctx⟩ ? ⟨autom_id⟩
                      ⟨autom_interface⟩ ⟨autom_body⟩ ?
⟨template_ctx⟩ ::= < ( ⟨type_template⟩ | ⟨const_template⟩ )* >
⟨type_template⟩ ::= ⟨type_ids⟩ : type
⟨const_template⟩ ::= ⟨const_ids⟩ : ⟨type⟩
⟨templated_instantiation⟩ ::=( ⟨func_id⟩ | ⟨autom_id⟩ )
                      ( < ( ⟨type⟩ | ⟨expression_const⟩ )* > )?
```

A *template_ctx* is essentially a collection of local *type_def*s and *const_def*s. When instantiating a templated component, a *type_template* can take any data type, and a *const_template* should only accept $expression_{const}$, just as in *const_def*. A *templated_instantiation* must define all templated fields to match the declared *template_ctx*. Because both *type* and $expression_{const}$ are static, all template resolution can be accomplished during compilation before performing complete static type checking.

3. Enhancement: Identifier Array and Static Iterator

The identifier array and static iterator introduce mechanisms for declaring an *array* of any type and generating a constant number of *assignment*s, *transition*s and *connection*s. The identifier array can group ports and entities of the same declaration into a one-dimensional *array*. A compile-time foreach style loop is added to handle operations on any general *array*s, such as repeatedly assigning values, transitioning states, or establishing connections. We introduce *for_each* and *id_arrays* syntax to support generalized repetition:

```
⟨for_each⟩ ::= for ( ⟨index_id⟩ , )? ⟨item_id⟩ in ⟨array_expr⟩
               ( ⟨repeatable_stmt⟩ | { ( ⟨repeatable_stmt⟩ * ) } )
⟨repeatable_stmt⟩ ::= ⟨assignment⟩ | ⟨transition⟩ | ⟨connection⟩ | ⟨for_each⟩
⟨id_arrays⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ [ ⟨expression_const⟩ ]
               (, ⟨id_arrays⟩ )?
```

for_each can be used in place of any *repeatable_stmt*s, and *id_arrays* may be used among comma separated *ids*. Every *for_each* creates a block scope with *item_id* representing the item in the *array* and *index_id*, if used, to represent the item index. These two identifiers are marked constant.

It's important to note that MedTiny's *for_each* differs from the full-fledged runtime for loop found in programming languages. MedTiny's *for_each* and *id_array* are always expandable at compile-time, thus producing core MedTiny. As a result, these two features are backward compatible with the previously discussed core MedTiny semantics.

The static iterator enables the concise expression of generalized terminating bounded algorithms without introducing additional states. For example, in programming, the greatest common denominator (gcd) algorithm is typically written in a while loop. However, core MedTiny does not support runtime loops. Thus, a `general_gcd` algorithm can only be implemented as an *Automaton*, as illustrated in Code Example 1. Using a static iterator, a generalized `bounded_gcd` can be implemented as a *Function* with the fact that the gcd algorithm will terminate in $2(log_2 n + 1)$ steps (where n is the `max` input value), as shown in Code Example 2. The use of a static iterator makes the `bounded_gcd` closer to programming intuitions, without the need to create extra states.

4. Enhancement: Automatic Synchronous Flag Manipulation

MedTiny *Automata* communicate via ports through connections as delineated in Section 2-D. Communication in MedTiny is asynchronous unless synchronization flags, `readReady` and `writeReady`, are used appropriately. The semantics of port flags and synchronization remain the same as those in original Mediator, except that the flags have been renamed to more accurately reflect their meanings. Upon surveying other modeling languages, we discover that such synchronization is typically written in fewer lines. Inspired by

Code Example 1. Greatest Common Denominator in Core MedTiny

```
automaton general_gcd() {
    states {
        int x=120; int y=36; // input values
        int a=0; int b=0;
        int result=0;
        bool starting=true;
    }
    transitions {
        starting -> {
            (a,b)=x>y?(a,b):(b,a);
            //swap values so a>b
            starting=false; //skip this transtion
        }
        b!=0 -> (a,b)=(b, b%a);
        //iterate until b is 0
        true -> result=a;
        //stores the result
    }
}
```

Code Example 2. Greatest Common Denominator with Static Iterator

```
function <len:int> fill(value:int):int[len];
//fill function is a built-in that returns
    an array of the same value
function <max:int> bounded_gcd(x,y:int 0..max):int {
    temps {
        bool found = false;
    }
    statements {
        (x, y) = (x < y)?(x, y):(y, x);
        for _ in fill <2*((int)math.log<2>(max)+1)>(0)
            {
            //gcd will definitely terminate
            //in this many steps
            found=y==0;
            (x, y) = found?(x,y):(y, y%x);
        }
        return x;
    }
}
```

this, we analyze the implications of these flags in synchronous communications.

From a programming perspective, `readReady` and `writeReady` abstract the implementation details of synchronous communication, such as handshakes in network protocols or mutex locks in thread synchronizations. Setting `readReady` (or `writeReady`) signifies that the reader(or writer) side has confirmed its readiness for data transfer. Data can be transferred atomically and reliably only when both sides are ready or at the "fire" state. In other words, from the "default" state in which both flags are cleared, a reliable synchronous communication must have a deterministic sequence of reaching the "fire" state in which both flags are set and recovering to the initial "default" state to prepare for the next round of synchronization.

As per the ports' definition and access restrictions shown in Table III, only two deterministic sequences exist differentiated by the communication initiator. We have named these sequences active mode and passive mode, each forming a deterministic automaton using the values of `readReady` and `writeReady`:

- Active mode automaton: data are generated upon reader's *active* request, as shown in Fig. 1
  1) `readReady,writeReady:false,false` - Start from the default state.
  2) `true,false` - The reader sets `readReady` to request data.
  3) `true,true` - The writer finds a read request, writes `value`, and sets `writeReady` to confirm `value` is settled.
  4) `false,true` - The reader completes the read of `value` and clears the request.
  5) `false,false` - The writer releases data settle confirmation and reverts to the default state.

  An implication of this sequence is that if the data are not ready, the writer is still expected to send a default value to signal the reader. Otherwise, the sequence stalls until the writer can send valid data;
- Passive mode automaton: data are *passive*ly generated by the writer for processing, as shown in Fig. 2
  1) `readReady,writeReady:false,false` - Start from the default state.
  2) `false,true` – The writer writes `value` and sets `writeReady` to confirm `value` is settled;
  3) `true,true` – The reader finds a ready signal, reads `value`, and sets `readReady` to confirm read completion;
  4) `true,false` - The writer finds read completion and releases data settle confirmation;
  5) `false,false` - The reader clears read completion status and reverts to the default state.

  Similarly, an implication of this sequence is that if the data cannot be processed normally, reader is still expected to accept the data. Otherwise, the sequence stalls until the reader can process the data.

Any other flag sequences apart from the two mentioned above would lead to either a deterministic failure or a nondeterminism regarding the success or failure of a communication attempt. In real-world scenarios, engineers aim to minimize the rate of unhandled faults, which measures reliability. Communication systems are no different: successful communication is definitely the common case for modeling, but modeling fault handling is still critical in problem solving. For this reason of practicality, we retain these flags to model and abstract malformed or unreliable synchronous communications.

From our analysis, functional fault handling only happens on the initiator side in our model. With the complete understanding of this synchronization mechanism, we design a backward compatible syntax that simplifies the expression of two deterministic synchronization modes while maintaining its core expressiveness:

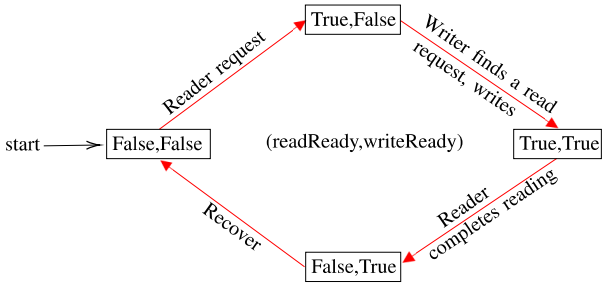

Fig. 1. Active Mode Synchronization

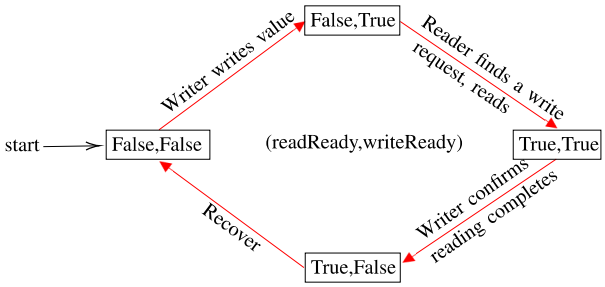

Fig. 2. Passive Mode Synchronization

⟨*link*⟩	::=	⟨*active_link*⟩ | ⟨*passive_link*⟩
⟨*active_link*⟩	::=	⟨*in_port_id*⟩ | `internal <-` ⟨*out_port_id*⟩
⟨*passive_link*⟩	::=	⟨*out_port_id*⟩ | `internal ->` ⟨*in_port_id*⟩
⟨*auto_sync_transition*⟩	::=	⟨*auto_sync_flags*⟩ ? ⟨*guarded_statement*⟩
⟨*auto_sync_flags*⟩	::=	(`auto-sync` | `active-only` | `passive-only`)

link is a refined *connection*, and the left side of the arrow indicates the initiator of the communication. As the initiator only matters when modeling synchronous communications, *link* is equivalent to core *connection* when the communication is asynchronous or when we want to model fault handling. With the information provided by *link*, three kinds of `auto-sync` labels maybe added in front of a data transfer *transition*:

- `auto-sync` adds flags manipulations for both active-style links and passive-style links;
- `active-only` adds flags only if the port is actively linked, and removes the transition otherwise;
- `passive-only` adds flags only if the port is passively linked, and removes the transition otherwise.

The design of `active-only` and `passive-only` allows for the possibility of writing generalized communication libraries in different modes, as only the writer(or reader) is expected to produce error signals in active(or passive) modes.

By combining the enhancement of static iterator defined in Section 3 and automatic synchronous flag manipulation, we can fully implement generalized libraries of `Duplicator`, `Merger`, and `Queue` used in general communication channels as shown in Code Example 3:

- `Duplicator` automaton has one `input` port and an arbitrary number `numOut` of `dupOut` ports. In passive mode, when the `input` port sees a value is ready, `Duplicator` immediately broadcasts the value to all the `dupOut` ports. In active mode, when any of the `dupOut` ports identifies a read request, it checks whether the `input` port happens to be ready and sends its value to the requesting port. If the `input` port is not ready, it sends `null` value to the port.
- `Merger` automaton has an arbitrary number `numIn` of `mergeIn` ports and a single `output` port. In passive mode, it selects the lowest indexed `mergeIn` port with a ready value, and sends the value to the `output`. In active mode, when the `output` port identifies a read request, it checks all the `mergeIn` ports starting from the lowest indexed one and sends the first ready value. If none of the `mergeIn` ports is ready, it sends `null` value to the port.
- `RingQueue` automaton models a `size`-buffered ring queue. RingQueue not only handles the normal `enqueue` and `dequeue` functionality but also performs error signaling by dequeuing invalid data on an empty buffer in active mode and discards the extra data on a full buffer in passive mode. This implementation includes general error handling in all use cases, and it will not stall communication no matter how it is connected.

These communication channels in the original Mediator can only be defined with nondeterministic basic connections. Enhanced MedTiny expands this, extending the expressiveness of connections into a deterministic domain.

5. Case Study

Various works related to code generation from Mediator exist, with Mediator able to generate Arduino C and NuSMV [11], PRISM [15], and SystemC [16]. Since MedTiny is a subset of Mediator, code generation algorithms from MedTiny to programming and verification languages can draw from the frameworks mentioned in these related works. However, the code generation framework employed by previous works necessitates first flattening all layered automata into a single canonical automaton to eliminate the synchronization flags. While this canonical automaton approach doesn't impact the correctness of model checking, the original implementations can only translate a model into a single-threaded program, which undermines the inherent parallelism of the algorithm. More importantly, code generated from the single canonical automaton approach is difficult to read and modify manually, especially for locating and referencing symbols, because all code structure is flattened.

Having thoroughly explored the semantics of two-flag synchronization, we can now confidently convert the auto-synced automaton into similar code structures in language targets directly. This makes the generated code more comprehensible for additional manual bootstrapping and modification, which is often needed when utilizing unique features of the targeted languages or tools.

Code Example 3. Generalized Communication Channels

```
automaton <numOut:int,T:type> Duplicator(input:in T,
    dupOut[numOut]:out T) {
    transitions {
        auto-sync true -> {
            sync input;
            for oi in dupOut {
                oi.value=input.value;
            }
            sync dupOut;
        }
        active-only true -> {
            for oi in dupOut {
                oi.value=defaults. fill <T>(null);
            }
            sync dupOut;
        }
    }
}

automaton <numIn:int,T:type> Merger(mergeIn[numIn]:in T,
    output:out T) {
    transitions {
        for ii in mergeIn {
            auto-sync true-> {
                sync ii;
                output.value=ii.value;
                sync output;
            }
        }
        active-only true -> {
            output.value=defaults. fill <T>(null);
            sync output;
        }
    }
}

automaton <T:type,size:int> RingQueue(enqueue:in T,
    dequeue:out T) {
    states {
        T[size] buffer=defaults. fill <T[size]>(null);
        int 0..size-1 phead=0;
        int 0..size-1 ptail=0;
    }

    transitions {
        //enqueue if not full
        auto-sync (ptail+1) % size!=phead -> {
            sync enqueue;
            buffer[phead]=enqueue.value;
            phead=(phead+1)%size;
        }
        passive-only (ptail+1) % size==phead -> {
            sync enqueue;
        }
        //dequeue if not empty
        auto-sync ptail!=phead -> {
            dequeue.value=buffer[ptail];
            ptail=(ptail+1)%size;
            sync dequeue;
        }
        active-only ptail==phead -> {
            dequeue.value=defaults. fill <T>(null);
            sync dequeue;
        }
    }
}
```

5.1. Code Generation

We rewrite a new prototype compiler in Java, with ANTLR tool-set [17] to build front end and the StringTemplate library [18] for code generation. Even though MedTiny is simplified and fully-defined, a complete compiler implementation still represents a substantial amount of implementation work. For demonstration purposes, we selected Python, with its built-in threading and queue libraries [19] for parallel code execution, and PRISM Model Checker for model checking. After manual grammar correction and bootstrapping, the generated code was successfully executed in Python and its properties were verified in the PRISM model checker.

MedTiny has an almost one-to-one translation to the corresponding Python code, but PRISM code generation is significantly more complicated. Specifically, we have roughly overcome the three unique challenges in PRISM generation:

1) Lack of array data structure: States in arrays must be expanded and declared individually for each index. Furthermore, dynamic indexing of an array requires duplicating transitions or statements, with checks for all indices.
2) Synchronize states with labels: Synchronizations occur between transitions labeled by the connected ports. In MedTiny, the *out_port* and *in_port* identifier pair is the perfect label for PRISM transition synchronization. We also use the formula feature in PRISM to pass port values.
3) Limited formula and renaming templating capability: While PRISM provides renaming to reduce module duplication, this renaming occurs after formula replacement. In other words, PRISM can only rename the contents of formulas, but not formula names. As we've already used the formula feature for synchronization, module renaming breaks label synchronization. As a result, we avoid the renaming feature and instead generate every instance of an automaton module.

5.2. Character Counting in MapReduce

The central idea of parallelism in MapReduce is to break down large computational work into map and reduce phases. Map, reduce, and any other intermediate phases are supposed to be obvious to understand and implement. Each phase in MapReduce must be designed independently of each other, resulting in a highly scalable and highly fault-tolerant architecture. Consider a classic character counting example [20], the input string is firstly split into different independent partitions by a `driver`, and each partition is fed into corresponding `mapper`. Secondly, each `mapper` tokenizes the partition into characters and emits those character with an occurrence count of 1 to the responsible `combiner`. The `combiner` sums up the occurrence counts and then routes these individual counts to the `reducer`, which collects the character counts. Finally, the `driver` retrieves the output of `reducer` and forms the final result.

MedTiny is fully capable of describing the high-level connections and low-level behaviors of each individual component in character counting: `driver`, `mapper`, `combiner`, and `reducer`, with a helper `MergerQueue` component

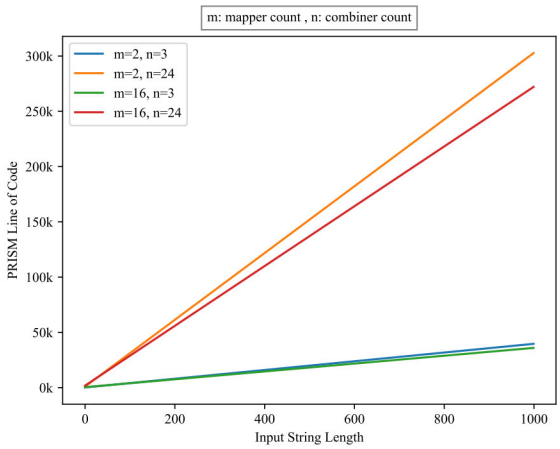

Fig. 3. PRISM Code vs Input Length on Different Configurations

to sequentially collect `mapper` outputs to the `combiner`s. Due to space limitation, we only include the fully detailed driver and mapper code in Code Example 4.

MedTiny exhibits a high efficiency in expressing the model, while still being a general purpose language. Regardless of the input size or configuration, the full model is written in just 149 lines of MedTiny code, as compared to the 144 lines of automatically generated Python Code. The length of generated PRISM code, however, scales with the configuration size and input size. We performed code generation experiments on four different mapper and combiner setups, as shown in Fig. 3. The generated line of code of PRISM model exhibits an asymptotically trend of $O(s)$, where s is the input string length, positively correlated with both mapper count and combiner count. The following two reasons mainly contribute to the PRISM code size explosion:

- PRISM does not support data arrays. While it is possible to create loop automata to reduce the use of states and formulae, this would not be a faithful representation of the original model. This problem contributes to the linear increase in code size, notably where the entire input string is communicated from `driver` to `mapper` as port values and later stored in the `mapper` buffer as states.
- PRISM's formula and module renaming features are not as flexible as Function and Automaton templates. Formula and module renaming are conflicting for the connection topology used by MapReduce: the entire `mapper` module is repeated `mapper_count` times, and the entire `MergerQueue` and `combiner` modules are repeated `combiner_count` times.

The prototype compiler-generated PRISM code still requires manual modifications to be accepted by the model checker. The fast increasing code size makes it impractical to perform manual modifications for larger setups. Therefore, we can only verify properties on a meaningful model with input string length 6, mapper count 2, and combiner count 3. This small

Code Example 4. MapReduce Character Counting in MedTiny

```
automaton mapper(text_in: in lower_alpha[
    mapper_buffer_length], map_out_arr[combiner_count]:
    out bool) {
    states {
        int 0..mapper_buffer_length+1 pointer =
            mapper_buffer_length+1;
        lower_alpha[mapper_buffer_length] text_buff = fill
            <mapper_buffer_length>('a');
    }
    transitions {
        auto-sync pointer == mapper_buffer_length+1 ->{
            sync text_in; pointer = 0;
            text_buff=text_in.value;
        }

        auto-sync pointer == mapper_buffer_length -> {
            for o in map_out_arr { o.value=0; }
            pointer = pointer + 1;
            sync map_out_arr;
        }

        for oi,o in map_out_arr {
            auto-sync text_buff[pointer] == 'a'+oi->{
                o.value=1; pointer = pointer + 1;
                sync o;
            }
        }
    }
}

automaton driver() {
    states {
        int 0..mapper_count+1 mapper_launched = 0;
        mapper m[mapper_count];
        MergerQueue<mapper_count,mapper_buffer_length>
            mcq[combiner_count];
        combiner cb[combiner_count];
        reducer r;
    }

    inits {
        for mi in m {
            internal -> m.text_in;
            for ci,c in mcq {
                mi.map_out_arr[ci] -> c.mergeIn;
            }
        }

        for ci,c in cb{
            mcq[ci].dequeue -> c.map_in;
            c.reduce_out->r.reduce_in[ci];
        }
        r.result_out -> internal;
    }
    transitions {
        auto-sync mapper_launched == mapper_count -> {
            ...
        }

        for o in prepare_input() {
            auto-sync mapper_launched<mapper_count->{
                m[mapper_launched].text_in.value=o;
                mapper_launched = mapper_launched + 1;
                sync m[mapper_launched].text_in;
            }
        }
    }
}
```

setup still represents over 450 lines of PRISM code.

6. Discussion and Future Work

While the enhanced MedTiny can express models more concisely than Mediator, our usage experience suggests for more language improvements. We propose the following MedTiny enhancements to address the most representative limitations:

Component Interface Subtyping is introduced in original Mediator but remains unused due to Mediator's inability to fully generalize components. The enhanced MedTiny, capable of defining libraries, justifies the inclusion of this feature. We also plan to introduce an interface array to enable the concise expression of, for instance, an array of ring buffers instantiated with varying sizes.

Recursive Template Metaprogramming is a potent technique that involves using templates to execute computations during compile-time[21]. MedTiny's design curtails the recursive *Function* calls to eliminate stateful runtime recursions. However, compile-time recursion does not conflict with *Function* semantics. This feature allows for the natural expression of recursive algorithms.

Clock abstraction suggested in distributed Mediator enables the expression of refined continuous time asynchrony. Every closed entity should have a unique time domain, so MedTiny does not need extra syntax other than a specialized `clock` variable available globally within each closed entity.

Probabilistic extension of Mediator is introduced in [22]. This extension allows for precise analysis and verification of systems exhibiting stochastic (random or probabilistic) behavior, which are common in real-world applications.

7. Conclusion

In this paper, we have outlined the MedTiny modeling language, a well-defined follow-up of Mediator fixing some oversights in the original design. We have enhanced the readability and writability of formal models, particularly in the domain of scalable algorithms, with the help of programming language inspired generalized array and compile-time foreach loop. Moreover, we have provided a comprehensive explanation of the two-flag communication synchronization mechanism. This understanding allows us to write generalized synchronous MedTiny libraries with convenient and reliable syntactic sugars. Our future research will focus on the continued grammar refinement and the development of superior tooling, so MedTiny is as usable as programming languages to lower the learning curve of formal methods.

Acknowledgment

This research was sponsored by the National Key R&D Program of China (2022YFB2702200), NSFC under Grant No. 62172019, and CCF-Huawei Populus Grove Fund.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Amazon.com, "Secure and resizable cloud compute – amazon EC2 – Amazon Web Services." https://aws.amazon.com/ec2/.

[3] C. Kim, "Ethereum 2.0: How it works and why it matters." https://theblockchaintest.com/uploads/resources/file-844047896485.pdf, 2020.

[4] A. Gawanmeh and A. Alomari, "Challenges in formal methods for testing and verification of cloud computing systems," *Scalable Computing: Practice and Experience*, vol. 16, no. 3, pp. 321–332, 2015. Number:3.

[5] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *Computer Performance Evaluation: Modelling Techniques and Tools: 12th International Conference, TOOLS 2002 London, UK, April 14–17, 2002 Proceedings 12*, pp. 200–204, Springer, 2002.

[6] "Information technology — Object Management Group Unified Modeling Language (OMG UML)." https://www.iso.org/standard/32624.html, 2012. ISO/IEC 19505-1:2012 (Part 1: Infrastructure) and ISO/IEC 19505-2:2012 (Part 2: Superstructure).

[7] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.

[8] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, "NuSMV 2.6 User Manual." https://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf.

[9] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—a Tool Suite for Automatic Verification of Real-Time Systems," in *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control*, (Berlin, Heidelberg), p. 232–243, Springer-Verlag, 1996.

[10] O. Badreddin, "Umple: a model-oriented programming language," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pp. 337–338, Association for Computing Machinery, 2010.

[11] Y. Li, W. Sun, and M. Sun, "Mediator: A Component-Based Modeling Language for Concurrent and Distributed Systems," *Sci. Comput. Program.*, vol. 192, jun 2020.

[12] J. Lee, "ANSI C Yacc Grammar." https://www.lysator.liu.se/c/ANSI-C-grammar-y.html, 1985. Published for the April 30, 1985 draft version of the ANSI C standard. Reposted to net.sources in 1987 by Tom Stockfisch.

[13] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[14] Oracle Corporation, "Java Platform, Standard Edition & Java Development Kit Version 17 API Specification." https://docs.oracle.com/en/java/javase/17/docs/api/index.html, 2021.

[15] W. Sun and M. Sun, "PRISM Code Generation for Verification of Mediator Models.," in *SEKE*, pp. 271–354, 2019.

[16] Q. Zhang, Y. Li, and M. Sun, "Automatic SystemC code generation of mediator model," *Computer Engineering & Science/Jisuanji Gongcheng yu Kexue*, vol. 41, no. 5, 2019.

[17] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd ed., 2013.

[18] T. Parr, "Enforcing Strict Model-View Separation in Template Engines," in *Proceedings of the 13th international conference on World Wide Web*, pp. 224–233, 2004.

[19] "Queue — a synchronized queue class." https://docs.python.org/3/library/queue.html.

[20] Talend, "MapReduce 101: What it is & how to get started." https://www.talend.com/resources/what-is-mapreduce/.

[21] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.

[22] A. Liu, S. Liu, and M. Sun, "Probabilistic Mediator: A Coalgebraic Perspective," *Journal of Logical and Algebraic Methods in Programming*, vol. 129, p. 100808, 2022.