MODEST Language Syntax for Structural Model Parameters

Rifqi Adlan Apriyadi
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
rifqiadlanapriyadi@student.utwente.nl

ABSTRACT

MODEST, a modeling language designed for stochastic timed and probabilistic systems with support for complex continuous aspects, provides users with a wide variety of tools at their disposal for the purpose of quantitative evaluations and modeling. Nevertheless, a caveat of the language is that structural model parameters (SMP) is not yet supported. SMP refers to the parameters that structurally alter the symbolic semantics of the automata generated from the parsed model. An instance of the consequence of such a feature's absence is the inability to flexibly initialize a process in a parallel composition a certain number of times with respect to a constant or model parameter. The contribution of this paper is the design of an extension to MODEST's syntax that facilitates the feature addressing this issue. The design is accompanied by a viability study that is in the form of a the design's prototype along with its performance on existing models.

Keywords

MODEST, Modeling Language, Syntax Extension, Syntactic Sugar

1. INTRODUCTION

MODEST [26] is a modeling language that is accepted by the integrated quantitative modeling and verification environment that is the MODEST Toolset [27]. It is a high-level textual modeling language with expressive programming language-like syntax to produce compact models. MODEST supports model parameters, which are constants whose values are specified by users at tool runtime. However, what it did not yet support was *structural model parameters* (SMP). For example, before this work, it was not possible that a process is called the exact same number of times as the value that is given as a model parameter.

For a set of tools meant to be used repeatedly to observe behaviours of different models with different configurations, the absence of such a feature is very troublesome for the user. Problems from this are as follows:

- Users would need to frequently open the MODEST file and manually add or delete statements for model checks with different configurations. Alternatively, they would need several files for this, which would require them to edit all files for any changes.
- Users are prone to make mistakes as they would need to precisely count the number of repeated declarations and statements written.

Listing 1 shows a more comprehensible example of this issue. It was not possible that instances of `:: Host()` in Line 59 automatically corresponds to the constant declared in Line 8. Either multiple files needed to be written for different values of `H`, or the user needed to frequently update `H` and the `Host()` process calls for different configurations of the model. If `H` is not given a value in the file, then its value depends on the specified model parameters. Checking the model via the command line with arguments `modest mcsta beb.modest -E "H=3"` utilizes model parameters to give `H` the value 3. In either case, the syntax extension presented in this paper allows for the number of `:: Host()` instances to correspond automatically to `H`.

```
8   const int H = 3;  // number of hosts (must
                         correspond to the number
                         of Host() instantiations
                         in the global composition)
...  ...
59  par { :: Clock() :: Host() :: Host() ::Host() }
```

Listing 1: Part of an Existing Model: `beb.modest`

Several research questions are present in this paper to assist in the design of an extension that provides MODEST with this missing functionality. They are as follows:

- **Research Question 1**: What design would be appropriate for a syntax extension that facilitates SMP for the usage by those who frequently use the modeling language?
- **Research Question 2**: Could the designed syntax structure be integrated into MODEST? If so, how?
- **Research Question 3**: What kind of models can be improved from the addition of this syntax? What are the limitations of the syntax extension?

The above research questions reflect the approach that the research this paper details took. Firstly, a design was devised based on a set of requirements. Secondly, the chosen design was implemented to provide a proof of concept of how it could work in the language and give insight towards its viability. Finally, existing models were examined and modified to highlight how the extension aids in their improvement along with any possible impediments.

This paper records the research done in the design process of the syntax extension. Section 2 discusses the background information of the domain that is frequently referred to by the paper. The chapter following then presents the chosen requirements that the design ought to satisfy.

35^{th} *Twente Student Conference on IT* Jul. 2^{nd}, 2021, Enschede, The Netherlands.

The design itself and methodologies on its formation are examined in Section 4. Sections 5 and 6 convey the viability of the design and the conclusions of the research done, respectively.

2. BACKGROUND

2.1 Modeling Languages

The development of modeling languages and their tools is fueled by embedded software [37]. Embedded software's key role is the interaction with the physical world. Other than its functional properties, nonfunctional ones — such as resource usage and robustness — are therefore just as relevant. Failure in recognizing this principle can be costly. The ignorance of error rates of implantable cardioverter-defibrillators, for example, may even be lethal [20].

Model checking [15] is a technique for determining if a system's finite-state model meets a set of requirements. This is most often associated with the previously mentioned embedded systems where the specification includes both liveness and safety criteria [5]. Given these definitions, the use of modeling languages gives their users their much-needed insight into quantitative properties that are of great consequence for such frequently complex embedded systems.

2.2 MODEST

2.2.1 More on MODEST

As mentioned in Section 1, MODEST is one of the aforementioned modeling languages. The MODEST Toolset is based on a single overarching semantic model: networks of stochastic hybrid automata (NSHA) [13, 18], which are collections of asynchronous automata that can communicate via shared actions and global variables.

In a variety of applications, the absence of randomness makes accurate modeling and analysis difficult, requiring a *probabilistic system* [39]. On the other hand, discrete state changes and continually varying quantities are combined in a *hybrid system* [31, 6]. MODEST shines in not only providing the combination of both systems, but also in the location where randomization is introduced. Probability distributions across mode leaps might be used to replace deterministic mode transitions. In some cases, stochastic differential equations can be used to substitute differential equations that describe the development of continuous variables within a mode.

MODEST's semantics are split into two parts: a *symbolic* semantics based on NSHA, and a *concrete* semantics based on nondeterministic labelled Markov processes (NLMP) [17]. The former entails converting a process's process calculus constructs into an NSHA. The components of the model that aren't connected to process-calculus-based definitions, like model variables or assignments, will be kept in symbolic form. The latter provides meaning those definitions and also determines the MODEST specification's continuous behavior.

Writing multiple MODEST files or changing one file every time a different configuration is required is not ideal, as mentioned in Section 1. Nonetheless, a possible workaround for this feature — albeit only solving process calls in `par` constructs — was to do recursive calls to some processes which aims to achieve the same effect. Listing 2 shows an example that is identical to the code snippet in Listing 1. Evidently, it is still cumbersome for the user to create such processes every time a scalable construct is required. As seen in Listing 2, the process `ParHost` specifically calls the `Host` process only, requiring similar processes for the scaling calls of other processes, increasing the chances of human error.

```
 8  const int H = 3;
...  ...
59  process ParHost(int i) {
60      alt {
61          :: when(i == 1) Host()
62          :: par { :: Host() :: ParHost(i - 1) }
63      }
64  }
65  par { :: Clock() :: ParHost(H) }
```

Listing 2: A Workaround for Scaling Process Calling

Moreover, it is often desirable to be able to process a model with different configurations consecutively. With SMP, it would be possible check the model in Listing 1 with different configurations by model parameterizing `H` and using the command `modest mcsta beb.modest -E "H=3" -E "H=4"`. This would consecutively check the model each with the different model parameter for `H` provided.

2.2.2 Structural Model Parameters

With model parameters, constants in the model can be specified when the tool is requested via the command line. It is able to change concrete semantics via its values, as with constants and variables. SMP, on the other hand, are parameters that could change the structure of the symbolic automata of the model. Therefore, SMP refers to parameters that, when changed, affects the structure of the generated NSHA of the symbolic semantics. In this context, this is done by the number of alternatives created, or by the determination of the actions done.

For the model in Listing 1, MODEST generates three separate automata for each `Host()` process. With the addition of SMP, however, the number of times `Host()` is written can be synchronized corresponding to some value. Altering that value also alters the number of times `Host()` is called in the `par` block, causing the structure of the automata to change as well, namely because of the number of `Host()` automata present. Additionally, SMP also allows for the denotation of such alterations in a much more intuitive manner, unlike the workaround previously presented. Other constructs that mainly suffered from this problem include `alt`, `do`, `palt`, `property`, and `action` (explained in Section 2.2.3).

2.2.3 The MODEST Language

A MODEST specification consists of a sequence of *declarations* and a process *behaviour*. Declarations are where users declare the existence of constants, variables, properties, processes, exceptions, and actions. A process behaviour is a statement users write that utilizes the declarations made to behave in the ways the users intend. The partial grammar relevant for this paper is as follows:

$$
\begin{aligned}
dcl ::= {} & \mathsf{property}\ id\ =\ PropExp; \mid \\
& [\mathsf{patient} \mid \mathsf{impatient} \mid \mathsf{binary} \mid \mathsf{broadcast}] \\
& \mathsf{action}\ id\ (,\ id)*; \mid \\
& \ldots \\
P ::= {} & \mathsf{alt}\ \{ :: P_1 \ldots :: P_k \} \mid \mathsf{par}\ \{ :: P_1 \ldots :: P_k \} \mid \\
& \mathsf{do}\ \{ :: P_1 \ldots :: P_k \} \mid \mathsf{extend}\ \{act_1 \ldots act_k\}\ \{P\} \mid \\
& \mathsf{relabel}\ \{act_1 \ldots act_k\}\ \mathsf{by}\ \{act_1 \ldots act_k\}\ \{P\} \mid \\
& \mathsf{palt}\ \{ :w_1 \colon asgn_1;\ P_1\ \ \ldots\ \ :w_k \colon asgn_k;\ P_k \} \mid \\
& \ldots
\end{aligned}
$$

dcl depicts a declaration and P represents a process. *PropExp* is the expression exclusive for properties. *act* refers to actions, in the form of their identifiers specified in a process' declarations. *id* signifies a string used as an identifier. w represents an arithmetic expression. Finally, $asgn_i$ are type-consistent assignments of the form

$$\{= x_i = e_i, ..., x_k = e_k =\}$$

where x constitutes a variable and e an expression. A more detailed description of the language and grammar is mentioned here [26].

As mentioned in previous sections, the constructs relevant in this paper are those with keywords `alt`, `par`, `do`, `palt`, `property`, and `action`. Their purposes are as follows:

- `alt`: the specification of nondeterministic choices between different behaviours (alternatives).
- `par`: the specification of behaviours (alternatives) to execute concurrently, possibly synchronising on certain actions.
- `do`: a variant of `alt` that restarts from the beginning when the chosen behaviours (alternatives) terminate successfully.
- `palt`: assigns weights to process behaviours, resulting in a probabilistic choice (probabilistic alternatives).
- `property`: the declaration of a property that the model specification should to satisfy.
- `action`: performs an action and then terminates. Could be referenced to be relabeled, extended, or restricted (more on [26]).

2.2.4 The Compiler

The front-end of the compiler consists of a scanner, lexer, and parser and create an intermediate representation of the model that is utilized in the next parts of the compiler. The scanner reads the string that represents the model, and the lexer converts them into tokens that the recursive descent parser [7] uses to create the intermediate representation. The intermediate representation is represented by an abstract syntax tree called `ProcessSymbol` (PS).

The parser utilizes the PS to store the declarations and behaviours that it detects that conform to a grammar. Each type of declaration has its own class and its own list in the PS object. Each type of behaviour also has its own class that inherits the abstract class `Construct`. In other words, a behaviour is represented by a `Construct` in a PS.

After the parsing step finishes, the resulting PS is passed onto the next part of the compiler which includes optimization — simplifying the PS as much as it can. The optimization includes replacing expressions without variables with their resulting values and assigning the corresponding values to the parsed constant symbols `ConstantSymbol` in expressions. Additionally, a *binding* is also done that binds model parameters declared at tool runtime to their corresponding constants.

After all optimizations and transformations of the PS are done, the intermediate representation is then transformed into an NSHA. Each behaviour has its own formal definition of its behavioural semantics in the automata, which reflect the behaviour of these graphs accordingly.

2.3 Related Work

Promela [32] has a syntax whose semantics behave very identical to that of the functionality this paper discusses. Its `for` keyword works similarly to the feature this paper intends to facilitate. However, PRISM [35] and JANI [10] still lack this feature. This is evident in that problematic PRISM models from the Quantitative Verification Benchmark Set (QVBS) [30] require multiple files for different configurations in its original language. Similarly, MODEST models translated into JANI also result in multiple files. A thorough investigation into LOTOS', LNT's [21], and mCRL2's [23] documentation also suggests the absence such a feature. UPPAAL [9] possesses a partial availability of the feature. While it has the conventional and yet convenient `for` keyword which acts similar to conventional for-loops, it lacks functionalities that allow for scalable system declarations. To declare their *templates* (UPPAAL's version of MODEST's processes) and properties, users still need to declare each one independently.

On parameterization, recent work [29] has been done with MODEST on *Parametric Probabilistic Timed Automata* (pPTA) — extending PTA [36, 22] — where the probabilities of some events are unknown and are expressed as polynomials over a finite set of parameters. Another study [14] examines how to describe configurable probabilistic systems using families of Markov chains (MCs) [33]. It introduces an abstraction-refinement scheme over the MDP representation. The abstraction loses track of which member of the family the MDP affects. Every attainable state of a family member has a single representative in the resulting quotient MDP. The possibility of different topologies among family members extends beyond the class of parametric MCs examined in parameter synthesis [40] and model repair [8].

3. REQUIREMENTS

This section discusses the properties that the extension should ideally possess.

3.1 Flexibility

The SMP problem exists for several constructs in the MODEST language. Therefore, the ideal extension is one whose syntax covers all constructs that possess the SMP issue. It is not desirable to have solutions that are rigid in nature to their specific constructs or use cases. Preferably, the syntax ought to be flexible and could determine its semantics depending on when and where the syntax is used. This is such that "it avoids distinguishing between different problem domain concepts" [1].

3.2 Convenience

The syntax also ought to posses properties that contribute in convenience. It should be that using the new feature is not incommodious, which is exactly what the syntax extension aims to prevent. Additionally, the syntax should also be designed in an comprehensible fashion that contributes to readability. Readability allows for the convenience of users to understand the code in a more familiar way, and thus, more understandable way [2].

3.3 Conformity

Different languages have different syntax styles. Therefore it is also important that the designed syntax conforms to MODEST's language style. This includes aspects such as the keywords and symbols used, the structure, and the notation of the expressions.

4. DEVISED DESIGN

4.1 Methodology

The designing step of the extension consisted of the drafting of a few grammar extension options and review of the draft designs from the relevant parties. Design drafts were compiled with their own syntax structure, keywords used, and each with their own perks. Albeit, each of the different

drafts were not necessarily mutually exclusive. The combination of different designs were used in the final design. Some were also used simply as feedback on how the final design should be. Different designs were separated into different drafts such that the discussions of the properties of each could be done separately.

4.2 The Design

The finalized syntax design — extending the grammar from Section 2.2.3 and only including rules that have been altered — is as follows:

$$
\begin{aligned}
dcl &::= \mathsf{for}\ FH\ \{FD_1 \dots FD_k\} \mid \\
&\quad \dots \\
FD &::= \mathsf{property}\ id\,[\texttt{[}Expr\texttt{]}]\ =\ PropExp \mid \\
&\quad [\mathsf{patient} \mid \mathsf{impatient} \mid \mathsf{binary} \mid \mathsf{broadcast}] \\
&\quad \mathsf{action}\ id\,[\texttt{[}Expr\texttt{]}]\ (,\ id\,[\texttt{[}Expr\texttt{]}])*; \\
P &::= \mathsf{alt}\,[FH]\,\{\,alt_1 \dots alt_k\,\} \mid \mathsf{do}\,[FH]\,\{\,alt_1 \dots alt_k\,\} \mid \\
&\quad \mathsf{par}\,[FH]\,\{\,alt_1 \dots alt_k\,\} \mid \mathsf{for}\ FH\ \{\,P\,\} \mid \\
&\quad \mathsf{palt}\,[FH]\,\{\,walt_1 \dots walt_k\,\} \mid \\
&\quad \dots \\
FH &::= (id : Const\ ..\ Const) \\
alt &::= \ ::P \mid \mathsf{for}\ FH\ \{\,alt_1 \dots alt_k\,\} \\
walt &::= \ :w\text{: } asgn;\ P \mid \mathsf{for}\ FH\ \{\,walt_1 \dots walt_k\,\}
\end{aligned}
$$

The main theme of the design is the use of the keyword `for` for areas in the language that lack an iterative construct and, more primarily an SMP feature. Such constructs include the previously mentioned `par`, `alt`, `do`, `palt` constructs, and property and action declarations. Additionally, the `for` keyword is also added as a process behaviour.

An example of an existing model can be seen in Listing 3. It highlights the SMP problem in the declaration of properties and statements of the `alt`, `palt`, and `par` blocks. Using the designed syntax extension, the model can be rewritten more concisely, as depicted in Listing 4.

A common structure that the grammar now has is in the inclusion of the FH (*ForHeader*) rule. An example of a usage of this rule is in the form of `(i : L..U)`. With this FH, everything within the scope of the construct that the FH is declared with repeats $\texttt{U} - \texttt{L}$ times, with `i` incrementally increasing from `L` (inclusive) to `U` (exclusive) in each repetition. In Listing 4, this can be seen in Lines 12, 17, 18, and 26.

Furthermore, syntax extensions are added to the `par`, `alt`, `do`, and `palt` constructs wherein the usage of these keywords can be followed immediately by an FH. These forms are referred to as their *extended forms*. This is designed such that if an FH `fh` follows immediately after the keyword `k` of one of the mentioned constructs with the body (contents between curly braces) `b`, the construct should represent exactly as if what was written was the normal construct `k` whose contents are only a singular for-loop whose FH is `fh` which, in turn, has the body `b`. In other words, the extended forms are shortcuts for the constructs they extend whose only alternative or behaviour is a single `for` block. In Listing 4, Line 26 can be replaced with `par (i : 0..SIM_DICE) {:: SimulatedDie(0)}`. Likewise, the `palt` block in Lines 18–19 can be simplified to `palt (j : 0..2) {:1: {= state = i*2+j =}}`. A similar simplification cannot be done to the `alt` block starting from Line 17 from the same model, however, as the block contains other alternatives not to be included in the `for` block, namely the alternatives starting from after Line 19.

```
 7 property Term = Pmin(<> (done == 2)) == 1;
 8 property P2  = Pmax(<> (done == 2 && value == 0));
 9 property P3  = Pmax(<> (done == 2 && value == 1));
   ... // Similar properties until P12
   ...
20 process SimulatedDie(int(0..7) state) {
21 alt {
22 ::when(state == 0) palt { :1: {= state = 1 =}
23   :1: {= state = 2 =} }
24 ::when(state == 1) palt { :1: {= state = 3 =}
25   :1: {= state = 4 =} }
26::when(state == 2) palt { :1: {= state = 5 =}
27   :1: {= state = 6 =} }
   ... // Other alternatives (rest of alt)
32 } ; SimulatedDie(state) }
33
34 par { :: SimulatedDie(0) :: SimulatedDie(0) }
```

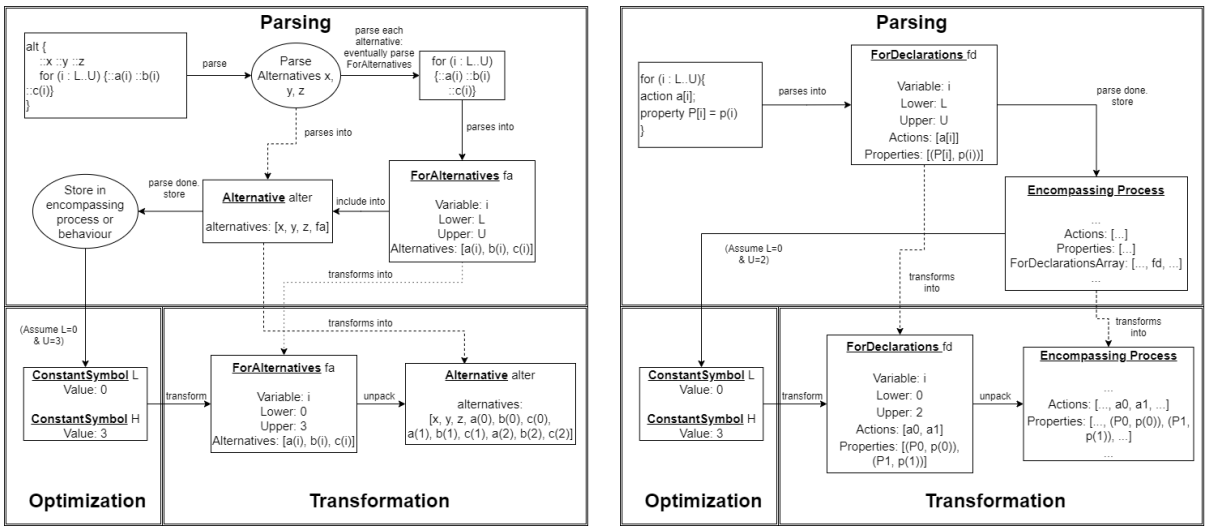
Listing 3: Part of an Existing Model: `knuth-dice-sum.modest`

```
 4  const int PROPS = 12;
 5  const int PALT_REPS = 3;
 6  const int SIM_DICE = 2;
    ...
11  property Term = Pmin(<> (done == 2)) == 1;
12  for (i : 2..(PROPS+1)){
13      property P(i) = Pmax(<>(done==2 && value==i-2));
14  }
15
16  process SimulatedDie(int(0..7) state) {
17      alt { for (i : 0..PALT_REPS) {
18     ::when(state == i) palt { for (j : 0..2) {
19     :1: {= state = i*2+j =} } }
       ... // Other alternatives (rest of alt)
24    } ; SimulatedDie(state) }
25
26  par { for (i : 0..SIM_DICE) { :: SimulatedDie(0) }}
```

Listing 4: Part of an Existing Model Improved by the Design: `knuth-dice-sum.modest`

Moreover, it is apparent that the grammar rules for property and action declaration and action reference are distinct from the rules of the other constructs. This is a consequence from how all properties and actions in a model require unique identifiers. Therefore, for property and action declarations, their names in a `for` construct ought to include the iterating values to avoid identical property identifiers. This can be done by including an expression, which utilizes the iterator variable, in between square brackets suffixing the properties' or actions' names. The strategy applied to this design is that properties with similar conditions often have similar names, such as the sequence starting with `P1`, `P2`, and so on. Such identifications are achievable through this design. An example of this is seen in Lines 12–14 in Listing 4.

Additionally, suffixing an expression to an action reference in a process behaviour is interchangeable to referring to the action with an identifier equal to the concatenation of the action string with the value of the suffixing expression. This denotation is most useful when done in a `for` construct, but need not be. For example, the behaviour `for(i:1..4){go[i]!}` is equivalent to the behaviour `go1!;go2!;go3!`.



(a) ForAlternatives (b) ForDeclarations

Figure 1: The States the Placeholders Undergo in Different Phases

5. DESIGN VIABILITY

This section examines the viability of the transpired design. A discussion of the proof-of-concept of the syntax extension is first done, along with its validation. Further exploration of the benefits and limitations of the syntax extension on different models will also be discussed in the subsequent subsection. Finally, the requirements from Section 3 will be reflected upon based on the designed syntax extension.

5.1 Feasibility Study

5.1.1 Implementation

A feasibility study was done by doing direct implementation into the source code and extending it to include the designed syntax. However, due to the limitation of the research's duration, the prototype only included the extensions for constructs with alternatives as their body contents, iterative property and action declarations, and suffixed action references.

The design was implemented by means of *syntactic sugar* [38]. Briefly, syntactic sugaring is the provision of syntax forms that allows programmers to write statements in simpler forms, which would be transformed into their most basic forms later in the compilation process. A popular example of a syntactic sugar is `i += 2`, which is transformed into `i = i + 2` later in the compilation process. In terms of the examples of Listings 3 and 4, the statements in lines 18–19 of the latter are internally transformed into the statements in line 16 of the former before continuing to the NSHA generation stage of the compiler.

New structures — `ForAlternatives` and `ForDeclarations`, which are referred to as *placeholders* — were written. The former in itself is fundamentally a behaviour which could only exist directly within `alt`, `par`, or `do` constructs, while the latter is seen as a new type of declaration. Placeholders contain information on the identifier of the iterator variable, the lower and upper bounds of the FH range, and the contents of the accompanying block. These blocks each opens a new scope and also adds the iterator variable to it. These placeholders will be crucial in the transformation in a later stage of the compilation. Additionally, the class that represents an action reference was extended to be able to store an expression that constitutes the optional suffix at the end of an action reference identifier.

After the optimization stage, an additional inspection — dubbed the *transformation* stage — is done to scan the PS for placeholders and action references such that they can be transformed accordingly. Since expressions without variables have been replaced with their results (e.g. $2 + 2$ is replaced with 4), usages of constants have been given their corresponding values, and model parameters have been bound, their values can be directly utilized in this step. These values are to be used suitably to transform placeholders and suffixed action references into their intended forms.

To achieve this, a *preorder traversal* [34] is done on the PS. Processes and constructs *unpack* any placeholders they encompass. After replacing the lower and upper bounds with their corresponding numerical values, `Unpack()` repeats the body contents with respect to the bounds while replacing any usages of the iterator within expressions with increasing values in each repetition. Furthermore, as expressions suffixing action references have been optimized into numeric values, suffixed action references are transformed into their normal forms, concatenating their identifiers with their suffixing numeric values, eliminating their suffixes.

Figure 1 shows the states that each placeholder type undergo in the compilation process from the parsing to the transformation phase. It highlights and expresses only states relevant to the placeholder in the different stages. Action references are processed similar to that of their declarations as shown in Figure 1b, with the only difference being that rather than being stored in their encompassing `ForDeclarations` and processes, they are stored within their encompassing behaviour instead.

5.1.2 Validation

Validation of the prototype was done by comparing the results of the use of tools on models that use the designed syntax with the results of using the same tools on their original file versions (without using the syntax extension). Needless to say, the generation of the former was done with the MODEST environment on which the prototype was added, while the latter was from the version left untouched. Out of the models available in QVBS combined with the ones that are provided by the MODEST Toolset as sample models, a few models were chosen that exhibit different characteristics that are relevant to the implemen-

tation done, namely those who possess the SMP problem in alternatives constructs (`par`, `alt`, and `do`), property declarations, and action declarations or references.

The comparisons were based on two MODEST Toolset tools:

- **mcsta** (check): a fast explicit-state model checker that employs secondary storage to avoid state space explosion [28] and implements state-of-the-art MA-specific algorithms [12].
- **modes** (simulate) [11]: a statistical model checker with automated rare event simulation capabilities.
- **mosta** (export-to-dot): visualises the symbolic semantics of models by exporting it to a DOT graph [3].

The models used for this validation — chosen from the criteria mentioned — were `beb.modest`, `knuth-dice-sum.modest`, `dpm.v2.modest` (Partially portrayed in Listings 5 and 6), and `dining-cryptographers-4.modest` (similar to Listings 7 and 8). For each, the outputs of each tool listed above on the original and improved versions of the model were compared.

mcsta may be the most important tool from the list. It returns precise numbers in regards to a model's states, transitions, branches, and probability ranges and errors of each of its properties. Comparisons based on it were done by checking whether the numbers returned by the tool for models not using SMP are identical to their versions that do and are exactly the same in relevant areas, demonstrating that semantics are not tampered in the compilation process as a consequence of the prototype. Values that vary in each execution of the tool, such as those that relate to the performance of the execution, are among the values that are irrelevant to match in the comparisons.

This is the similar case for **modes**, but evidently for different values. For comparisons with regards to this tool, the number of runs done and the final estimated probability for each property differ in each simulation. Values related to the error bounds of properties, however, remain static throughout different simulations.

mosta returns the textual form of the automata generated by the MODEST Toolset in the DOT language. The tool allows for the confirmation that the resulting symbolic semantics of the improved versions of models compiled by the prototype do not differ from their unimproved versions compiled by the the unmodified MODEST Toolset. Therefore, given the outputs of the tool for the two different versions of a model, it must be the case that the two automata are *isomorphic* [19]. Python packages NetworkX [25] and PyGraphviz [24] were used in conjunction to test the isomorphism of the two versions of each model. The latter is required to read the automata and the former determines their isomorphism.

The comparisons were satisfactory. Outputs of **mcsta** were identical for relevant values, such as in all forms of probability, and in the number of states, transitions, and branches. To no surprise, this is the case for **modes** simulations as well. The resulting (ε, δ) values [16, 4] of the error bounds prove to be identical in the simulations of both versions of each model. Finally, the use of **mosta** and the two Python packages proves that the automata generated by the prototype for the improved versions of the chosen models are isomorphic to that of their original versions that were processed by the unmodified MODEST Toolset. This is also apparent through the visualization of each graph via the utilization of the graph visualization tool Graphviz [3].

5.2 Improved Models

This subsection discusses the models improved by the designed syntax.

Out of the 24 distinct MODEST models from the combination of MODEST models from QVBS and MODEST Toolset's sample models, 14 possess the SMP problem. These 14 models could be improved by the devised design, albeit only to some degree for some. From these 14 improvable models, 4 are formulated in this paper that are relevant enough in terms of their different characteristics and this subsection discusses these characteristics and how they are improved by the design.

Listing 1 is a relatively simple model to improve as it only needs to synchronize the number of `Host()` process calls with the constant `H`. The multiple `:: Host()` alternatives in Line 59 can be replaced with `for (i : 0..H) {:: Host()}` and the NSHA generated by the compiler for both forms would be identical.

`knuth-dice-sum.modest` of Listings 3 and 4 are noticeably also of the improvable models. Though it is relatively simpler than some of the other models, it simultaneously highlights the use of the syntax extension on multiple distinct constructs. The model can be improved by using the syntax extension on its property declarations, alternatives, and weighted alternatives in different parts of the model, as seen in Listing 4.

`dpm.v2.modest` is unique in its relevance here as the repeated alternatives are dependant on integer variables. One alternative in each of two separate `alt` blocks corresponds to exactly one bounded integer variable. This is presented in Listing 5.

```
 2 const int C; // queue size (model parameter)
... ...
 5 int(0..C) items1 = 0;
 6 int(0..C) items2 = 0;
... ... // Similar "items" variables until "items10"
... ...
42 :: alt {
43   :: when(N>=1 && items1>0) tau {= t=1, items1-- =}
44   :: when(N>=2 && items2>0) tau {= t=2, items2-- =}
...  ... // Until items10
53 };
... ...
79 alt {
80   :: when(N >= 1) rate(0.1 * 1 + 0.4) tau;
        when(items1 < C) {= items1++ =}
81   :: when(N >= 2) rate(0.1 * 2 + 0.4) tau;
        when(items2 < C) {= items2++ =}
...  ... // Until items10
90 }; ...
```

Listing 5: Part of an Existing Model: `dpm.v2.modest`

Though the syntax extension designed in this paper does not facilitate the instantiation of variables, MODEST provides the ability to create arrays of values that achieve the same effect. An array of bounded integers `itemss` can be declared to substitute all `items` variables in Listing 5. Furthermore, its values can be read and written in the same way by utilizing the iterator variable for the indices of the array. This allows the previously mentioned alternatives to be written using the syntax extension. This is shown in Listing 6. Additionally, this gives the user the ability to change the number of elements that are to be present in `itemss` by means of simply changing a single constant or giving a different model parameter.

```
 2 const int C; // queue size (model parameter)
... ...
 5 const int N_ITEMSS = 10;
 6 int(0..C)[] itemss = array(i, N_ITEMSS, 0);
... ...
34 :: alt (i : 0..N_ITEMSS) {
35   :: when(N >= i+1 && itemss[i] > 0)
        tau {= t = i+1, itemss[i]-- =}
36 };
... ...
62 alt (i : 0..N_ITEMSS) {
63   :: when(N >= i+1) rate(0.1 * (i+1) + 0.4) tau;
        when(itemss[i] < C) {= itemss[i]++ =}
64 }; ...
```

Listing 6: Part of an Existing Model Improved by the Design: `dpm.v2.modest`

`dining-cryptographers-7.modest`, depicted in Listing 7, is an example of a model that has the SMP issue that is dependant on action declarations and references. It is also an example of a model that requires multiple files that each represent a different configuration of the model (from `dining-cryptographers-3.modest` to `...-7.modest`). Each file variant declares different numbers of `show_heads` and `show_tails` action pairs along with their references in their process behaviours represented by their respective `par` blocks (Line 63 of Listing 7 for `dining-cryptographers-7.modest`).

```
 6 action show_heads, show_tails, see_heads, see_tails;
 7 action show_heads1, show_tails1;
... ... // similar action declarations
13 action show_heads7, show_tails7;
... ...
63 par { :: Payment()
64 :: relabel {show_heads, show_tails,
        see_heads, see_tails}
65    by {show_heads1, show_tails1,
        show_heads7, show_tails7}
66    Cryptographer(1)
67 :: relabel {show_heads, show_tails,
        see_heads, see_tails}
68   by {show_heads2, show_tails2,
       show_heads1, show_tails1}
69   Cryptographer(2)
... ... // Similar alternatives until "Cryptographer(7)"
85 }
```

Listing 7: Part of an Existing Model: `dining-cryptographers-7.modest`

Listing 8 shows `dining-cryptographers-7.modest` improved using the designed syntax. The constant `N` dictates the number of `show_heads` and `show_tails` action pairs declared in the `for` block on Line 7. Similarly, the alternatives of the `par` block on Line 59 — along with the actions referred to in each — are dependant on that constant as well. Relevant action references are written with suffixes that reflect the action reference pattern shown by the action references made in the `by` blocks of Lines 65 and 68 of Listing 7. Due to the modulo pattern of action references, the numbers of the declared actions were changed to start from 0. Besides the fact that the model has been greatly simplified, the improvement also eliminates the need to have multiple files for different configurations. If `N` is made to be a model parameter, then different configurations of the model could be experimented only by giving the model parameter `N` different values. `dining-cryptographers.modest` would then be the only file required for the model.

```
 5 const int N = 7;
 6 action show_heads, show_tails, see_heads, see_tails;
 7 for (i : 0..N) {
 8   action show_heads[i], show_tails[i];
 9 }
... ...
59 par { :: Payment()
60   for (i : 0..N) {
61       :: relabel {show_heads, show_tails,
             see_heads, see_tails}
62          by {show_heads[i], show_tails[i],
             show_heads[(i+N-1)%N], show_tails[(i+N-1)%N]}
62          Cryptographer(i+1)
63   }
64 }
```

Listing 8: Part of an Existing Model Improved by the Design: `dining-cryptographers-7.modest`

Though the devised design could improve all 14 of the previously mentioned improvable models, it is not without difficulty for some. Occasionally, utilizing the syntax extension requires more than only replacing repetitive declarations or statements with the syntax extension based on their patterns. This is as the case with `dpm.v2.modest` of Listings 5 and 6, where an array of bounded integers are required to replace bounded integer declarations. Another example would be of the model `ftwc.v3.modest`, which uses a great number of different actions that describe their purposes clearly through their identifiers. Replacing their declarations and references via their suffixed versions, however, would require most of them to have common identifiers with only distinct numbers at the end. Though this shrinks many parts of the file, the intuition that came with the original identifiers is sacrificed.

5.3 Reflection on Requirements

This section reflects the syntax extension design denoted in Section 4.2 with regards to its requirements specified in Section 3.

5.3.1 Flexibility

The `for` keyword is used for not only for the repetition of alternatives, but also for probabilistic alternatives, the declaration of properties, and the declaration and references of actions. Its flexibility lies in that though it signifies different denotations with the use of the different constructs, `for` is one keyword used to solve the SMP problem in them all, providing a common construct for the problem domain even in different areas.

5.3.2 Convenience

The keyword `for` is chosen for its common use across many different languages. In other words, it is chosen for its familiarity. For example, reading a MODEST model which uses the `for` keyword, through familiarity, it could be expected that the body of the `for` construct be repeated corresponding to the FH associated to it, which is the case for many popular languages. This would especially be convenient for new users.

Furthermore, the extended forms of the constructs `par`, `alt`, `do`, and `palt` are included in the syntax extension to further make it more convenient to write and read models that utilize the syntax extension. Though one of the primary objectives of this syntax extension is to avoid the nuisance of the writing and reading SMPs, it might not serve this purpose if users would need to open a scope (e.g. `alt {...}`) to only have another scope `for (...) {...}` written within. This is especially the case with programmers' tendencies to use different indentations for different scopes

and to have one line dedicated to close scopes, making models and programs more bothersome to read and write. In this regard, the extended forms offer a more convenient alternative in writing such constructs that reduces the amount of scopes that needs to be written and the lines that will be used for them.

Lastly, the syntax extension eliminates the nuisance of having multiple files or requiring to frequently edit files for different experiment configurations. The convenience presented by the syntax is to the point that only at most a few values need to be changed in the model file or in the command line to experiment the model with a different configurations.

5.3.3 Conformity

The extension is designed to have a similar style as other existing constructs in the language. The keyword and use of braces follow the similar styles to the already existing constructs in the language (e.g. `alt {...}`).

However, the use of FH is a new addition to the language which currently only serve the purpose of this syntax extension, specifically the `for` construct along with the extended forms of `par`, `alt`, `do`, and `palt`. Nevertheless, its range specification uses a familiar syntax to that of bounded values (e.g. `int(0..1)`).

6. CONCLUSIONS

The absence of SMP proves to be a major inconvenience in the area of modeling languages. This is especially the case for models that need to be examined repeatedly with different configurations. A feature that allows for SMP does not only greatly reduce the tediousness of writing models, but would also allow the user to do more rapid experiments via the utilization of the model parameters feature.

A fitting syntax for SMP that answers the first question of this research is one that possesses appropriate properties in addition to facilitating the SMP feature. The keyword `for` is flexible in that it is used to facilitate SMP in different varieties of constructs. The familiarity of the this keyword as well as the addition of extended forms for certain constructs contribute to the design's convenience. This convenience is further enhanced by the fact that multiple files are no longer required for different configurations of a model. The conformity of the language is also upheld through the use and choice of scope delimiters, body contents, and the expression of ranges within the novel constructs.

The second question this research investigates is the practicality of the designed syntax. The implementation of a prototype to substantiate the feasibility of the design indicates that SMP is integrable into MODEST with the accompanying syntax. The success of the prototype with syntactic sugaring suggests that such a method is a viable approach. The novel construct should essentially repeat its contents corresponding to the given range and substitutes the placeholder with the resulting repetitions.

The compilation and examination of the compiled models indicate that the models improvable by the syntax extension that would answer the third research question are those whose behaviours, alternatives, weighted alternatives, properties, or actions are repeated and each depend on a systematic choice of variables can be improved by using arrays of said variables, which can then be utilized by the iterator variable acting as their indices. Additionally, repeated statements whose differences are only a pattern of values denoted in their expressions can be revamped by using the new constructs and substitute the values with a formula that utilizes the iterator variable. Finally, models with identical action or property identifiers could be conveniently improved by iteratively declaring them.

Future work could further the development brought from the research detailed in this paper by investigating how actions could be placed within arrays for a more robust declaration or usage of actions as opposed to the concatenation of identifiers with numeric values. This would be all the better if the action identifiers need not be identical even if they are stored in the same array. Alternatively, exploration could be done to incorporate the syntax extension further than a syntactic sugar. The SMP feature could be more resilient if it could be integrated into the semantics of the model. This is such that aspects of the the designed syntax would not be bound to only constant values, but could utilize variables as well

7. REFERENCES

[1] Good syntax. `https://wiki.c2.com/?GoodSyntax`, journal=wiki.c2.com, year=2006, Nov.

[2] Readability | what makes a good programming language? `https://www.informit.com/articles/article.aspx?p=661370&seqNum=4`, Oct 2006.

[3] The DOT Language. `https://graphviz.org/doc/info/lang.html`, Jun 2021.

[4] A. P. Adiredja. The pancake story and the epsilon–delta definition. *PRIMUS*, pages 1–16, 2019.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[6] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.

[7] M. Barash. Recursive descent parsing for grammars with contexts. In *SOFSEM 2013 student research forum Špindleruv Mlỳn, Czech Republic, 26-31 January*, pages 10–21, 2013.

[8] E. Bartocci, R. Grosu, P. Katsaros, C. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 326–340. Springer, 2011.

[9] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. 2006.

[10] C. E. Budde, C. Dehnert, E. M. Hahn, A. Hartmanns, S. Junges, and A. Turrini. JANI: quantitative model and tool interaction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–168. Springer, 2017.

[11] C. E. Budde, P. R. D'Argenio, A. Hartmanns, and S. Sedwards. A statistical model checker for nondeterminism and rare events. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 340–358. Springer, 2018.

[12] Y. Butkova, A. Hartmanns, and H. Hermanns. A Modest approach to modelling and checking Markov automata. In *International Conference on Quantitative Evaluation of Systems*, pages 52–69. Springer, 2019.

[13] C. G. Cassandras and J. Lygeros. *Stochastic hybrid systems*. CRC Press, 2018.

[14] M. Češka, N. Jansen, S. Junges, and J.-P. Katoen. Shepherding hordes of Markov chains. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 172–190. Springer, 2019.

[15] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[16] B. Cornu. Limits. In *Advanced mathematical thinking*, pages 153–166. Springer, 2002.

[17] P. R. D'Argenio, N. Wolovick, P. S. Terraf, and P. Celayes. Nondeterministic labeled Markov processes: Bisimulations and logical characterization. In *2009 Sixth International Conference on the Quantitative Evaluation of Systems*, pages 11–20. IEEE, 2009.

[18] A. David, K. G. Larsen, A. Legay, and D. B. Poulsen. Statistical model checking of dynamic networks of stochastic hybrid automata. *Electronic Communications of the EASST*, 66, 2014.

[19] D. L. Deephouse. Does isomorphism legitimate? *Academy of management journal*, 39(4):1024–1039, 1996.

[20] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

[21] H. Garavel, F. Lang, and W. Serwe. From LOTOS to LNT. In *ModelEd, TestEd, TrustEd*, pages 3–26. Springer, 2017.

[22] H. Gregersen and H. E. Jensen. Formal design of reliable real time systems. 1995.

[23] J. F. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, W. Wesselink, T. Willemse, et al. The mCRL2 toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, page 53, 2008.

[24] A. Hagberg, D. Schult, and M. Renieris. pygraphviz. `https://github.com/pygraphviz/pygraphviz`, 2006.

[25] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[26] E. M. Hahn, A. Hartmanns, H. Hermanns, and J.-P. Katoen. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design*, 43(2):191–232, 2013.

[27] A. Hartmanns and H. Hermanns. The Modest Toolset: An integrated environment for quantitative modelling and verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 593–598. Springer, 2014.

[28] A. Hartmanns and H. Hermanns. Explicit model checking of very large MDP using partitioning and secondary storage. In *International Symposium on Automated Technology for Verification and Analysis*, pages 131–147. Springer, 2015.

[29] A. Hartmanns, J.-P. Katoen, B. Kohlen, and J. Spel. In *Tweaking the Odds in Probabilistic Timed Automata*, 2021.

[30] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, and E. Ruijters. The quantitative verification benchmark set. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 344–350. Springer, 2019.

[31] T. A. Henzinger. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

[32] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[33] J. G. Kemeny and J. L. Snell. *Markov chains*, volume 6. Springer-Verlag, New York, 1976.

[34] M. Kural. Tree traversal and word order. *Linguistic Inquiry*, 36(3):367–387, 2005.

[35] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[36] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282(1):101–150, 2002.

[37] E. A. Lee. Embedded software. In *Advances in computers*, volume 56, pages 55–95. Elsevier, 2002.

[38] J. Pombrio and S. Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 361–371, 2014.

[39] M. O. Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.

[40] Češka, Milan and Dannenberg, Frits and Paoletti, Nicola and Kwiatkowska, Marta and Brim, Luboš. Precise parameter synthesis for stochastic biochemical systems. *Acta Informatica*, 54(6):589–623, 2017.