A Model Reduction Approach for Improving Discrete Event Simulation Performance

Alexander Pacholik, Wolfgang Fengler
Ilmenau Technical University
Helmholzplatz 5
DE-98693 Ilmenau, Germany
alexander.pacholik@tu-ilmenau.de
wolfgang.fengler@tu-ilmenau.de

Tommy Baumann, Michael Rath
Andato GmbH & Co. KG
Ehrenbergstraße 11
DE-98693 Ilmenau, Germany
tommy.baumann@andato.com
michael.rath@andato.com

ABSTRACT
Discrete event simulation (DES) performance is a crucial factor when applying large scale simulation models. It limits the ability to produce high quality simulation results within given time bounds, and thus limits the ability for iterative model evaluation. In a holistic hierarchical modeling approach, models are built from basic building blocks, which define the model granularity. Hence, the resulting model granularity is not optimal with respect to the execution semantic, and thus has a limiting impact on simulation performance (granularity gap). The Purpose of this article is to propose an automated model reduction approach to close the granularity gap. The key idea is to merge basic model entities in order to lower model granularity and improve simulation performance. The article discusses requirements and limitations of model reduction in DES and presents the architecture of a research prototype.

Categories and Subject Descriptors
I.6 [**Simulation and Modeling**] Miscellaneous
I.6.4 [**Model Validation and Analysis**]: Discrete Event - model reduction, synthesis, simulation performance optimization.

General Terms
Performance

Keywords
DES, optimized synthesis, model reduction, performance

1. INTRODUCTION
To design, analyze, evaluate, validate, and optimize systems and processes, modeling and simulation methodologies and technologies are applied [1]. For this purpose the real world systems or systems to be designed are captured in form of executable models, also designated as virtual prototypes. Prerequisites to apply executable models are so called execution domains. In this context the execution domain DES has gained significance since computing power increases in a way that simulation based approaches can be utilized in operative planning. DES is used in a broad set of application areas, e.g. energy, telecommunications, production, logistics, avionics, automotive, business processes, and system design. Inter alia DES is applied for dimensioning of resources, to answer questions about topology, scalability and performance regarding operational scenarios, and to estimate risks. Hence, the simulation performance needs to keep up with the enormous complexity increase of executable models (size, granularity), which in turn comes from the complexity increase of systems and processes as well as the customer requirement to create holistic, integrated, high accuracy models delivering more expressive simulation results. Other performance demanding tasks are long term simulations, iterative optimization loops, test batteries, real-time models (higher reactivity to market) and automated specification and modeling processes [2]. Hence the model size/granularity as well as execution performance are crucial factors in successfully applying modeling and simulation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Simutools 2013, March 05-07, Cannes, France
Copyright © 2013 ICST 978-1-936968-76-3
DOI 10.4108/icst.simutools.2013.251734

In the scientific community techniques used in DES approaches have been studied and constantly improved. Recent work is concentrated on scheduler optimization [3, 4, 5] and parallelization approaches [6, 7]. Nevertheless, even with parallelization approaches, the ability of runtime scaling is limited and long term simulation requires a lot of computing time, which rarely profits from the availability of computing-clouds, since the scalability is limited depending on model characteristics [8, 9].

To reduce DES computation time the conventional approach is to optimize performance critical parts by hand-written source code. This may be time consuming and error prone. Thus, the basic idea is to look on the ability to merge simulation entities, as depicted in Figure 1, and to develop an automatable method for model reduction prior to simulation runs. How such a method affects execution time for sequential and parallel DES simulations is the goal of an ongoing research project.

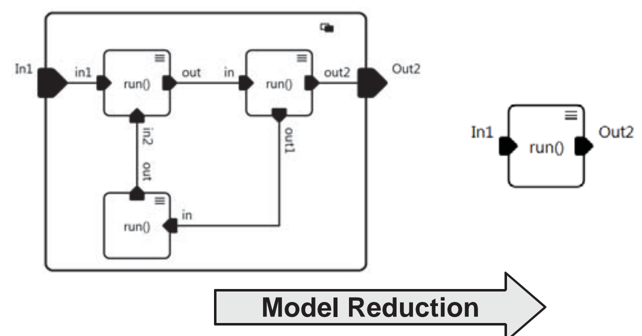

Figure 1: Basic Model Reduction Concept

The idea of model reduction is related to causality analysis [10] and model checking [11], since it is generally necessary to reason about the causality preserving compositionality of actor based systems. The structural decomposition regarding simulation semantics and optimization based on it is related to the concept of *models of computation* (MoC) and to multi-domain simulation [12, 13, 14].

The paper is organized as follows. In section 2 the challenges of model reduction are explained. Section 3 introduces the overall concept and steps involved in the method. Preliminary results are discussed in section 4. The paper closes with a conclusion and an outlook to current and future work.

2. MODEL REDUCTION CHALLENGE

In actor-oriented holistic DES simulation systems state of the art is to distinguish between atomic model entities, which are described in any programming language and compiled into binary code, and composite model entities, which describe the composition of atomic and composite models [10].

Composite models provide the structuring and composition of models, but do not contribute to the execution semantics. Atomic model entities interface with the simulation kernel and contain the whole behavioral description, including event consumption and creation, time advance of events, manipulation of data entities, etc. Thus, without any further knowledge about the internal behavior of atomic model entities, a fusion of atomic model entities is not allowed. Equivalent behavior can be assured by checking the following properties:

1. The execution order (event scheduling) of entities is preserved, except hidden events of merged entities.
2. The value and order of the global simulation state is preserved, except hidden states of merged entities.

When the first requirement is fulfilled also for hidden events, the second requirement is also fulfilled, which requires to preserve the granularity, e.g. by hierarchical scheduling.

2.1 Atomic Interface Definition

Actor-oriented holistic DES simulation systems rely on a well defined interface for atomic and composite simulation interfaces [10]. An example interface, which is also used as prototype is presented in this section, which contains some extensions for model reduction (bold italic).

An *atomic entity* can contain *ports*, *states*, *parameters*, and *methods*. The simulation entity definition is extended by a ***run-condition*** property, which specifies the synchronization of incoming events. Atomic entities can *inherit* from other atomic entities.

A *port* represents a communication interface from or to another simulation entity by a logical communication channel (*connection*). *Ports* are characterized by *data-type*, *direction*, *priority*, and optional ***zero-delay***. The ***zero-delay*** specifies that all tokens are transmitted without delay to the receiver.

A *state* represents a data entity (memory), which contains a part of the simulation state. Depending on the *visibility* (internal/external) a state can be either only accessed locally or shared with other simulation entities. States are further characterized by *data-type* and ***access-type*** (read/write/snoop).

A *parameter* holds a data value, which is defined at modeling time and remains constant during simulation. A *parameter* is defined by *data-type* and *value* can be calculated from other parameters.

A *method* encapsulates a user-defined functionality, which process data or affect states. *Methods* are defined by *signature* and *visibility*.

Some features extend the basic DES simulation algorithm, in order to simplify modeling a certain behavior and avoid cumbersome replacements by providing shortcuts in the simulation kernel. Examples are event cancellation, state snooping (state event). The drawback of such *advanced DES features* is the increase of complexity and introduction of possible side effects when analyzing possible behaviors of the model for model reduction and parallelization. As a result the utilization of *advanced DES features* have to be decidable, e.g. by ***port-type*** specialization, ***access-type*** of states or restriction of utilized API functions.

2.2 Decidability of the Equivalence Problem

Deciding the functional equivalence of two models (submodels) is a nontrivial task. There exist several approaches to equivalence checking depending on the conceptional layer of the models, on which the equivalence checking is applied, see Table 1.

Table 1: Conceptional Layers for Model Analysis

Conceptional Layer	Vocabulary	Decidability
DES Composition Layer [10] [18]	interconnection of atomic ports, states, parameters	cycle free, causality dependencies, unused ports, read-write dependencies for state access
DES Atomic Interface Layer [18]	Ports, States, Parameters, Lifecycle Methods, Inheritance, entity types	Restrictions on utilization of kernel interface features, model-data type resolution, external references
Code Layer (C, C++, Java) [15]	C++ Types, Variables, Instructions, Method calls, Control- / Dataflow, Inheritance	Control-/ dataflow dependencies, reachability of method calls, dead code, initialization, c++ type resolution, path coverage, utilization of external code
Machine Layer (Object code) [16][17]	Register, Memory, Instructions, Jump/ Call Instructions	Data values, reachability, deep control-/dataflow dependencies, model checking

The conceptional layers relate to the technical representation of the simulation entities, which defines the information available for analysis (vocabulary). The *DES Composition Layer* describes interconnection of atomic entities within a model, based on the atomic entities interface definition. The *DES Composition Layer* allows analyzing structural dependencies and properties, e.g. Read-Write order on states or causality cycles [10].

The *Atomic Interface Layer* describes the interface of atomic entities (see section 2.1) as well as restrictions on features available for the functional description.

The *Code Layer* contains the functional implementation of a certain atomic entity, in the form of lifecycle methods and custom code sections. It enables deciding about data-flow and control-flow dependencies by software model checking [15] with instrumented code. However, some dependencies cannot be discovered due to dependencies on external code and simulation APIs, or require to include the simulation kernel or an abstraction of it.

The *Machine Layer* contains preprocessed and compiled code for simulation entities and references to external runtime libraries. In principle, exhaustive analysis in terms of model checking is possible [16, 17].

If two models are completely independent, e.g. different atomic entities and granularity, the equivalence must be decided on DES Atomic Interface Layer and Code Layer including the whole model, using model checking methods. We recognize such an approach as not feasible for deciding about equivalence, due to the potential state complexity of a model. When one of the two models is conducted from the other by transformations, such as composition and decomposition, the equivalence checking task may be reduced to decide the validity of compositionality [18]. In the application domain of DES simulation two models are not coupled directly, but by scheduler interaction, and the interface is designed for interoperability by a generic port interface method. Such methods alone are not feasible, because they do not consider state changes. Methods based on a tagged signal and model causality requires complete models to conduct decisions [10].

2.3 Combined Approach

As a consequence we decided to develop an own approach for deciding the equivalence between an original and a composed model, which partly covers all conceptual layers by integrating existing techniques. The first step is to classify model entities regarding their conformance levels for composition based on its execution semantics/ MoC. This depends on the entity interface (DES Atomic Interface Layer), utilization of simulation kernel application programming interface features and behavioral properties (Code Layer / Machine Layer). The second step is to analyze the model structure regarding valid composition patterns, including scheduling priorities, to determine valid composed models (DES Composition Layer).

2.4 General Restrictions on Model Reduction

A real fusion of model entities with a changing of granularity is only valid when certain restrictions can be assured, e.g. no event delay, synchronization, data flow semantics, data access[1].

This is exactly what manual optimization does, and we call it *model reduction*. When looking on nontrivial composition patterns, a number of aspects, such as scheduling strategy, type and order of data accesses, hidden model states, etc. have to be considered. The validity of fusing model entities can be treated as solution of an equivalence problem. The change of granularity is accomplished by model synthesis.

[1] Reasoning about delay and synchronization issues would requires a global analysis scope including dynamic effects in of the global DES event queue, see section 2.

3. MODEL REDUCTION CONCEPT

The main goal of model reduction is to increase simulation performance in terms of execution time[2] by a fully automatic preprocessing step prior simulation model execution. Figure 2 depicts the proposed model reduction process, consisting of four main steps. In the first and optional step, *performance profiling and hotspot analysis*, the model is profiled in conjunction with the desired target scenario, since the optimization may depend on a concrete parameterization of a system model, see section 3.1. In the second step *behavioral analysis* is applied to check whether a certain model entity conforms to preconditions for applying optimization, see section 3.2. The next step, *model optimization*, includes the structural analysis of the original model, determination of *optimization regions*, and identification of an optimized model structure, see section 3.3. In the last step, *model synthesis*, the target model entities and back annotation are generated according to the optimized model structure. The *model synthesis* accomplishes a change of granularity by generating different code or defining a different scheduler, as explained in section 3.4. The current status of the concept implementation is explained in section 3.5

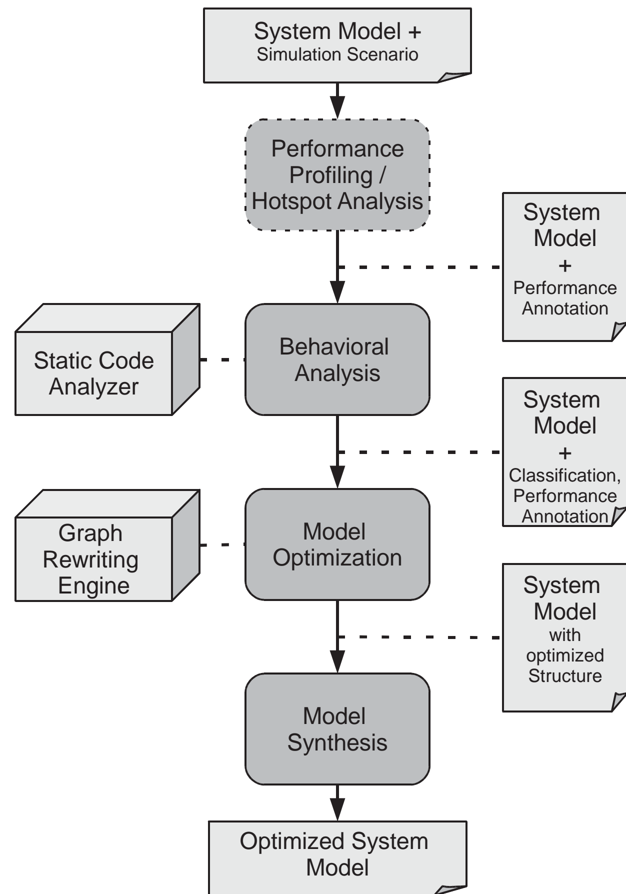

Figure 2: Model Reduction Process

[2] Other aspects, e.g. memory consumption may also be interesting, but this usually requires changes in the modeling concept and is thus not considered in this article.

3.1 Performance Profiling

Performance profiling is suggested to be a preprocessing task in order to provide information for the optimization step. Performance profiling assesses CPU utilization and event statistics and thereby block-granularity and communication-density produced during a certain simulation scenario. The required information can be collected by the simulation kernel, simulation log and performance profiling tools. This information may be used to provide a better tailoring of the model for optimization alternatives or parallel DES simulation. E.g. a large optimization model reduction may be split in order to enable parallelization, or an identified optimization region may be skipped due to size or lack of impact. Performance profiling is not generally necessary for model reduction and model optimization, but the gained information can be used to influence the optimization process and improve the optimization results, see section 3.3.

3.2 Behavioral Analysis

Deciding about the compositionality of model entities requires knowledge about the behavior of atomic entities as well as their interaction and scheduling, see section 2. Behavioral information is derived from:

- Structural model information, e.g. state access, connectivity. (DES Composition Layer)
- Dynamic model information, e.g. static synchronization, priorities, event order. (DES Composition Layer)
- Structural user code information, e.g. custom code sections, simulation kernel APIs utilized in user code. (DES Atomic Interface Layer)
- Dynamic behavior of user code, e.g. port queue limitation, dynamic synchronization. (Code Layer, Machine Layer)

The structural information can be gathered from the internal structure (definitions, methods, attributes) and external dependencies (object symbol table) of user code.

3.2.1 Classification of Model Entities

The three conformance classes *DES scheduling* (*D*), *Hierarchical scheduling* (*H*), and *Dataflow based Synthesis* (*S*) are considered for atomic model entities, based on the simulation semantics (model of computation) required to preserve a correct simulation. Between the conformance classes the subset relation $D \supset H \supset S$ holds, e.g. all model entities in classes *H* and *S* can also be used without optimization, i.e. with standard DES scheduling of class *D*. The resulting classified model structure is comparable to a multi-domain simulation model, with computation domains DES, hierarchical DES (HDES) and synchronous dataflow (SDF), as depicted in Figure 3.

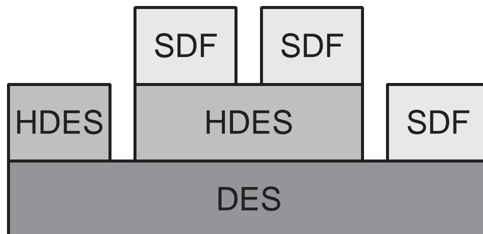

Figure 3: Layering of Conformance Classes

3.2.1.1 DES Scheduling (D)

DES scheduling is the basic conformance class, which means that no optimization is possible for a certain atomic block and thus the standard scheduling algorithm is used.

3.2.1.2 Hierarchical DES Scheduling (H)

In hierarchical scheduling, timed and untimed regions (optimization regions with zero-delay, depending on structure and priorities) are handled by different scheduling algorithms in a hierarchical manner. Globally, timed DES scheduling is applied, whereas the untimed regions can be managed by a simpler local scheduling algorithm based on a priority queue. If an atomic block conforms to hierarchical DES Scheduling, it can be applied for local scheduling, which avoids some of the overhead coming with a global event queue and maintaining time. Instead, a local event queue is used for respective untimed regions. For elements without utilization of a time delay, we basically need to ensure the absence of advanced DES features, see section 2.1. Analysis of Restrictions regarding dynamic behavior is not required. Another advantage is that hierarchical scheduling can be applied using original simulation model objects, without changing any line in the user code.

3.2.1.3 Dataflow based Synthesis (S)

For advanced optimization in terms of dataflow based code synthesis, it must be assured that an atomic block provides deterministic behavior, which conforms to the synchronous dataflow (SDF), Boolean dataflow (BDF) or scenario-aware dataflow [14] (SADF) model of computation. The SDF model of computation states, that on availability of a given number of data tokens (events) for input of an atomic block, the computation produces a given number of output tokens (events) at each output. This allows a predictive scheduling, allows replacing ports by local variables, and completely removes the need for dynamic scheduling within such an optimization region.

3.2.2 Formal Behavioral Analysis

The absence of advanced simulation features can be ensured on *DES Atomic Interface Layer* by a restricted set of kernel API functions which is available within the use code.

Checking that the dynamic behavior of ports conforms to the synchronous dataflow MoC states, that during an execution of the entity a certain number of event tokens is consumed a the input ports, and eventually a certain number of event tokens is produced at the output ports according to the specified synchronization behavior, for any (repeated) sequence of execution. For simple SDF models the number is *one* for all ports.

Checking dynamic behavior with model checkers like the *Static Driver Verifier Research Platform* [17] or *SATABS* [16] requires additional steps, in order to provide proper platform model and test setup for analyzing atomic model entities. Figure 4 depicts the proposed test setup. The simulation kernel is replaced by a *non-deterministic kernel model* (platform model), which provides all possible interactions to the *atomic model entity* under test, including event generation, state access, and triggering of the atomics simulation methods (test patterns). The behavior is checked by state machines for *ports* and *states*, which are coupled with the Event and State APIs by hook methods. The state machines check the kind and number of accesses to ports. This, together with a set of properties allows verification of expected results and validity of interactions.

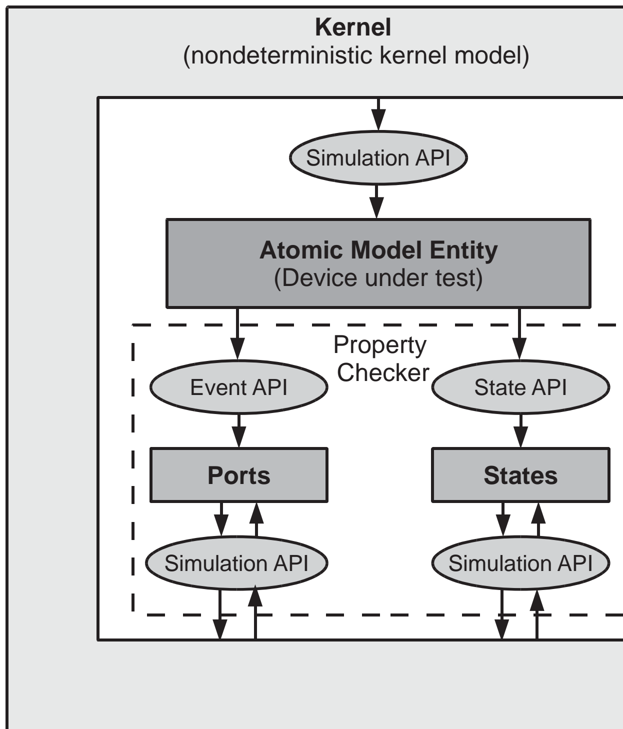

Figure 4: Test Setup for Behavioral Analysis

We are going to use the *Static Driver Verifier Research Platform* [17] as a static code checker. This requires generating interaction models and conformance rules (*property checker*) for each atomic entity, depending on its interface. For simple SDF models, which consume and produce exactly one event token in each simulation step, the interaction model and conformance rules can be generated automatically. The *kernel model* and generic state machines for ports and states need only be provided once. The result of behavioral analysis is a classification of atomic model entities which can be used for model optimization.

3.3 Model Optimization

After classification of atomic model entities, the structural analysis and tailoring into optimization regions takes place. An optimization region determines a connected part of the model, which belongs to a conformance class. This does not only depend on the atomic model entities, but also on the structure, such as the priority of ports/connections, structural cycles, and interference of state accesses. For simplicity we assume a flattened model hierarchy. The optimization regions are determined by recursively merging optimization regions with the same conformance level, starting with conformance level *S:*

1. Each optimization regions consists of at least one entity.
2. Two optimization regions can be merged, if they are connected to each other and belong to the same conformance class and all interconnections are without delay and have a higher priority than other connections outside of the optimization regions. Merging entities of conformance class *S* also requires the synchronization condition to be unchanged.
3. When no further merging is possible, all regions consisting of one entity are shifted to the next lower conformance level, and the algorithm is repeated with the next conformance level, until *D* has been reached.

At the end, the simulation model is tailored into optimization regions, which can be optimized using model synthesis or a hierarchical scheduling, as explained in section 3.2.1 and depicted in Figure 3. There is only one DES region, containing HDES and SDF optimization regions. We propose to use a graph rewrite engine [19] to implement the model optimization algorithm. Using a graph rewrite engine instead of direct coding allows operating on model level in a declarative way.

In certain conditions the optimizations regions may not be unique, but depend on the order of rule applications.[3] Another important point is, that an improvement requires a certain size of the optimization region, e.g. more than one element, otherwise there will be no effect or even a performance degradation due to increased overhead. On the other side, one Optimization region is bound on one computing resource (logical processor), which affects the granularity and parallelism available for parallel DES (PDES). This may also result in performance degradation, if the granularity of optimization regions becomes too large. We will need to add heuristics and include performance profiling results to solve these issues in future work.

3.3.1 Debugging and Datamining

Debugging and datamining during simulation are two important aspects for the model optimization process, since the model reduction should be as transparent as possible. I.e. the modeling engineer should not care about code objects and possible restructuring during optimization.

For datamining a separate layer is considered, which determines the location for extracting data which is used for data visualization and data analysis. This means the datamining objects do not contribute or inference with the simulation it selves, but utilize the data generated by the simulation. This allows decoupling simulation and datamining. When the model is changed during optimization, the datamining must be relocated and if necessary additional ports must be provided. Supporting datamining as additional layer is optional. If not supported, according elements are regarded as normal simulation objects

Model based debugging allows following the execution according to the scheduling strategy and model granularity. During model execution the physical granularity and structure of the model is changed, so the debugging must provide a feedback about the simulation-step-granularity and associate debugging information to the correct model entity.

3.4 Model Synthesis

The model synthesis consists of two parts. First, the new model structure resulting from the model optimization must be established. For each optimization region a composite containing the atomic model entities is created, with one top-level composite of type *D*, which is controlled by the original DES scheduler. Regions of optimization type H are annotated with a local scheduler, resulting in hierarchical scheduling.

The second part is to apply synthesis for the regions of type *S*. This step requires creating new atomic entities which combine the functionality of the region, while avoiding the overhead resulting from simulation kernel API and dynamic scheduling. Technically this means replacing ports by local variables, and merging the original state sets. The practical implementation requires a

[3] E.g. an element might be legally merged into more than one region. However, so far we could not produce such a situation.

number of detail solutions, e.g. for providing replacements for equal port names, equal state names and providing access to interfaces via pointers.

3.5 Tool Support and Current Status

The DES modeling and simulation tool *Modeling and Simulation Architect* (MSArchitect) [20] is considered to implement the proposed model reduction process. MSArchitect was chosen, since it brings layered APIs with access to model manipulation, simulation kernel interaction and a rigid code synthesis mechanism, which enables to structurally analyze features and dependencies in the generated code. Thereby, utilization of *advanced DES features*, access to the kernel API and access to external code are decidable.

Currently, the implementation and integration of the model reduction process is in progress. Behavioral analysis and model optimization have been implemented for conformance classes *D* and *H*. Static scheduling has been implemented as lightweight extension, by annotating a composite entity of conformance class *H* with a local scheduler which takes control during simulation time. The extension to conformance class *S* and realization of its model synthesis is currently work in progress.

4. Preliminary Experimental Results

To validate the overall concept, we applied model reduction on two basic test cases manually. Both test cases contain a composite for optimization, which is executed in a loop. The purpose of the tests is to validate feasibility and effect of optimizations by example. Therefore we compare results from the original composite model with DES scheduling, with the hierarchical scheduling approach, and with a manually conducted atomic model as replacement for the original composite. The original composite represents the conventional DES scheduling approach without optimization. The manually conducted model reduction represents the reference for an automated synthesis solution. All results were obtained using a Workstation with an Intel i7-990X processor and 12 GB RAM running MSArchitect [20] on Windows 7 Professional.

4.1 Hamming Number Generator

The first example a Hamming number generator, which produces an ordered sequence of integer numbers, which can be written as $2^i \cdot 3^j \cdot 5^k$.
The whole model is depicted in Figure 5. The white rounded rectangles represent atomic model entities. The arrow shaped pentagons represent ports according to their direction, which are connected according the solid lines. The grey rounded rectangle contains the composite containing the main computation, which also represents the optimization region. The composite is embedded in a loop structure and executed once for each Hamming number. The simulation is terminated by the *DumpValue1* block, if a certain count of hamming numbers has been generated. All output ports except *Impulse1.out* emit Events of data-type *integer*. All atomic entities except *Impulse1* and *DumpValue1* conform to the *Dataflow based Synthesis* class and can be optimized.

Table 2 contains the execution time, the relative performance improvement to the original model, and number of simulated events for generating 10 million Hamming numbers. The granularity of atomic simulation steps within the composite, which are handled by the conventional DES scheduler, is evaluated in average CPU cycles. With hierarchical scheduling on the composite the execution time is reduced by 16.5 %, the utilization of the DES scheduler is reduced by 75 %, although the granularity increases by 600%. It can be concluded, that the reduced overhead outweighs the increase of granularity. The manually conducted atomic model is able to reduce the execution time by 78 %, with the same scheduler utilization. A decreased granularity is observed in the atomic model, indicating the presence of low-level effects in the small model.

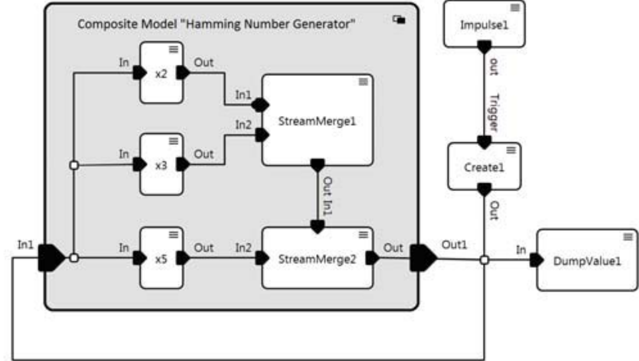

Figure 5: Hamming Number Generator Example Model

Table 2: Hamming Number Generator Results

Version / Results	Composite (D)	Hierarchical Scheduling (H)	Atomic Model, manual (S)
Execution Time	20.71 s	17.92 s	4.53 s
Improvement	-	16.5 %	78.1 %
Events (DES)	80 million	20 million	20 million
Granularity	708 CPU cycles	4203 CPU cycles	613 CPU cycles

4.2 One-To-Go Example

The second example contains the generic functionality "One-To-Go", to filter out duplicate events occurring at the same time instance. Figure 6 contains the according "One-To-Go" algorithm. Keywords are bold italic (`var`, `void`, `if`, `else`), API functions are italic, e.g. `hasEvent()`, `receive()`, `send()`, `getCurrentTime()`. Elements of the atomic entity interface are bold, such as ports (`In1`, `Out1`), states (`LastTime`) and the simulation function `run()`.

```
var LastTime = -1; //internal state

void run() {      // simulation function
  if (In1.hasEvent()) {
    var time = getCurrentTime();
    if (time > LastTime) {
      LastTime = time;
      Out1.send(In1.receive());
    } else {
      In1.receive();
    }
  }
}
```

Figure 6: One-To-Go Algorithm

The simulation model is depicted in Figure 7. It contains the grey-shaded composite, which realizes the "One-To-Go" algorithm. All atomic model entities in the composite conform to the *Dataflow based Synthesis* class, whereby the *ConditionGate* is associated to the BDF/SADF MoC domain. The atomic model entities *Read-*

State and *WriteState* access the internal state *LastTime* storing the timestamp of the most recent incoming event. The model contains two event generators *Clock1* and *Clock2* with equal parameterizations, such that for each time instance one event is filtered out.

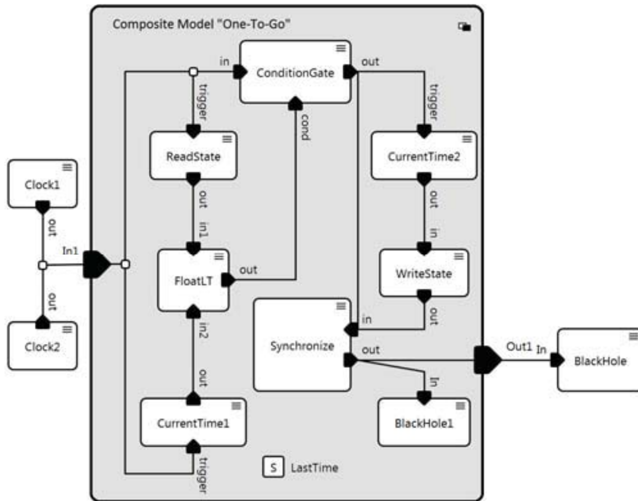

Figure 7: One-To-Go Example Model

Table 3 summarizes the results of simulating 2 million incoming and 1 million outgoing events. We can observe a 33 % performance increase when using hierarchical scheduling instead of pure DES scheduling. Thereby the utilization of the DES scheduler is reduced by 77 %, while the granularity increases by factor 8.65. The manually optimized model provides a performance increase by approximately 80 %, providing the same model granularity.

Table 3: One-To-Go Example Results

Version / Results	Composite (D)	Hierarchical Scheduling (H)	Atomic Model, manual (S)
Execution Time	5.63 s	3.76 s	1.14 s
Improvement	-	33.1 %	79.8 %
Events (DES)	22 million	5 million	5 million
Granularity	455 CPU cycles	3936 CPU cycles	455 CPU cycles

4.3 Result Interpretation

While the expressiveness of the experiments is limited due to the small size of the models, we can draw some conclusions regarding the general applicability of a model reduction method as proposed in this paper.

The result of model reduction is a decreased number of computed events and thus a reduced simulation overhead. Hierarchical scheduling produces a coarser model granularity with respect to conventional DES scheduling. However we can expect only a fairly constant reduction of overhead, which also depends on the concrete scheduling algorithm implementation. Nevertheless, the reduction may become more significant, when considering large event queues.

The synthesis method is yet not fully implemented. However, we can expect a significant effect for fine grained simulation models containing optimization regions which can be used for dataflow based synthesis.

4.4 Supposed Advantages for PDES

Some advantages are supposed to occur when combining model reduction with PDES simulation. The main advantage is reduced simulation overhead. Hierarchical scheduling result in a coarser model, thus PDES simulation may benefit, e.g. the potential number of rollback events is reduced. However, the effect strongly depends on the model structure and distribution of model entities onto processing resources. Potentially model reduction reduces the applications inherent parallelism by synthesizing parallelly executable entities into one entity. Another aspect is the distribution of memory requirements by PDES, which may be changed by model reduction. Balancing between model reduction and parallelization will be necessary, and can be achieved by using structural information and performance profiling information. Model reduction is expected to be in principle more flexible than manual optimization, e.g. it may depend on model structure or utilize profiling information from previous simulation runs.

4.5 Limitations

Compositional data flow synthesis cannot reduce the inherent computational complexity and effort of a simulation (time and memory), but only reduce or eliminate the overhead which is introduced by a holistic modeling and simulation approach. Optimization regions must not contain advanced DES features which require kernel interaction, such as cancelable events. The model entities must not use features, which evaluate or manipulate the model structure on run time.

5. Conclusions and Future Work

In this paper an automated model reduction method for improving execution performance of DES models has been proposed as work in progress. The steps for analyzing models and conducting an optimized structure have been explained. Preliminary results, based on small examples, indicate the feasibility of the approach. We expect that the improvement for real world models will be slightly smaller, than in the examples presented in this paper.

In fact, classification of model entities and having access to the required structural and behavioral information is the crucial point of our approach. Nevertheless, we think the approach could be adapted to other tools without loss of generality. The effort required for porting will strongly depend on the target platform, e.g. for implementing the model entity classification and the synthesis, while the structural model optimization could be easily reused.

The next steps are the completion of the model reduction approach for MSArchitect, exhaustive experimental analysis, e.g. with the toll collect model [2]. During these steps some refinements on the proposed methods will be developed, e.g. extensions on the set of restricting interface features, refinements on the behavioral analysis model, and extensions on the set of model optimization option. We expect that the proposed model reduction approach can be conducted automatically. The feasibility of extended optimization ideas requires experimental validation after finishing the implementation, e.g. loop handling, time shifting, and considering new conformance classes, e.g. state machines.

Future work will also contain quantification of effects on PDES simulation, when PDES is available for MSArchitect. Some aspects have been discussed in section 4.4, such as influence on rollback events, inherent parallelism, and memory distribution. We expect that techniques for balancing between model reduction and parallelization must be integrated to avoid side effects.

6. ACKNOWLEDGMENTS

This research has been partly funded by the Free State of Thuringia and the European Regional Development Fund ERDF through the Thüringer Aufbaubank under grants 2010 FE 9093 and 2010 FE 9094.

7. REFERENCES

[1] T. Baumann, “Simulation driven design of distributed systems.” in *SAE 2011 World Congress & Exhibition*, Detroit, Michigan, 12.-14. April 2011.

[2] B. Pfitzinger, T. Baumann, and T. Jestädt, “Analysis and evaluation of the german toll system using a holistic executable specification,” in *Hawaii International Conference on System Sciences, 45: (HICSS)*. Grand Wailea, Maui, Hawaii, Piscataway: IEEE, January 4 - 7 2012, p. 5632–5638.

[3] S. Oh and J. Ahn, “Dynamic calendar queue,” in *Simulation Symposium, 1999. Proceedings. 32nd Annual*, 1999, pp. 20–25.

[4] K. L. Tan and L.-J. Thng, “Snoopy calendar queue,” in *Simulation Conference Proceedings, 2000. Winter*, vol. 1, 2000, pp. 487–495.

[5] G. Yan and S. Eidenbenz, “Sluggish calendar queues for network simulation,” in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2006. MASCOTS 2006. 14th IEEE International Symposium on*, 2006, pp. 127–136.

[6] S. Leye, A. M. Uhrmacher, and C. Priami, “A bounded-optimistic, parallel beta-binders simulator,” in *DS-RT ’08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 139–148.

[7] R. M. Fujimoto, “Parallel and distributed simulation systems,” in *Simulation Conference, 2001. Proceedings of the Winter*, vol. 1, 2001, pp. 147–157.

[8] D. Pawlaszczyk and S. Strassburger, “Scalability in distributed simulations of agent-based models,” in *Winter Simulation Conference (WSC), Proceedings of the 2009*, 2009, pp. 1189–1200.

[9] R. Bagrodia, R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin, and H. Y. Song, “Parsec: a parallel simulation environment for complex systems,” *Computer*, vol. 31, no. 10, pp. 77 –85, oct. 1998.

[10] Y. Zhou and E. A. Lee, “Causality interfaces for actor networks,” *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[11] U. B. Ighoroje and M. K. Traoré, “A federated tooling framework for formal analysis of simulation models,” in *Summer Computer Simulation Conference 2012 (SCSC 2012) Simulation Series*, vol. 44, no. 10, 2012, pp. 64–71.

[12] H. D. Patel and S. K. Shukla, “Formal methods and models for system design,” R. Gupta, P. Le Guernic, S. K. Shukla, and J.-P. Talpin, Eds. Norwell, MA, USA: Kluwer Academic Publishers, 2004, ch. Truly heterogeneous modeling with systemC, pp. 83–101.

[13] H. D. Patel and S. Shukla, “Towards a heterogeneous simulation kernel for system-level models: A systemc kernel for synchronous data flow models,” vol. 24, no. 8, pp. 1261–1271, 2005.

[14] S. Stuijk, M. Geilen, B. D. Theelen, and T. Basten, “Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications.” in *ICSAMOS*, L. Carro and A. D. Pimentel, Eds. IEEE, 2011, pp. 404–411.

[15] V. D’Silva, D. Kroening, and G. Weissenbacher, “A Survey of Automated Techniques for Formal Software Verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[16] A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl, “Symmetry-aware predicate abstraction for shared-variable concurrent programs,” in *Proceedings of CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 356–371.

[17] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg, “The static driver verifier research platform,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 119–122.

[18] M. Mousavi, M. Sirjani, and F. Arbab, “Specification, Simulation, and Verification of Component Connectors in Reo,” Eindhoven University of Technology, Tech. Rep., 2004.

[19] R. Heckel, “Graph Transformation in a Nutshell,” *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 187–198, Feb. 2006.

[20] *MSArchitect 1.1.2 Teaser*, Andato GmbH & Co.KG, 2012. http://andato.com/index.php/en/business-areas/msarchitect