Journal Title
XX(X):1–21
©The Author(s) 0000
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/

Increasing Performance of a DEVS Simulator by Means of Computational Resource Usage "Activity" Models

Yentl Van Tendeloo[†] and Hans Vangheluwe[†,‡,§]
† University of Antwerp, Belgium
‡ Flanders Make, Belgium
§ McGill University, Montréal, Canada
Yentl.VanTendeloo@uantwerpen.be, Hans.Vangheluwe@uantwerpen.be

Abstract
Domain-specific simulators often have an edge on general-purpose simulators in terms of performance. Their intricate knowledge of the domain allows them to aggressively optimize and take shortcuts. In contrast, simulators for more general formalisms, such as DEVS, need to support a wider set of models. Their inability to use domain information prevents DEVS simulators from achieving as high performance as their domain-specific variants. To solve this problem, we introduce a way to enhance simulation performance of DEVS models, through the use of computational resource usage models, often termed "activity" models. These models augment general-purpose DEVS models with domain-specific information, which can be used by the simulator. We apply this information in the context of data structure optimization, load balancing, and model allocation. Activity-awareness is a non-invasive extension to the DEVS formalism, meaning that activity-augmented models remain perfectly valid for use in activity-unaware simulators. Similarly, models without activity can still be simulated by an activity-aware simulator. Our approach is validated by making PythonPDEVS, a Parallel DEVS simulator, activity-aware and evaluating the performance impact on a set of benchmark models.

Introduction

Domain-specific simulators often have an edge on more general simulators when it comes to performance. This is especially prominent in the simulation of DEVS models, which allows a wide variety of formalisms to be mapped onto it, often giving it the status of a simulation assembly language [29]. Although it is the generality of DEVS that makes this possible, there is a significant performance impact. DEVS simulators can not make the same optimizations as domain-specific simulators. For example, a discrete time formalism can go without event list: all models are always scheduled. Discrete event formalisms, however, need to diligently maintain an event list. Mapping a discrete time formalism onto DEVS thus implies a performance impact, unless the DEVS simulator is made aware of this potential optimization.

We propose computational resource usage models, often termed "activity" models, as a way of passing performance information along with the DEVS model. Whether they are used or not is up to the simulator: activity models only address performance, not correctness. As such, simulators that are not *activity-aware* can still be used and yield identical

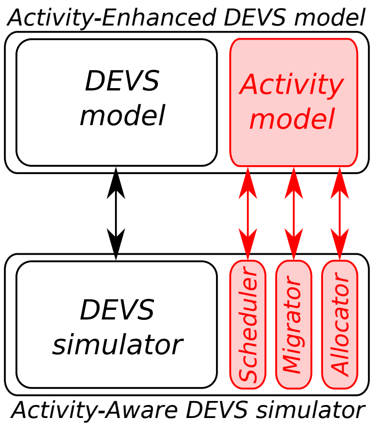

Figure 1. Activity as an optional augmentation to the DEVS model and simulator.

results. Similarly, activity-aware simulators can also simulate activity-unaware models and yield identical results. This augmentation is shown in Figure 1. Activity extends both the simulator and model, without touching any of the original specifications. This activity definition might itself be a model of the same system as the DEVS model, though at a higher level of abstraction and with different preserved properties. This is possible due to their distinct focus: the DEVS model focusses on behaviour, while the activity model focusses on performance. As a result, a DEVS model can be simulated using algorithms optimized for a specific domain.

Three components are presented that can profit from the addition of activity information: the event list scheduler (*i.e.*, internal data structure), model migration (*i.e.*, load balancing), and model allocation (*i.e.*, initial distribution of the model). For each component, we first discuss its function without activity. After activity is introduced, it is applied to these three components, making them activity-aware.

Different methods for obtaining activity metrics are presented next. Activity can be measured based on the past or the present. The future might even be possible with prediction. Domain knowledge can be combined with these measurements to further improve simulation performance.

Claimed performance improvements are made solid using several benchmarks. For this, we extended *PythonPDEVS* [25], a Parallel DEVS [4] simulator, with activity-awareness [24]. Several synthetic models are used to indicate the impact of parameters in the ideal situation. Two realistic benchmarks show the relevance to real simulation problems, and give pointers on performance improvements that can be expected in realistic scenarios.

The remainder of this paper is organized as follows. Section Performance Components presents the three simulator components that increase performance. Section Activity Definitions gives a brief summary of the different activity definitions that are used throughout the paper, and how they relate to definitions found in the literature. Section Applications presents different application domains for each kind of activity. Section Activity Measurement distinguishes three dimensions of activity measurement that we observed. Section Implementation and Benchmarks briefly discusses our implementation in *PythonPDEVS* and compares simulation performance for several models, both synthetic and realistic, with and without activity, and with and without domain-specific information. Section Related work explores related work and Section Conclusions concludes the paper.

Performance Components

This section briefly reviews three simulator components, which will be augmented with activity information later: the scheduler, allocator, and migrator. Each of these components already contributes to performance gains, but often require detailed model configuration. One major problem with manual configuration is that even minor changes to the model can require major changes in configuration. In later sections, these components are extended with activity to (semi-)automatically configure them.

Scheduler

One of the most complexity-defining parts of discrete-event simulation is the event list implementation (called the *scheduler* in the remainder of this paper).

	Average case	Worst case
List	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Heap	$\mathcal{O}(k \cdot log(n))$	$\mathcal{O}(n \cdot log(n))$

Table 1. Complexity of different scheduler types. k is the number of reschedules and n is the total number of models in the simulation.

It keeps track of event/model notices (*i.e.*, time-value pairs). Nearly every DEVS simulator uses a different data structure as the basis of the scheduler [28].

The two common types of general purpose schedulers are:

1. **list-based**, where complexity is independent of the number of colliding (un)schedule operations, but is relatively inefficient if only a single such operation happens, as the whole list is traversed.
2. **heap-based**, where complexity is dependent on the number of colliding (un)schedule operations, but it is very efficient if only a single such operation happens.

An overview of their complexities can be seen in Table 1.

The ideal scheduler is dependent on the model [25]. Specifying a good scheduler, however, proves challenging to the modeller. This is certainly the case when the modeller is unfamiliar with data structures and complexities. Even if the user has this knowledge, the number of colliding models might be difficult to estimate. Worse even, this amount can vary during simulation, and then no single ideal scheduler exists.

Allocation

Many parallel simulators require manual model allocation during initialization. This clutters the model and means that the sequential model has to be altered before it can be distributed. While this is feasible for simple models, complex models are much harder to allocate this way.

The ideal model allocation depends on many parameters, including the used simulation algorithms. With allocation hardcoded in the model, changing any parameter of the simulator or model, requires changes to the model to update the allocation logic. Whereas one of the core strengths of DEVS is its strict separation between model and solver, model allocation implicitly links them, breaking this separation. Ideally, the model, which concerns behaviour, should not be responsible for allocation, as that is an implementation detail. Simulating the same model, with the same allocation, in different simulators might give completely different performance results.

The allocator resolves this problem. After model construction, a function is called that performs the allocation using global knowledge of the constructed model. Multiple such functions can exist, each specialized for a specific synchronization protocol or simulator implementation. The model remains untouched, so comparing several allocation strategies becomes much simpler.

Migration

Even when the ideal allocation is chosen at the start of simulation, computational load might shift throughout simulation. With migration, the modeller provides migration rules, which redistribute models during simulation, thus rebalancing the load.

Note the difference with allocation: allocation happens once, at the start of simulation, whereas migrations happen throughout the simulation. Allocations specify a complete model distribution from scratch, whereas migrations specify some specific models to move.

Without support for migration, the initial distribution will be kept throughout the complete simulation run. Varying some of the model parameters can have a significant impact on the behaviour of the model, and thus on the ideal set of rules to use.

Activity Definitions

The term "*activity*" has been used for a variety of purposes in the literature. Only several of these definitions are relevant to this paper. An excellent overview is given by [17], which forms the basis for this section. In the scope of this paper, we consider activity for the *monitoring and optimization of computational resource consumption.* Although we slightly deviate from the definitions in the literature, and in particular how it is measured, definitions are conceptually similar.

Qualitative Activity

Qualitative activity distinguishes between *active* and *inactive* models. In this context, there is no numerical value (quantity) linked to the activity. A model is *qualitatively inactive* when no events occur in the horizon, and *qualitatively active* otherwise. The *horizon* signifies the period in simulated time in which the (in)active model is consistently (in)active. In the context of Parallel DEVS, an atomic model is qualitatively inactive if its time advance does not schedule an internal transition within the horizon (*e.g.*, because the time advance is infinite).

Quantitative Activity

Quantitative activity links activity to a numerical measure. In this context, the horizon implies the period in simulated time over which the quantity is accumulated.

In the literature, a model's *quantitative activity* is defined as the sum of it's *quantitative internal activity* and *quantitative external activity*. *Quantitative internal activity* corresponds to the number of internal discrete events in the horizon. It counts the number of internal computations within atomic models. *Quantitative external activity* corresponds to the number of external discrete events in the horizon. It counts the number of events received at atomic models.

In our definitions, we further emphasize the distinction between *computational resource usage* (internal activity) and *connection usage* (external activity). Note that in our definitions, we remain vague about the activity unit on purpose. For example, these units can be wall-clock time, state transitions, or even the number of elements in a queue, depending on how the value is used later on.

Quantitative Internal Activity In our definition, quantitative internal activity measures resource consumption of state transitions. Internal, external and confluent transitions all cause state transitions, so all of them are measured. Since internal activity measures computation, it is only fair to also include the external and confluent transitions.

Quantitative External Activity In our definition, quantitative external activity measures resource consumption of event exchange. Contrary to the literature, metrics are stored about the connection itself, instead of only the receiving model. This way, the source of the event is also stored.

Domain-dependent

Due to our modified definition, domain-dependent activity measures become possible: its units are left for the modeller to define. For each type of activity, we describe how we extended this notion further with domain-dependent notions. In this context, we define domain-dependent information as information that is not applicable in general. For example, the time taken in transition functions or memory used to store the state is applicable for each model, as there are no accesses to details of the model. Domain-dependent information requires accesses to particular aspects of the state, such as number of cars on a road or temperature of a surface. These notions cannot be ported between domains, and therefore require aid from the user on how to access and interpret the values.

Qualitative Activity Qualitative activity, specifying whether a model is active or not, can be augmented with domain-dependent notions of what it means for a model to be active. For example, a model that times out every so often is considered active by the general-purpose definition. This is because, in general, it is unknown whether a transition indicates activity or not, so the safe option is to assume that a transition means activity. With domain-specificity, we can also consider other states as inactive, for example with a large time advance or an empty internal transition function.

Quantitative Internal Activity Quantitative internal activity profits the most from domain-specificity. Instead of keeping a transition counter, as proposed in the literature, we extend this to an invocation on the model in that particular state. This invocation can return an arbitrary value, as long as the simulation kernel knows its meaning. For example the number of cars on a road or number of events in the queue. General-purpose activity values can also be returned, such as the CPU time for the transition or the number of transitions.

Quantitative External Activity Quantitative external activity has not been extended with domain-specific notions in our work. We believe that domain-specific notions on what is the load of exchanging an event is difficult to justify: most of the cost is on the exchange of the event, not on the type of event being exchanged. Adding in domain-specific notions increases the overhead of event passing even more due to bookkeeping. While we don't think that domain-specific quantitative external activity has no purpose, future work is needed to find a convincing use case.

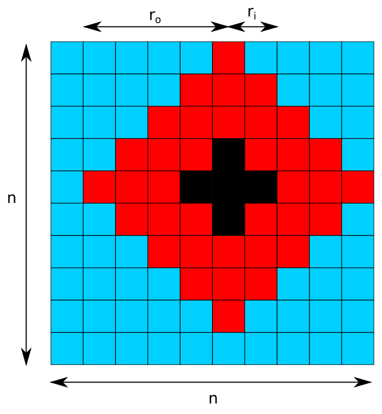

Figure 2. Application of qualitative activity can reduce complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(r_o^2 - r_i^2)$; *red* regions are burning (active), *light blue* regions are not yet burning (inactive), *black* regions are burned down.

Applications

We now focus on using the measured activity with the intention of increasing performance. All three definitions of activity will be used: qualitative, quantitative internal and quantitative external activity. In our applications, the first two can be modified by the user to add a domain-specific notion. Activity will be used to enhance the components presented previously.

Scheduler

Using activity, the scheduler can be extended in two directions. One direction is ignoring inactive models, thereby reducing the scheduler's complexity. Another direction is automatic detection and switching to a different type of scheduler during simulation.

Activity Scheduler By default, an *inactive* model is a model that has a time advance of $+\infty$ (*i.e.*, is passivated). Inactive models never become imminent, so they can be left out altogether. They can be activated by an external event, which is independent of the scheduler. By removing models from the scheduler, complexity decreases as it becomes dependent on the number of *active* models only. This optimization is easy, and is implemented in most performance-conscious simulators.

Our approach up until now was perfectly general and in widespread use. The user might, however, have additional domain-specific knowledge about the models.

A possible use case of this optimization is in a fire spread model [14]. Figure 2 shows such a model, where the influence of inactive models on simulation complexity is made clear. In the example, cells are either *not burning*, *burning*, or *burned out*. The burned out regions cannot burn again, and can be marked as permanently inactive. Not (yet) burning regions do not currently host any activity, though they might in the future. They can be marked as inactive too, further increasing performance. Complexity is significantly reduced, as we only check the burning cells. We define n as the number of cells in one dimensionx, and r_o and r_i as the radius of the outer and inner circle, respectively (refer to Figure 2). Using an activity scheduler, the complexity is then reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(r_o^2 - r_i^2)$, as the burning cells only make up a circle in the figure.

Polymorphic Scheduler Another activity-aware optimizations is the polymorphic scheduler. This is a scheduler that alters its data structure at run-time, based on observed access patterns. This idea was also implemented in tools such as Meijin++ [18].

A polymorphic scheduler doesn't use qualitative activity as defined before, but is similar in idea: *monitor behaviour* and *optimize for it.* Improved performance is obtained by alternating between schedulers at run-time, depending on the measured activity.

The polymorphic scheduler solves both problems identified when choosing the scheduler manually: (1) no knowledge about the model or different data structures is required; and (2) it becomes possible to alter the data structure during a running simulation.

An example polymorphic scheduler chooses between two different schedulers: one list-based, and another heap-based. When many collisions are detected, the scheduler switches to a list-based scheduler. Otherwise, a heap-based scheduler is chosen.

It is also possible to write a domain-specific polymorphic scheduler. Depending on the domain, different heuristics can be implemented, or a different set of schedulers can be selected, possibly even containing some domain-specific ones.

Migration

The *migrator* is also extended with activity information. Instead of using static migration rules, migrations are based on values measured during simulation.

Only quantitative internal activity is used in the migrator. Quantitative external activity is not used for several reasons:

1. It is difficult to determine the scope of what should be measured. Is it only the events exchanged between cores, or also all internal events? Is this including events that were rolled back, or not? And what about anti-events, in the case of a time warp implementation? While we don't believe that it is impossible to measure this kind of activity, we feel that we have to make many decisions, which are, at this point in our research, not well-founded.
2. While we have a clear use for internal activity, we don't know how useful the number of exchanged events will be for migrations. Most likely, the amount of actually exchanged events is unpredictable when combined with optimistic synchronization, as events are sometimes exchanged multiple times. Even changing model structure slightly can result in a significantly different number of exchanged events. We feel that the value of this measure is unreliable.
3. Technically, adding these measurements increases simulation overhead. Whereas computation can be easily measured, invoking multiple functions for every exchanged event quickly becomes a performance bottleneck. As our main motivation for the use of activity is the potential performance gain, we feel that the overhead outweighs the benefits.

An example of activity migration is shown in Figure 3, which shows the influence of a shifting computational load. In the example, the initial allocation is first chosen in such a way that there is a perfect load distribution. After some time, different models become highly active, whereas previously highly active models become almost inactive. Therefore, simulation becomes less efficient as it progresses. With activity migration, the distribution of activity is monitored throughout the simulation. The imbalance is detected and highly active models are migrated to a dedicated core again.

Activity-based migrations automatically search for migration rules to apply, very similar to existing work on load balancing [11]. Rules are no longer specific to the model configuration, but become general to all configurations of the model. Modifying simulation and model parameters no longer requires changes to the statically defined migration rules, as the migrator comes up with these rules automatically as long as its assumptions remain valid. These assumptions are usually valid for all models in the same domain and the same formalism. For example, with fire spread simulation, different configurations of the same fuel bed can be made and different sources of the fire can be selected. These configuration choices do not influence, for example, the fact that a burned out cell will be permanently inactive. The migrator is able to cope with changing model configurations, as it relies on characteristics of the domain and

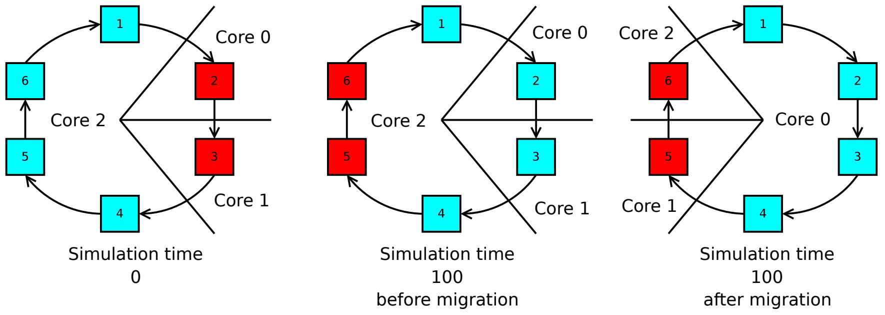

Figure 3. Activity-based migration example; (dark) red models are highly active, whereas (light) blue models are nearly inactive.

formalism, and not on the configuration. Whether or not it is domain-specific is up to the user: if it isn't domain-specific, the migrator only accesses general performance metrics; otherwise, the migrator can access the full model state.

Domain information is helpful to:

1. **Define the granularity of the migrations.** A general-purpose algorithm has no idea which two models are closely linked together. While good design of the model suggests that closely linked models are in the same coupled model, this is not necessarily the case. Domain-specific algorithms can help by suggesting which models should never be split up.
2. **Define a priority on some migrations.** A general-purpose algorithm has no migration cost metric. Generally, as few possible models should be migrated, each having a small state and a high computational load. This is difficult to determine automatically, as the state size and its computation fluctuates significantly. Domain-specific algorithms can help in making these decisions.
3. **Define some known bad distributions, which should be avoided.** A general-purpose algorithm has no idea which distributions are good and bad. It can make an educated guess, based on the observed activity, but it has no clue whether the new distribution is sane or not. Domain-specific algorithms can help by evaluating the new configuration before actually performing the migration.

Algorithm 1 Activity-aware migration.

```
while True do
    activity ← []
    for all i ∈ models do
        activity.append(fetch$_{activity}$(i))
    end for
    rules ← find_migrations(activity)
    for all i ∈ rules do
        perform_migration(i)
    end for
    sleep()
end while
```

Algorithm 1 describes the algorithm used by a domain-specific migrator in pseudo-code.

Allocation

Contrary to the migrator, the allocator uses quantitative internal and quantitative external activity. It has the same function as the static allocator discussed previously: finding a good allocation at the start of simulation. There are two major differences:

1. The allocator has a profiling time, specifying at what point in simulation time the allocation should be performed. For the static allocator, this was always set to zero (*i.e.*, the simulation

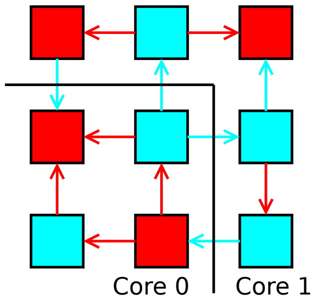

Figure 4. Activity-based allocation example; (dark) red means highly active, whereas (light) blue means nearly inactive.

start time). A static allocator is thus a degenerate case of the dynamic allocator, where no activity measurement is done. If the profiling time is longer, simulation runs sequentially (*i.e.*, all models are executed on a single core) in *profiling* mode until the profiling time is reached. In profiling mode, quantitative internal and external activity are monitored. When simulation progresses past the profiling time, the allocator is invoked with the gathered activity values. The horizon is thus equal to the profiling time. The allocator returns an allocation, just as was the case with the static allocator, which is used from then on.

2. Two extra arguments are passed to the allocator: the measured quantitative internal and external activity. These are the activity values measured up to the profiling time. Decisions are based on the activity of atomic models (quantitative internal activity), activity of connections (quantitative external activity), or both.

An example is shown in Figure 4, which shows that the activity on connections is important to make a good allocation. In the example, simulation starts with all models running locally on a single core, in profiling mode. When the profiling time has passed, the allocator is invoked with the measured activity information. The resulting allocation is shown as well: both cores have two highly active atomic models and no connection with high activity is an inter-core connection.

Algorithm 2 Main simulation algorithm combined with an initial profiling run.

```
initialize_model()
profiling ← [0 | i ∈ atomic_models]
while simulation_time < profiling_time do
    start ← time()
    simulate_step()
    end ← time()
    for all i ∈ transitioned_models do
        profiling[i] ← profiling[i] + end - start
    end for
end while
allocation ← find_allocation(profiling)
allocate(allocation)
while simulation_time < termination_time do
    simulate_step() {simulate like usual}
end while
```

Algorithm 2 shows our activity-aware simulation algorithm. As soon as profiling finishes, the model and its measured activity is passed to the allocator. After allocation, simulation continues where the profiled simulation stopped.

Normally, the user encodes the initial distribution either by embedding it in the model or by adding a static allocator. Static allocators have the same weakness as manual migrators: they are model-specific instead of domain-specific. This means that, if the model changes its behaviour ever so slightly, all information might become invalid. A dynamic allocator adapts to a changing model as long as its assumptions remain valid.

An allocator is completely different from a migrator, despite them having similar goals. Whereas the allocator returns a complete allocation, starting from scratch, the migrator returns only a set of modifications, starting from the previous distribution. So whereas a migrator can ignore mostly inactive models, as their migration overhead will be higher than the achieved performance, the allocator needs to find an allocation for each and every model. The migrator searches for slight updates to the current

	Qualitative	Internal	External
Scheduler	✓		
Migration	✓	✓	
Allocation	✓	✓	✓

Table 2. Overview of used types of activity.

distribution, whereas the allocator searches for a completely new distribution.

Overview

Table 2 summarizes the types of activity used by the different components. The scheduler only has access to qualitative activity. Migration can be based on quantitative internal activity, but this can be generalized to qualitative activity. All forms of activity can be used during allocation.

For each use of activity, it is possible to use either a general-purpose or domain-specific version. In the next section, we describe how activity values are obtained from the simulation. This shows that apart from different applications of these values, there are also different ways of obtaining the values, including domain-specific ways.

Activity Measurement

While we have already shown several applications of activity, there was no mention yet on how these values were measured. For qualitative activity, the model state is inspected at the moment it needs to be decided whether it is active or not. For quantitative external activity, we keep track of the amount of exchanged events between two different cores, using a simple counter. These measurements are trivial and not discussed further. In our approach, the most versatile kind of activity to measure, is the quantitative internal activity. The remainder of this section is devoted entirely to quantitative internal activity measurement.

Quantitative internal activity measures the computational load of a model. This is done by calling a *pre-transition* and *post-transition* function, invoked before and after the transition function, respectively. We have opted to use two separate calls to measure activity, with the second being passed the result of the first. This allows a comparison between the pre-transition and post-transition state without having two complete states in memory: only relevant values are retained. Furthermore, two calls are required anyway to measure specific physical resource consumption, such as CPU time spent or memory consumed: to compare the usage before and after the transition. Different approaches to the measurement of computational load are possible, such as the use of decorators for the transition function.

As this function can be defined by the user, there are two options: either generic information is provided (*e.g.*, CPU time, number of transitions), or domain-specific information is passed (*e.g.*, queue length, queue load).

We distinguish three dimensions to activity measurement:

1. **Time** specifies for which region in time the activity is measured. If the time is in the future, this implies that a prediction is made about values measured in the past. This time is always relative to the *Global Virtual Time* (GVT), which is the minimum of the simulation time in all solvers of all participating cores.
2. **Accumulation** specifies if activity values are accumulated or not. When accumulated, values can be averaged over the horizon. Without accumulation, a single consistent view is constructed for one point in simulated time.
3. **Data source** specifies where the data comes from. This can either be a generic function, such as measuring the time taken, or a domain-specific function, such as measuring the queue length.

Other activity-aware DEVS simulators are limited to general-purpose activity tracking, which accumulates values over the past in a general-purpose way. We contribute to this ongoing research by introducing new ways of measuring activity, in combination with domain-specific hints.

Measured values can be used in four ways: *activity tracking* (past, accumulated, general-purpose), *activity prediction* (future, accumulated, domain-specific), *activity in state* (now, no accumulation, domain-specific), and *activity in state prediction* (future, no accumulation, domain-specific).

Activity tracking

The simplest method is *activity tracking*. For each model, all activity values within the simulation time interval $[GVT - horizon, GVT]$ are accumulated into a single value. These accumulated values are presented as activity.

When using activity tracking, we optimize for the average situation in the past horizon. If the horizon does not offer a significant sample of the model's behaviour, results might be skewed. Other approaches, presented next, alleviate this problem.

Despite optimizing for the past, this method yields good estimates, on the condition that the horizon is large enough to be representative, but small enough to be recent. One of the major advantages is that it can be used as-is without any domain-specific knowledge, while still offering fairly accurate results. Unsurprisingly, this is the approach used by most algorithms found elsewhere. For example, general purpose load balancing can be seen as the application of activity tracking in the migrator.

Activity prediction

Activity prediction builds on top of values found during activity tracking. Instead of using the values as-is, future activity is predicted. Prediction algorithms are domain-specific. For example, predictions can use a simplified DEVS model to approximate load evolution, or it could use closed-form formulas to approximate the future.

If the prediction is accurate, we can approximate the near future, and thus optimize for it. When prediction is inaccurate, we optimize for an unrealistic situation, which are thus inoptimal for the actual future. Even with incorrect predictions, simulation errors never occur, as the model and simulation algorithms remain unaltered, only the choice between different components and algorithms is influenced.

Accurate predictions require domain-specific knowledge. The main question activity prediction needs to solve is: "How will activity evolve over the next horizon?" Activity prediction optimizes the model for the simulation time interval $[GVT, GVT + horizon]$ (*i.e.*, the future horizon). This is quite logical, as the next time at which measurements happen is around simulation time $GVT + horizon$. Migrations should therefore focus on optimizing the simulation up to that point, after which new migrations can be performed. The value of the next *horizon* is an unknown factor, as it is based on simulation pace. Predicting simulation pace requires domain-specific knowledge as there is no predefined relation between simulation time and wall-clock time.

The potential performance gain depends on the domain, prediction accuracy, prediction computation time, and how much the prediction differs from the past. Prediction is particularly useful when activity fluctuations are only temporary and don't require any reaction from the activity-aware optimization components.

Activity in state

The *activity in state* method differs from activity tracking, as it uses the activity in a single state, instead of the accumulated activity throughout the horizon. This single state is the last state of the interval that would be used by activity tracking. Consequently, it is the state of the model at simulation time GVT. The activity provides a *consistent snapshot* at the GVT. This requires domain-specific information, as otherwise it is impossible for the simulation kernel to interpret the state.

For some activity definitions, activity accumulation is counter-productive. An example is a fire spread model: activity can be defined as the temperature at the state. With accumulation, this returns the sum of all temperatures seen throughout the interval. Combined with the number of transitions, we obtain the average temperature in the horizon. With activity in state, the activity is the temperature at the GVT. The result is a consistent view on the complete model, containing the temperature of each cell. For optimization, the current temperature is more useful

	past	**future**
accumulation	tracking	prediction
single value	in state	in state prediction

Table 3. Categories of activity measurement. Only activity tracking can be used without domain-specific information.

than the average temperature over some period, as otherwise the temperatures are averaged out, making burnt-out cells seem active.

Activity values can be anything, as it doesn't need to be accumulated. However, the obtained values should be fairly compact, as they are gathered in a single core.

Activity in state prediction

The user is free to combine *activity in state* with *prediction*, where prediction is based on the activities of individual states. This might prove simpler than activity prediction over a complete simulation time interval. As it is the dual of activity prediction, all remarks about making predictions apply as well.

This method has the most potential, as it combines domain-specific information from both dimensions. Depending on how advanced the model is, basic predictions are easy to make this way. For example, if a queue contains ten elements, each requiring on average one second to process, we estimate that after five seconds, five elements will be left in this model, and five will be present in the next model.

Comparison

All of these different methods consider a different interval or point in simulation time. Table 3 categorizes these four methods based on the previously identified dimensions. Each of them uses domain specificity in its own way.

Activity tracking optimizes for the past, assuming the near future is similar to the recent past. Its advantage is that its data is correct if the horizon is adequate, as it doesn't involve predictions. Tracked values can be used as-is and its use for measuring the time spent in transition functions makes this the simplest method. It is insufficient when activity fluctuates drastically, when an adequate horizon is difficult to determine, or when accumulated values are unreliable. Domain-specific information can be used to have more reliable values than CPU time consumption.

Activity prediction optimizes for the near future, based on values measured in the recent past. As it is a prediction, it can take into account more domain knowledge, resulting in better results. It is possible, however, that the predicted data is (partially) incorrect. Domain-specific information must be used for the prediction of the activity evolution.

Activity in state optimizes for the current point in simulated time, using only the activity values of a single state. While it gives access to a consistent activity snapshot of the complete model, extracting meaningful activity values requires significant domain knowledge. Its usefulness depends on the domain in which it is used. Domain-specific information must be used to attach an indication of resource consumption to the current state.

Activity in state prediction optimizes for the near future, based on the current state. It gives access to a consistent activity snapshot, which might be easier to make predictions on. Predictions are based on a single point in time, instead of an interval. Domain-specific information must be used for both the resource consumption and the prediction.

Example

An example of the different measurements is shown in Figure 5, which shows the influence of measurement functions. While this might seem a new application, it is merely a more concrete version of the application in Figure 3, which presented a series of models through which activity migrates during simulation. In the example, three road segments are shown, two of which are occupied. Cars determine whether or not they can progress to the next road segment by sending queries, which are acknowledged.

Activity in state measurement can use the presence of a car as an indication of activity, whereas the absence of a car indicates inactivity. This is domain-dependent information, which cannot be transferred to other domains, such as fire spread simulation.

```
def preActivity(self):
    # Unused
    return None

def postActivity(self, preValue):
    # Only return number of cars
    return len(self.state.cars)
```

Listing 1: Code in Python for activity in state.

```
def preActivity(self):
    # Return wall clock time
    return time.time()

def postActivity(self, preValue):
    # Compute difference
    return time.time() - preValue
```

Listing 2: Code in Python for activity tracking.

	Segment		
	1	**2**	**3**
tracking	$0.0s$	$1.0s$	$0.5s$
prediction	$0.0s$	$0.0s$	$1.0s$
in state	1 car	0 cars	1 car
predict state	0 cars	1 car	1 car

Table 4. Measurements for Figure 5 at the end of horizon 1. Predictions are for horizon 2 based on horizon 1.

Activity tracking measures the wall clock time spent by the transition functions. These two options are encoded in Listing 1 and 2, respectively. In Listing 1, the model state is accessed to obtain domain-specific information. In contrast, Listing 2 merely logs the time taken by the transition function, independent of the value of the state. For prediction, we know that cars have slightly progressed before the end of the next horizon. At the end of horizon 1, this yields results similar to those in Table 4.

Due to the insignificant horizon, only the transitions of *segment 2* and some of *segment 3* are taken into account. Even though there are no transitions taking place in *segment 1*, it could still be considered active since it is processing a car. Activity tracking and activity prediction go wrong as the horizon is not representative. This problem is

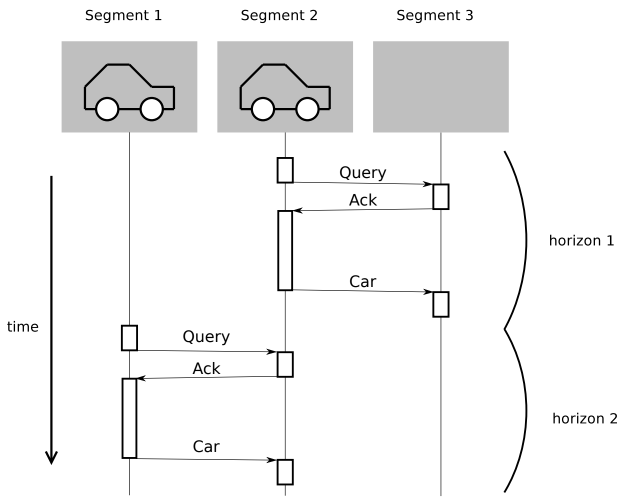

Figure 5. Measurement example

not present with activity in state measurement, as they are independent of the horizon.

Implementation and Benchmarks

We implemented the use of activity in *PythonPDEVS* [27], a modular Parallel DEVS [4] simulator, to validate our approach. *PythonPDEVS* supports both sequential [25] and distributed [26] simulation, using the previously presented performance components. The latter uses *Time Warp* [8] for optimistic synchronization, though some changes were made, similar to [9]. Whereas the name of the Parallel DEVS formalism might imply that it is always parallel, this is not necessarily the case [5]. Examples on the basic use of *PythonPDEVS* can be found online at `http://msdl.cs.mcgill.ca/projects/DEVS/PythonPDEVS`. More information on the implementation of activity in a Parallel DEVS simulator can be found in [24].

Next, we present several benchmark models: several synthetic benchmarks to evaluate specific aspects of the activity algorithms, and two realistic models, taken from the literature, to evaluate the impact of activity-aware simulation for realistic scenarios. Two realistic models, taken from the literature, show what activity-aware simulation still has its benefits in realistic models, though the impact is not as high as in synthetic models. This proves the applicability of our approach to realistic models.

We did not benchmark the use of the allocator. As the added value of an allocator strongly depends on the initial allocation, we did not have a baseline to compare with, as we would need to explicitly choose a bad allocation.

Methodology

All simulations were performed on a 30-node shared cluster of *Intel Core2 6700 dual-core @ 2.66GHz* machines with *8GB* main memory, running *Fedora Core 13, Linux kernel 2.6.34*. The number of CPU cores used is always indicated where appropriate. These machines have *CPython 2.7.5* and use *MPICH 3.0.4* as MPI implementation with the *MPI4py 1.3* wrapper. Local simulations only use a single core.

Results are averages of five simulation runs. Variability was low and therefore only the average is shown.

The PythonPDEVS formalism offers a lot of additional (not activity-based) hints that can be added to the model, such as state saving hints and message passing hints. These hints are always used in our benchmarks, both with and without activity.

Activity scheduler

The use of qualitative activity in the scheduler is the simplest form of activity in terms of implementation and use. Therefore, it is considered first.

Synthetic Our first benchmark consists of a model containing 1000 atomic models. Of these, only a fixed number of models is active (*i.e.*, has a time advance different from infinity). No connections exist between the different models, so an inactive model stays inactive indefinitely. Simulations are ran with an increasing number of active models, starting from no active models and going to all models being active. Results were obtained by using a heap-based scheduler using invalidation for reschedules. The activity heap is identical to the normal heap, but implements the check for $+\infty$.

Results are shown in Figure 7. It shows the relative difference in *total simulation execution time*, which is the sum of the simulation overhead and the theoretical simulation time (*i.e.*, the time it takes to execute the transitions only) [31]. Using an activity-based scheduler is more efficient, unless almost all models are continuously active. While the overhead is insignificant, it shows that the use of activity is not guaranteed to speed up simulation. Even when only 50% of the models are active, total simulation execution time is only decreased to 95% of its original value. This is due to the fact that simulation overhead is relatively low already: improving the scheduler only decreases the simulation overhead, and of course does not influence the theoretical simulation time. Nevertheless, simulation overhead is reduced in almost all cases.

Realistic For a more realistic model, we use the fire spread model given in [14]. Recall that this is the same model that was used as our rationale in Figure 2.

Quantized DEVS [33] was used to make sure that the fire (and thus computation) doesn't spread too fast. In normal DEVS simulation, temperature is broadcast from the cell immediately, starting a chain reaction. There is no threshold on when values need to be passed, meaning that even negligible temperature differences are communicated. These temperature differences have no significant impact on simulation results, but drastically increase simulation execution time. Quantized DEVS, on the other hand, only propagates events when the difference reaches a certain threshold. For example, in fire spread simulation, if the threshold is x degree Celcius and the last output event contained y, a new event is only sent when its value goes outside of the boundary $[y - x, y + x]$. After the event is sent, y is updated with the output event and the boundary thus moves. Simulation results get less accurate, since even these minor variations can have an impact. The threshold is therefore configured for an acceptable trade-off between simulation accuracy and efficient simulation [13, 30, 32].

In our model, the use of Quantized DEVS means that only some models become active, as the spreading of the values is mitigated. Were it not for this quantization, all models would become active

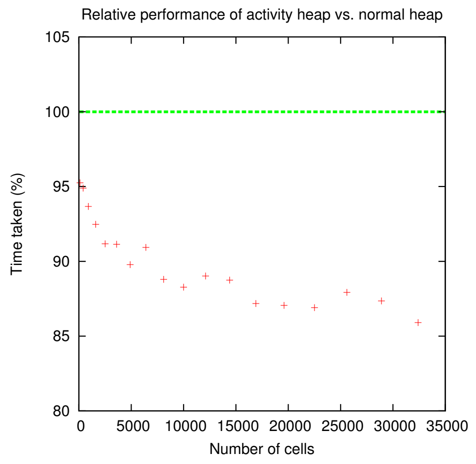

Figure 6. Time taken for simulation of the fire spread model using the activity heap, normalized with the time taken for simulation with the normal heap.

almost instantly, as even temperature changes of a fraction of a degree would be communicated.

Figure 6 shows how the different schedulers behave. Adding more and more cells, the size of the model increases significantly. These extra cells, however, only become active very late in the simulation, or even never, thanks to quantization. This makes the activity heap faster than the normal heap.

Polymorphic scheduler

For the polymorphic scheduler, we provide benchmarks for both parts of our rationale:

- The user does not need to have knowledge about data structures or the access patterns of the model. For this benchmark, we use a model with a parameter configuring the number of collisions.
- If the ideal scheduler varies throughout the simulation, the polymorphic scheduler changes its configuration at run-time. For this benchmark, a model is constructed which has different phases during its simulation, with each phase having distinct access patterns.

Different model configurations The synthetic model from before is used: a model containing several atomic models, none of them inter-connected. The

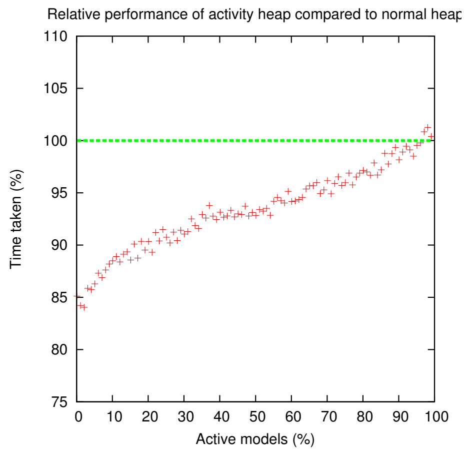

Figure 7. Time taken for simulation of the synthetic model using the activity heap, normalized with the time taken for simulation with the normal heap.

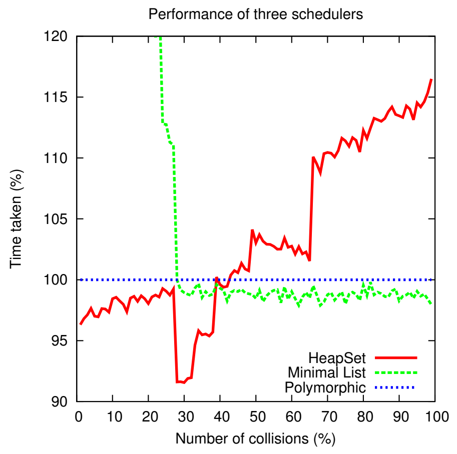

Figure 8. Relative performance for polymorphic scheduler with varying model.

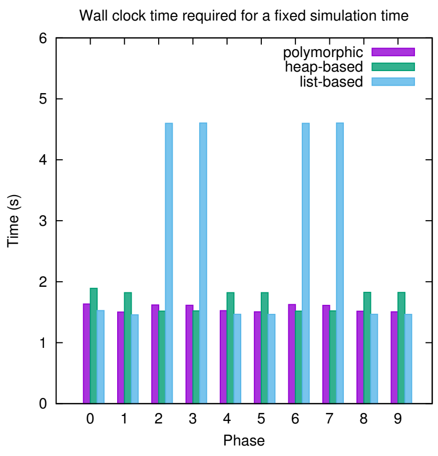

Figure 9. Polymorphic scheduler phased benchmark.

percentage of collisions is configured by altering the ratio of both types. But instead of having a mix of active and inactive models, we now have a mix of models with a random or fixed time advance, thus preventing or causing collisions, respectively.

Figure 8 shows the results near the tipping point. A heap-based scheduler is better when nearly no collisions happen, so the polymorphic scheduler uses this scheduler internally. By default, the polymorphic scheduler is calibrated to switch when 30% of models collide. We see three regions of interest in Figure 8:

1. With less than 30% of the models colliding, the heap-based scheduler is much faster than the list-based scheduler. Thanks to its heuristics, the polymorphic scheduler uses a heap-based scheduler.
2. With more than 30%, but less than 40% of models colliding, the heap-based scheduler is still slightly faster than the list-based scheduler. We notice that the internal scheduler has already switched to the list-based scheduler. This indicates that the heuristic is not perfect in this situation. Nonetheless, simulation remains correct, but only takes longer to finish.
3. With more than 40% of models colliding, a list-based scheduler becomes better. The heuristic of the scheduler becomes the right decision at this point.

In conclusion, the polymorphic scheduler often makes a correct guess about the ideal scheduler. If the user is oblivious about the ideal scheduler, the polymorphic scheduler adequately manages this situation. A small overhead is unavoidable due to the need for monitoring and potentially changing the internal data structures.

Phases in simulation For this benchmark, we use a model that cycles through two different modes: one where many collisions happen, and another where no collisions happen. At the start of the simulation, all models collide. Every 2 phases, the behavior switches between many collisions and few collisions. Using a single, statically defined scheduler is insufficient: no single scheduler is ideal in both situations.

Figure 9 shows the results of this benchmark, where the time taken in each phase is visualized. The polymorphic scheduler is never the fastest, though it is consistently fast by switching multiple times during the simulation. Summed over the complete simulation, the polymorphic scheduler is faster than either static schedulers.

Activity-based migrations

Activity-based migrations are achieved solely by using internal quantitative activity. We start with a synthetic model, closely resembling our motivating example in Figure 3. This example, however, is very similar to ordinary load balancing methods, though some information is taken into account about the model structure. Afterwards, we simulate traffic in a city during rush hour.

Synthetic This synthetic benchmark uses the model shown in Figure 3, though with a higher number of models. A ring of atomic models is constructed, of which 10% have a high computational load, and the others a neglible computational load. Activity migrates through the model, so a good initial allocation is only useful at the start of the simulation.

Figure 10 shows that activity migration helps to achieve a decent speedup. This benchmark was done using a dynamic allocator to find a good allocation

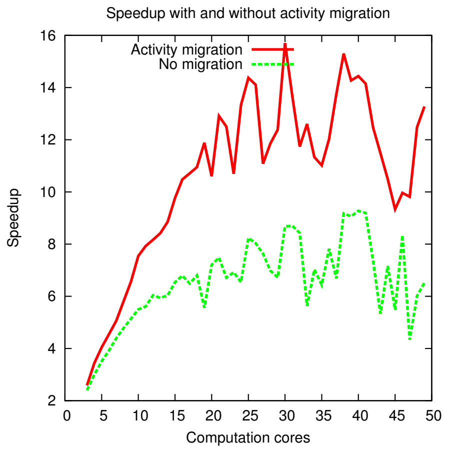

Figure 10. Activity migration synthetic benchmark for a varying number of cores.

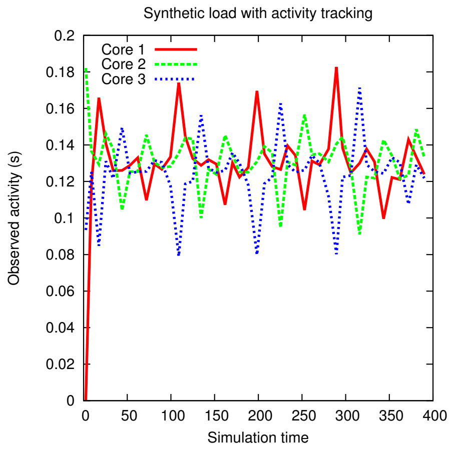

Figure 11. Activity with activity migrations.

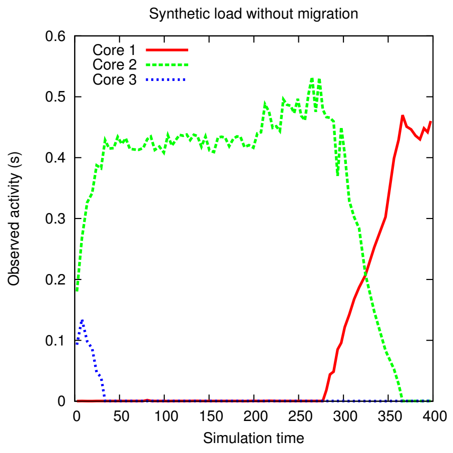

Figure 12. Activity without activity migrations.

at the start of the simulation. Models that are highly active at the start are placed on separate cores, whereas mildly active models are combined. Speedup increases slightly when no migrations are performed, though this is only due to the favorable situation created by the dynamic allocator. With migrations, speedup increases almost linearly up to about 20 cores. This comes as no surprise, as the migrator balances the load, achieving as much speedup as possible. With more than 20 cores, the model becomes too small to distribute over this many cores: each active model already has a dedicated core. This explains the sudden drops in performance for some configurations.

Apart from showing that activity can indeed increase performance, we also collected activity measurements per core from a simulation run on three cores. With activity tracking (Figure 11), the measured activity of all cores stays approximately equal, though with the occasional peaks that are quickly resolved. These peaks happen for only a brief period of time, which is the *reaction time* for the migrator. The average activity seems to be about $0.13s$ per core. Without activity tracking (Figure 12), the activity of all cores is always different. At first only core 2 and 3 are active, with activity shifting towards core 2. Node 2 quickly becomes the only core with any computation whatsoever, with an activity of $0.45s$. All other cores have an activity of (almost) $0s$. Near the end of the simulation, activity starts to shift towards core 1. This causes core 2 to become almost inactive, and core 1 to become highly active. With a bad distribution, simulation is similar to sequential simulation, but with the additional overhead of optimistic synchronization.

City Layout The city layout model is a realistic model, presented in [20]. A small example is shown in Figure 14, which contains 2 example routes. A Manhattan-style city layout is constructed with unidirectional roads between intersections. These intersections contain traffic lights, which toggle after a fixed delay. Every road segment contains either a residential building (source) or a commercial building (sink). Each road segment is part of a district, which is the atomic entity used in migrations. A district can be either residential or commercial, determining which type of buildings is constructed on its road segments. The route is pre-computed, and thus not part of simulation execution time.

The communication between different road segments is shown in Figure 13 and is part of the specification [20]. Residential buildings are generators and commercial buildings are collectors. At an intersection, queries are forwarded to the destination if the traffic light is green, or are immediately rejected if the traffic light is red. As soon as the traffic light becomes green again, all previously rejected queries are forwarded immediately.

While this is a realistic model, there are some disadvantages to distributed simulation:

1. Atomic models have almost no computation, as they only compute the new velocity. This is a simple formula, not warranting distribution or parallelization. Code is therefore added to make the computational load in the transition function configurable.
2. The state of the models is relatively complex, as it contains cars, queued queries, processing queries, acknowledgments, and so on. State saving thus imposes a significant simulation overhead. State saving overhead takes even longer than the transition functions in many cases.
3. Queries are answered almost instantly, thus lookahead is very small. This makes this a bad model for distribution, and certainly for time warp, as each message likely causes a rollback.

Distributed simulation now proves slower than sequential simulation, so our results only show absolute execution times. This observation does not invalidate the results observed concerning activity, as we merely compare different activity measurement approaches. Some models require distributed simulation, as they are simply too large to store in the memory of a single core. In that case, distributed simulation can become slower than sequential simulation without losing its value. Nonetheless, we still want to do the distributed simulation as fast as possible.

Activity is used to migrate districts between different cores during simulation. Intuitively, activity can have an impact here as the cars (and thus the activity) move through the model. At the start of simulation, residential districts are active, whereas the commercial districts are inactive. Near the end of simulation, the commercial districts become active, whereas the residential districts become inactive, as all traffic has shifted from the residential to the commercial districts.

Figure 15 shows the results for a distributed simulation using 5 cores. Four measurement methods are compared:

1. **No migration.** The basic case without any activity information, and thus no load balancing at all.
2. **Activity Tracking.** The wall clock time spent in the transition functions is used for activity. Migrations balance the time spent, by migrating districts between cores. Apart from the domain-specific migration (*i.e.*, migrating at district-granularity), this is identical to normal load balancing.
3. **Activity in state.** The number of cars in a district is used as activity metric. Migrations balance out the number of cars, which is a domain-specific measure.
4. **Activity in state prediction.** The number of cars in a district is used as activity metric. But instead of using this value as-is, we predict that 20%[*] of the cars exits the district and enters the next district. In commercial sections, we further assume that there is a chance that the car has arrived at its destination and is subsequently removed from the simulation. This prediction

[*]This value was obtained empirically and requires some tuning depending on the horizon.

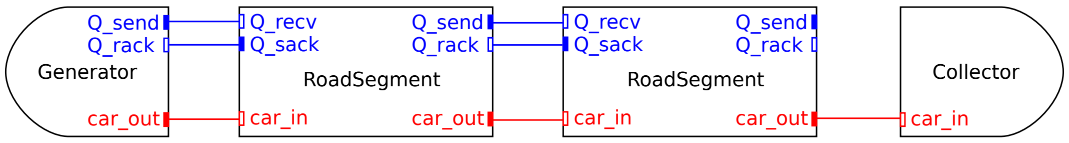

Figure 13. A road stretch.

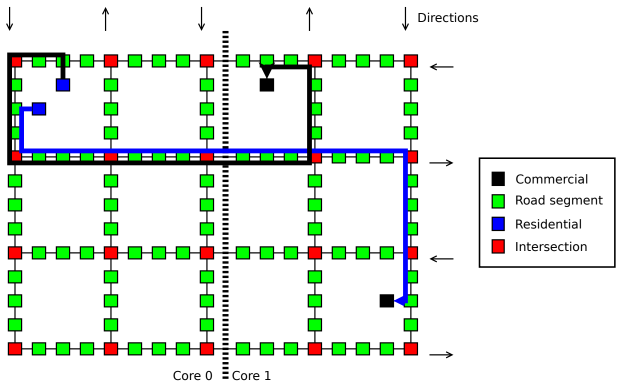

Figure 14. Example city layout model for 2 cars.

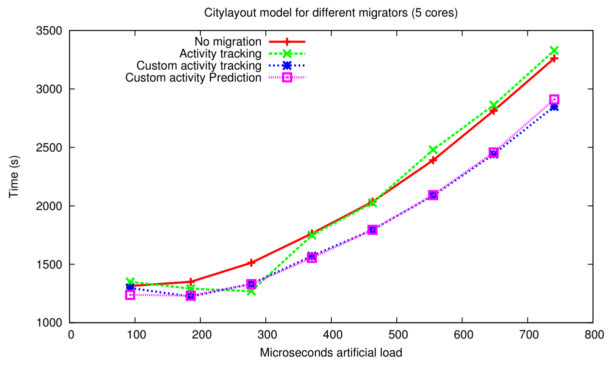

Figure 15. Activity migration city layout benchmark.

is at a much higher level of abstraction than the DEVS model, and much faster to compute. Nonetheless, results offer a fair indication of the future.

From these results, it is clear that both activity in state methods are always faster than no migration. Both perform nearly identical and their difference is negligible. Prediction is unable to exploit any information that can really make a difference: activity moves too slow throughout the model, making the prediction almost identical to the measured values. Only for low computational load can we see that prediction is marginally faster, though this is neglible.

The difference between activity tracking and activity in state is an important observation. As computational load is increased, the horizon becomes smaller. The horizon might become so small that a model only transitions a few times, if at all, making the measured activity statistically insignificant. This is the problem mentioned as a disadvantage of activity tracking. While increasing the horizon alleviates this problem, it causes slower reaction times. Activity in state methods are invulnerable to this problem. If the horizon is sufficiently large, such as for up to 300 microseconds load, activity tracking performance is similar to the other activity methods. On average, the performance improvement due to the use of (domain-specific) activity is around 5-10% for this model.

Related work

We have used the *PythonPDEVS* [25, 27] simulator as an example implementation of our approach.

PythonPDEVS supports the simulation of the Classic DEVS [33], Parallel DEVS [4], and Dynamic Structure DEVS [1] formalisms, each of which can benefit from activity-awareness. Several other simulators for these formalisms, or a subset thereof, exist, such as *adevs* [19], *vle* [22], *DEVS-Suite* [10], *PowerDEVS* [2], and *X-S-Y* [7]. Out of these simulators, none support distributed Parallel DEVS simulation [28]. While *adevs* offers parallel simulation using conservative synchronization, migration and allocation are not supported. Consequently, there is no opportunity to use activity for migration or allocation either.

Nonetheless, their schedulers can still be optimized to use activity information. For example, *adevs* and *vle* already filter out inactive models. This is equivalent to our simple *activity scheduler* presented in Section Activity Scheduler. They do not, however, support a polymorphic scheduler, nor can they import user-defined schedulers (potentially using activity). Neither supports domain-specific extensions to activity.

A *polymorphic scheduler* is implemented in the *Meijin++* [18] tool. However, this tool does not offer (Parallel) DEVS simulation, nor does it explicitly allow the user to chose the underlying data structures and the threshold parameters.

DEVSimPy [3], based on a modified version of *PythonPDEVS*, also has an activity tracking plugin [12, 23]. This plugin only visualizes measured activity, and is not used for load balancing or performance optimization. Modellers must manually use this information to optimize their model.

A frequently used example of activity is for the simulation of fire spread models, as used by [15, 16, 21]. In our case, this only exploits qualitative activity, as a cell is either burning (*active*) or not (*inactive*). Use cases for quantitative activity are more complex, such as the asynchronous electrical machine used in [12].

We have mainly touched the *computational resource usage*-aspect of activity, as our intention was to reduce simulation execution time. This was also the main focus of [12, 23]. Other directions could be *memory resource usage*, *energy consumption* [6], or even completely different notions [17].

A comparison between the terms seen in the literature and ours was previously made in Section .

Conclusions

We have shown that activity can give significant speedups for a wide variety of models. Three simulator components were extended with the notion of activity.

For the scheduler, it becomes possible to achieve a lower complexity than even the most efficient (static, activity-unaware) scheduler. These scheduler optimizations are compatible with all other features, in sequential and distributed simulation.

In distributed simulation, activity can be used to find a good initial allocation (with the allocator), or for optimizing this allocation at runtime using load balancing (with the migrator).

While these components are also usable without activity, activity makes them more flexible and dynamic. The components also require less user intervention, as soon as the domain-specific code is written. Activity is therefore not inherently linked to the possibilities for improved performance, though makes it easier to exploit.

Our approach enables users to increase simulation performance by providing (optional) domain-specific hints about the model being simulated.

This user information, however, can be wrong, depending on the skills of the user. Luckily, simulation can cope with wrong information, at the cost of lower performance.

It would be ideal to include such activity-based components in domain-specific tools or translators, which automatically generate DEVS models. Tool builders can then include activity hints in their generated models, as they have the necessary domain knowledge. As soon as the model and its activity-based extensions are created, no further user intervention is required. The end-user thus gets improved performance without any additional effort.

Future work is possible in several directions. First, other aspects of the simulator might also be able to profit from the use of activity. Further analysis of the algorithms might yield additional opportunities for activity-awareness. Second, determining statistical relevance of the measured activity values can potentially limit or event prevent operations based on unreliable activity measurements caused by a too small horizon. Third, our approach still has the problem that no information is known about activity at the start of the simulation. We tried to overcome this problem by using a small profiling run in the allocator. Static analysis of the model, such as in [12], might be possible to help in analyzing the statistical significance of the obtained horizon, and to aid in obtaining a decent initial allocation without the need for a profiling run. Fourth, domain-specific extensions for quantitative external activity are to be considered. Fifth, a more detailed analysis of the trade-offs between increasing performance and requiring more domain knowledge can help determine the ideal level of domain knowledge required. This trade-off is highly domain-dependent and also depends on the users and how they have encoded the solution.

Acknowledgement

This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO). Partial support by the Flanders Make strategic research centre for the manufacturing industry is also gratefully acknowledged.

References

[1] Fernando J. Barros. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation*, 7:501–515, 1997.

[2] Federico Bergero and Ernesto Kofman. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation*, 87:113–132, 2011.

[3] Laurent Capocchi, Jean-François Santucci, Bastien Poggi, and C. Nicolai. DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*, pages 170–175, 2011.

[4] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, WSC '94, pages 716–722, San Diego, CA, USA, 1994. SCS.

[5] Jan Himmelspach and Adelinde M. Uhrmacher. Sequential processing of PDEVS models. In *Proceedings of the 3rd European Modeling & Simulation Symposium*, pages 239–244, 2006.

[6] Xiaolin Hu and Bernard P. Zeigler. Linking information and energy - activity-based energy-aware information processing. *Simulation*, 89(4):435–450, 2013.

[7] Moon Ho Hwang. X-S-Y. `https://code.google.com/p/x-s-y/`, 2012.

[8] David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[9] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, 13(3):135–154, 1996.

[10] Sungung Kim, Hessam S. Sarjoughian, and Vignesh Elamvazhuthi. DEVS-Suite: a simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the Spring Simulation Conference*, 2009.

[11] Will Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. *SIGMETRICS Performance Evaluation*, 14(1):54–69, May 1986.

[12] Alexandre Muzy, Laurent Capocchi, and Jean François Santucci. Using activity metrics for DEVS simulation profiling. In *Activity-Based Modeling and Simulation*, 2014.

[13] Alexandre Muzy, Eric Innocenti, Antoine Aiello, Jean-François Santucci, and Gabriel Wainer. Cell-DEVS quantization techniques in a fire spreading application. In *Proceedings of the Winter Simulation Conference*, pages 542 – 549, 2002.

[14] Alexandre Muzy, Eric Innocenti, Antoine Aiello, Jean-François Santucci, and Gabriel Wainer. Specification of discrete event models for fire spreading. *Simulation*, 81(2):103–117, 2005.

[15] Alexandre Muzy, Rajanikanth Jammalamadaka, Bernard P. Zeigler, and James J. Nutaro. The activity-tracking paradigm in discrete-event modeling and simulation: The case of spatially continuous distributed systems. *Simulation*, 87(5):449–464, 2011.

[16] Alexandre Muzy, Luc Touraille, Hans Vangheluwe, Olivier Michel, David Hill, and Mamadou Traoré. Activity regions in discrete-event systems. In *SpringSim/TMS-DEVS*, pages 176–182. SCS, April 2010.

[17] Alexandre Muzy, Franck Varenne, Bernard Zeigler, Jonathan Caux, Patrick Coquillard, Luc Touraille, Dominique Prunetti, Philippe Caillou, Olivier Michel, and David Hill. Refounding of the activity concept? towards a federative paradigm for modeling and simulation. *Simulation*, 89(2):156–177, 2013.

[18] Patrick Nicolas. Meijin++, reference manual, 1991.

[19] James Nutaro. Adevs. `http://www.ornl.gov/~1qn/adevs/`, 2015.

[20] Ernesto Posse. *Modelling and simulation of dynamic structure discrete-event systems*. PhD thesis, School of Computer Science, McGill University, October 2008.

[21] Martin Potier, Antoine Spicher, and Olivier Michel. Topological computation of activity regions. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 337–342. ACM, 2013.

[22] Gauthier Quesnel, Raphaël Duboz, Éric Ramat, and Mamadou Traoré. Vle: a multimodeling and simulation environment. In *Proceedings of the 2007 summer computer simulation conference*, SCSC, pages 367–374, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[23] Jean François Santucci and Laurent Capocchi. Implementation and analysis of DEVS activity-tracking with DEVSimPy. In *Activity-Based Modeling and Simulation*, 2012.

[24] Yentl Van Tendeloo. Activity-aware DEVS simulation. Master's thesis, University of Antwerp, Antwerp, Belgium, 2014.

[25] Yentl Van Tendeloo and Hans Vangheluwe. The Modular Architecture of the Python(P)DEVS Simulation Kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*, TMS/DEVS '14, part of the Spring Simulation Multi-Conference, pages 387 – 392. Society for Computer Simulation International, 2014.

[26] Yentl Van Tendeloo and Hans Vangheluwe. PythonPDEVS: a Distributed Parallel DEVS simulator. TMS/DEVS '15, part of the Spring Simulation Multi-Conference, pages 844–851, 2015.

[27] Yentl Van Tendeloo and Hans Vangheluwe. An overview of PythonPDEVS. In Collectif Workshop RED, editor, *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*, pages 59 – 66. Éditions Cépaduès, April 2016.

[28] Yentl Van Tendeloo and Hans Vangheluwe. An evaluation of DEVS simulation tools. *SIMULATION*, 93(2):103–121, 2017.

[29] Hans Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134, 2000.

[30] Gabriel Wainer. Applying Cell-DEVS methodology for modeling the environment. *Simulation*, 82(10):635–660, 2006.

[31] Gabriel Wainer, Ezequiel Glinsky, and Marcelo Gutierrez-Alcaraz. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *SIMULATION*, 87(7):555–580, 2011.

[32] Gabriel Wainer and Bernard Zeigler. Experimental results of timed Cell-DEVS quantization. In *Proceedings of AIS Artificial Intelligence, Simulation and Planning*, 2000.

[33] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, second edition, 2000.