Research Internship II: Distributed and parallel DEVS simulation

Yentl Van Tendeloo
Supervisor: Hans Vangheluwe

June 24, 2013

Abstract

In this report, we will continue to improve the performance of the PythonDEVS simulator. In our previous work[45], only the algorithms were changed in an attempt to obtain substantial speedups. Whereas now, we will use distributed simulation to gain even bigger speedups. Due to the complex nature of distributed simulation, the first chapter will provide an introduction to the most popular distributed simulation methodologies. In the second chapter we will direct our attention to the actual algorithms and features present in our new version of PythonPDEVS, together with a rationale for each decision. In the third and final chapter, the actual implementation will be discussed. Apart from these implementation details, problems, solutions and benchmarks, we also provide an updated *'Programming reference'* for the much updated API.

Contents

1 Distributed and parallel simulation

This chapter will provide a summary of several different possibilities for distributed simulation of DEVS models. It provides the necessary background to understand several design decisions and features that were made in PythonPDEVS, which will be explained in the following chapters. As PythonPDEVS is an *optimistic* simulator, the main focus will be on optimistic simulation algorithms, as these will be actually implemented and referenced to. But to be able to defend this choice, conservative algorithms will also be discussed.
Most of this information is based on [16, 15], unless stated otherwise.
Even though there is a clear distinction between both methods, hybrid methods also exist. One of these methods is to perform conservative simulation, but allow for optimistic simulation to happen locally.

1.1 Conservative

Conservative synchronization algorithms will process events in timestamp order. If it cannot be guaranteed that an event is the next one, e.g. when there is a possibility that a remote event with a smaller timestamp may arrive, the simulation will block until the simulator is certain about the next event. It will thus avoid causality errors at the cost of blocking until it can be guaranteed that no causality errors can occur by processing the next event.
This section will provide the general idea of conservative simulation and present several algorithms and problems that occur while using these algorithms. Some more advanced aspects, like *bounded lag* and *conditional information* will not be discussed here.

1.1.1 Local causality constraint

Definition 1.1.1. A discrete event simulation is said to obey the *local causality constraint* if and only if it processes events in increasing time stamp order.

Conservative simulation is than defined as a simulation algorithm in which it is guaranteed that the local causality constraint will always be respected. So all events are always processed in timestamp order.
Note that the local causality constraint alone is not sufficient to guarantee a simulation run that is exactly equal to the sequential simulation of the same model. The additional constraint is that simultaneous events (events with the same timestamp) be processed in the same order[1].
Even though adherence to these two constraints is sufficient to satisfy correct simulation, it is clearly not a necessity to prevent causality errors. Multiple LPs in the same model may be independent of one another, thus a (slightly) different processing order does not necessarily cause causality errors.

1.1.2 Basic algorithm

To adhere to this local causality constraint, each LP has a FIFO queue which will queue all incoming external messages. Thus a message that is received from LP_A means that LP_A has progressed up to the time that is present in the incoming message. If

[1] Some other problems still occur, though they are mostly at a lower abstraction level, e.g. floating point operations with different precisions on different machines

an LP has at least one message in every queue, the simulator knows all about its influencees and thus has the guarantee that no messages before the minimal time of all queues will arrive. This gives rise to algorithm 1, which is the main simulation loop of a (naive) conservative LP.
Because the previous algorithm clearly mentions that *blocking* occurs as soon as no guarantee can be made about the processing

Algorithm 1 Basic synchronization algorithm

```
while simulation is not over do
    wait until each FIFO queue contains at least one message
    remove smallest time stamped message M from its FIFO queue
    clock ← timestamp of M
    process M
end while
```

order, it seems likely that deadlock can occur in specific situations. Indeed, it is possible that deadlock arises due to multiple LPs waiting for each other to guarantee their next timestamp, as presented in figure 1.1. It is possible that there is a loop which causes several LPs to wait for each other, in case all of them still have an empty queue and are thus uncertain about the state of the other LPs. Due to this uncertainty, they will wait until they receive a message from their influencer, though this might be deadlocked too.
Deadlocks with this algorithm can occur frequently in cases where there are few (unprocessed) messages in the queues, while there are a lot of links. Several solutions are presented in the following subsections.

Deadlock avoidance: null messages

Definition 1.1.2. If a logical process at simulation time T can only schedule new events with timestamp of at least $T+L$, then L is referred to as the *lookahead* for the logical process. This means that, if an LP is at simulation time T, all connected LPs can be sure that no event will be received from that LP before simulation time $T+L$.

Using information about the *lookahead* of an LP, a logical solution to the deadlock problem is the Chandy/Misra/Bryant algorithm, which introduces *null messages*. Null messages are messages without any actual simulation content and only function as a notification from the sender to the receiver that a specific simulation time has been reached. Such messages will artificially fill up the FIFO queue and subsequently allow the algorithm to progress. After the processing of an event, this raises the need to send a null message to all connected LPs. Since these messages are used to make a guarantee, it is possible to send a message with $T+L$, with L being the lookahead, instead of T and thus the other nodes will be able to advance even further into the future.
When the null message itself is processed (i.e. when it is the first element of all queues), the LP will only have to update its clock and resend null messages with a higher timestamp. This provides a solution for a deadlock in most, but not all, cases. One of the problems that is not solved is a deadlock when the lookahead is equal to 0, as it will now keep sending exactly the same null messages without any advancement of the simulation time.
Another performance problem might occur if the lookahead is very small compared to the actual time increase required to allow simulation. In such a case, the LPs will have to keep sending null messages until the deadlock is solved. Actually, this case becomes very much like normal *discrete time* simulation as we sometimes have to progress with a fixed timestep if the difference becomes too big for the lookahead. To make things worse, these messages must traverse the network, possibly causing high latency until the next message is actually processed.
Often it is redundant to send null messages, for example when there are still unprocessed messages in the influencee's FIFO queue. In such cases it could be advantageous to use a *demand-driven* approach for sending such null messages. And while this can significantly reduce the number of messages on the network and the respective overhead, it causes higher delays in situations where a null message has to be requested and thus possibly a longer time of waiting.
This gives the observation that the performance of the *null message algorithm* is dependent on the size of the lookahead. Therefore, this algorithm is only recommended in cases where the lookahead is big enough. Sadly, the lookahead itself is dependent on the model and thus it doesn't provide a general solution. Several models are even impossible to simulate using this algorithm, specifically models with a lookahead loop that is equal to zero. It is also possible for lookahead to vary during the simulation of the model, though it cannot simply decrease instantaneously due to previous *promises* to the influencees.

Detection and recovery

Another solution to the deadlock problem is to allow deadlocks to happen, but provide a mechanism to detect them and recover from this situation. The main advantage is that it doesn't require null messages to be sent and therefore lowers the message overhead (both sending, receiving and processing). If no problem occurs, there is also no need to perform such synchronization,

Algorithm 2 Synchronization algorithm using null messages

while simulation is not over **do**
  wait until each FIFO contains at least one message
  remove smallest time stamped message M from its FIFO
  clock ← timestamp of M
  process M
  send null message to neighboring LPs with time stamp equal to lower bound on time stamp of future messages (clock plus lookahead)
end while

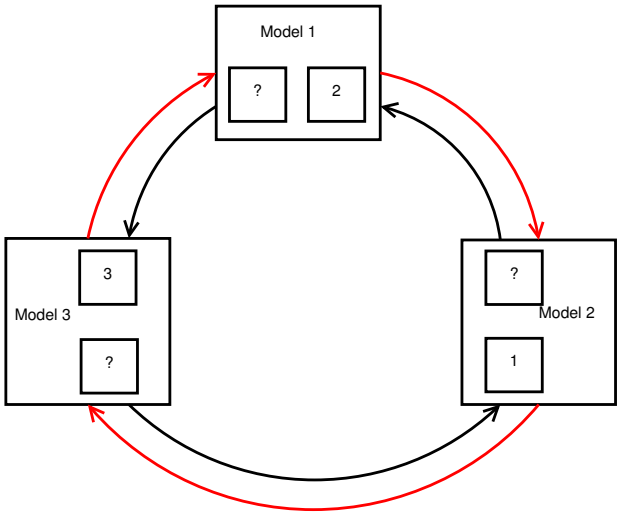

Figure 1.1: Deadlock using algorithm 1, the cycle is indicated in red

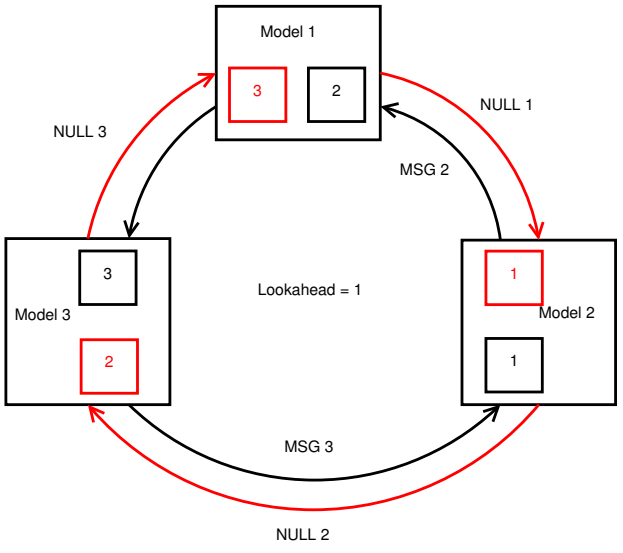

Figure 1.2: Null message sent by *Model 1* is able to break the deadlock from figure 1.1

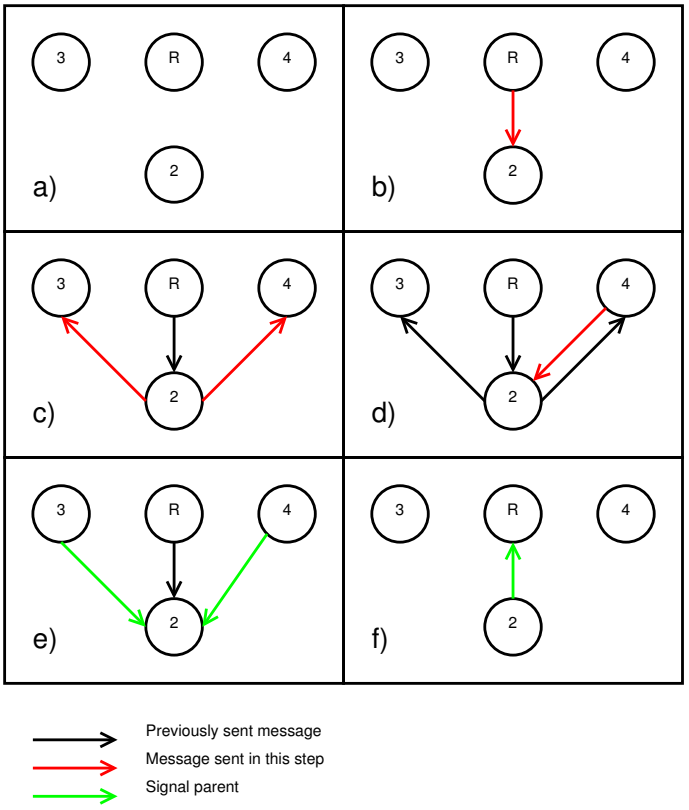

Figure 1.3: Example of deadlock detection

which allows for a low-overhead simulation. A disadvantage is that such detection can only happen as soon as the deadlock has already occurred and the simulation is already halted. Even before the deadlock actually happens, several LPs are probably already idling, which is undetectable with this algorithm. In worst case situations, it is possible that the simulation comes to a slow halt, where most of the LPs are waiting most of the time.
The algorithm for this method is the same as algorithm 1, though an extra process is added to the *controller*. This *controller* will be responsible to coordinate the additional algorithm in algorithm 3. This algorithm still has several vague parts, like determining whether or not deadlock has occurred and which messages will be safe to process. These details will be discussed in the next few paragraphs.

Algorithm 3 Deadlock recovery

```
if computation is deadlocked then
    controller sends messages to LPs to inform them of safe events
    for all LP that now has safe events do
        process the incoming events
        send out additional messages generated by the processed events
        signal parent as soon when deadlocked again
    end for
end if
```

Detection To be able to detect whether or not deadlock has occurred, we construct a *tree* with the *controller* as root node. Children in the tree will be models that have received messages from their parents and thus are able to process new events. As soon as the *controller* declares that messages are safe to process, the tree will be constructed while messages - and subsequently computation - spread throughout the network. As soon as an LP becomes deadlocked, it will remove itself from the tree and signal its parent about this removal. If the parent becomes a leaf in the tree and has no computations of its own, it will also be deadlocked and thus remove itself from the tree. The complete simulation is than deadlocked as soon as the *controller*, which was the root node, becomes a leaf node.
An example is shown in figure 1.3 and an implementation is mentioned in [16].

Recovery As soon as a deadlock is detected, it is still necessary to recover from them. As was mentioned in algorithm 3, this is done by having the *controller* send out messages which are safe to progress. The actual problem that remains is the question of *which messages are safe to process*. A straightforward solution is to select the message with the lowest timestamp in the complete simulation and mark it as *safe*. There is still a problem however, as it might be the case that some messages are still *in transit* and are therefore not taken into account when searching for the message with the lowest timestamp. This problem of so called *transient messages* will be further explored in section 1.1.4 and we will currently assume that no such messages exist.
To determine this lowest timestamped event, communication between all nodes is necessary, as we are trying to establish a global state. In order to prevent massive communication overhead and prevent the creation of a bottleneck at the controller, an algorithm to compute this safe message can use a spanning tree, where the lowest message is simply passed over the edges. At the end, the actual minimum will have arrived at the root node, which can then broadcast this value, possibly using the same spanning tree again.
While this algorithm is relatively easy, it has the major drawback that it only allows *one* message to become safe and thus there is the possibly of recreating a deadlock after that message is processed. In this case, lookahead information could be used in order to declare multiple messages as *safe* and thus allow more messages to be processed, increasing concurrency.
Further optimizations for these recovery algorithms are possible, like working on only a subset of all LPs and thus not requiring the complete simulation to be deadlocked before declaring the next event as safe.

Evaluation The main advantage of *detection and recovery* over other algorithms, like null messages, is that they allow for (near) zero overhead in the simulation if the model rarely deadlocks. Additionally, cycles of zero lookahead are not a problem as was the case with null message. Such cycles do lower concurrency, as it causes more deadlocks and a lower number of messages can be declared safe in a single step, so the performance is again very tied to the amount of lookahead that can be guaranteed.

1.1.3 Synchronous execution

Another possibility is to use *barriers* for synchronization instead of relying on deadlock to happen. This way, simulation will always be halted explicitly and while this causes a certain overhead, it avoids the situations where a deadlock is *in creation* and where concurrency keeps lowering until *eventually* a deadlock does occur that can be detected and solved.
Several algorithms for such barriers exist, which will not be discussed in detail as they are relatively general. The most common kind of barriers are:

- **Centralized barriers**: a global controller is contacted as soon as a barrier is reached and simulation is continued as soon as a return is received. Clearly, this doesn't scale well.
- **Tree barriers**: all LPs are ordered in a balanced tree. As soon as the barrier is reached locally *and* in all of its children, the LP will notify its parent that it has reached the barrier.
- **Butterfly barriers**: LPs are ordered in pairs and they achieve a barrier together. After each step, the synchronized pairs are merged into a new group, which will then again try to achieve a barrier with another such group.

1.1.4 Transient messages

Definition 1.1.3. A transient message is a message that has been sent but not yet received at the destination. The message is effectively still "in the network".

These messages become problematic due to the use of asynchronous messages, as the sending LP has finished sending the message (and doesn't require to be notified), while the receiving LP has not yet received the message and therefore cannot take its timestamp into account. Asynchronous messages are helpful to increase performance, as the sending process will not block until the receiving process to acknowledge the receivement of the message.
Several algorithms exist to allow such transient messages to be detected and to make sure that no transient messages exist in the network before progressing. Such messages should not be present in the network when determining the global state, as they will still affect the state as soon as they are received. An example of what might go wrong if they are neglected is shown in figure 1.4. Therefore, either the sender or the receiver should take the contents of the message into account.
The solution to this problem consists of using message counters: LPs that try to synchronize will notify each other of the number of messages that they have sent and have received. Clearly, no transient messages are present if all LPs together have sent and received exactly the same number of messages. [16] presents several applications of such an algorithm in the different barrier algorithms.

1.1.5 Distance

Definition 1.1.4. The *distance* between two LPs defines a lower bound on the amount of simulated time that must elapse for an event in one process to affect another.

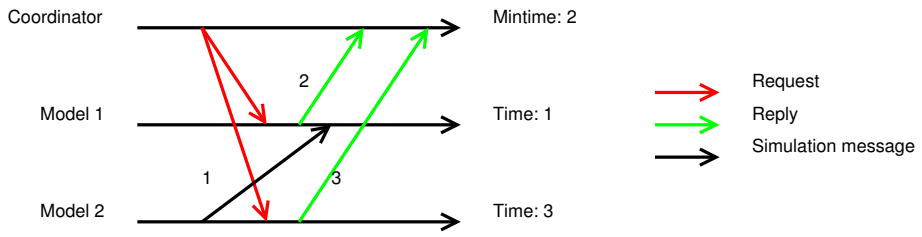

Figure 1.4: Message with timestamp 1 is transient and causes a wrong minimal time in the root coordinator

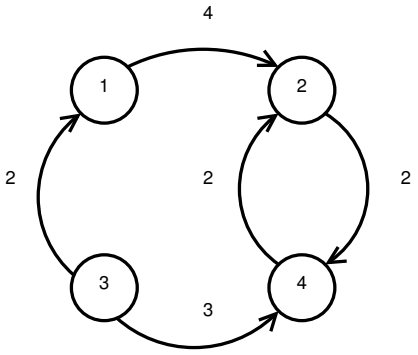

Figure 1.5: Example network, arcs are annotated with the associated lookahead

Information about the distance can be used to achieve better bounds on which events are safe to process, at the cost of knowing (and analyzing) the network topology. The distance can be computed by taking the minimum path in terms of lookahead, from LP_A to LP_B. Such information can be used to exploit information about the network topology and combine this information with the lookahead that was previously determined. Furthermore, it is possible to have an infinite distance between two LPs, in which case they do not have to take each other into account when determining whether or not an event is safe to process.
Using this information, safe events are defined as in definition 1.1.5.

Definition 1.1.5. An event E in LP_A is said to be *safe* if it is not possible for a new event to be generated and sent to LP_A that contains a time stamp smaller than Es time stamp.

The main drawback of the *distance* is that it requires all LPs to communicate with each other, in order to exchange their distances. If lookaheads can change dynamically during the simulation, distances will also have to be recomputed. Some solutions for this problem exist and are mentioned in [16].
The example provided in figure 1.5 has the distances mentioned in table 1.1. For example, a message in 3 will always be safe, because the distance is always ∞. If a message in 3 with timestamp t is present, then all messages in 2 up to $t+5$ will be safe (in the assumption that no messages are present in 1 and 4).

1.1.6 Repeatability

In many cases, repeatable simulation is important in order to verify results that were previously obtained. Some of these problems are related to heterogeneous systems, like floating point round-off errors and are out of the scope of this research internship. Other problems, which are more concerned with the actual simulation algorithm, are in scope and will be considered here. Thus the only problem that may arise is the different ordering of messages with *exactly* the same message timestamp. Whereas in sequential simulation this does not pose a problem, it is a source of race conditions in a distributed setting. For example an LP that receives messages from different LPs and the message has the same timestamp. The order in which these are processed is important, as the network will introduce nondeterminism in the order in which the messages are actually received.

	1	2	3	4
1	∞	4	∞	6
2	∞	4	∞	2
3	2	5	∞	3
4	∞	2	∞	4

Table 1.1: The distance matrix associated with the model in figure 1.5

Several solutions are possible:

- **Hidden time stamps**: the time stamp is extended with several extra fields, called *age* and *id*, to provide a unique ordering. This way, no two timestamps are *exactly equal* as we also compare on the enclosed age and id.
- **Priority numbers**: a custom ordering scheme is implemented and messages are assigned with a priority to denote the order in which they should be processed. This approach allows for a semantically meaningful decision.
- **Receiver-specified order**: the receiver is notified that it has received all messages and can then use a sorting algorithm to impose an order. This is a very flexible solution, though it has a problem if zero lookahead is allowed.

1.1.7 Performance

The actual performance of conservative simulation is dependent on the type of model and the associated lookahead, but also on the chosen simulation algorithm.
Clearly, simulation using the naive algorithm (without any checks for deadlocks) is very efficient, as it doesn't have any overhead for models that are fit for the algorithm. On the other hand, most models will simply deadlock after a certain period of time, which is unacceptable.
Null messages that happen on-demand could be ideal for models that deadlock very infrequently and that have a high enough lookahead. Deadlock detection and recovery is also efficient in such situations.
Barrier synchronization algorithms always block after a certain period of time, causing high overhead in situations where no deadlock would ever occur. On the other hand, models that have many deadlocks in several subcomponents could profit from this, as they do not need to wait for the complete simulation to deadlock, thus increasing concurrency in these situations.

1.1.8 Conclusion

The common goal of conservative synchronization algorithms is to ensure that the *local causality constraint* is respected and thus guarantee that messages are processed in timestamp order. If this is the case, in combination with repeatable input message ordering, it is guaranteed that the same results will be produced as in a sequential execution. Problems might arise with deadlock, which has several solutions depending on the model that has to be simulated. To allow for more concurrency, *lookahead* can be exploited which is again dependent on the model that will be simulated.
Performance is very much related to the kind of model, though high performance can be obtained in situations where the model has a high lookahead. On the other hand, a very small lookahead causes performance that is lower than sequential execution.
These limitations cause us to believe that conservative simulation is not recommended for general models, as there are several distinct algorithms, each of which have their own specific kind of models that can achieve high performance with them. Furthermore, the effect of lookahead is too big for the modeller to ignore and therefore the modeller should take the algorithm into consideration while modelling[15].

1.2 Optimistic

In contrast to *conservative* simulation, where only safe events are executed, *optimistic* simulation will process as many events as possible in the assumption that they are safe. While this completely avoids the overhead associated with determining whether or not a message is safe, it is possible for violations to happen. Such violations must then be discovered and rectified. The first and most well known optimistic simulation method is *time warp*[21], which will be discussed in this section. Other methods, like *lightweight time warp*[25, 26] are not elaborated on, as they are not implemented in PythonPDEVS.

1.2.1 Synchronization

An important aspect about *time warp* is that it does not use any additional communication between different LPs at all. It furthermore doesn't require the other LPs to send messages in timestamp order, nor does it require the network to deliver these messages in time stamp order. It even provides relatively simple support for models with zero lookahead, which is known to be problematic in several *conservative* simulation algorithms.
While no additional communication is really mandatory, some global communication is highly recommended in order to have some kind of knowledge about all other LPs. The message overhead of this synchronization is relatively small and should not incur a huge slowdown or cause the network to become a bottleneck. Additionally, this synchronization only happens sporadically and is often a user configurable configuration parameter.

1.2.2 Stragglers

Definition 1.2.1. A *straggler message* is a message with timestamp T_m that is received at simulation time T_n, with $T_m < T_n$. It is thus a *late* message.

Because no guarantees are made about the safety of messages before they are processed, there is always the possibility for

messages to arrive at a time before a previously processed time. This could mean that the simulation at a certain LP is already at simulation time 200, when suddenly a message with timestamp 100 is received. Such situations were impossible in *conservative* simulation, as the LP would then be cautious and assume that there is a possibility for such a message to be received even before progressing the local clock to time 100.
Should no straggler messages ever happen, optimistic simulation is (in theory) a perfect technique because each LP can just simulate at its own maximum pace without taking other LPs into account. Sadly, reality is different and *straggler messages* appear relatively frequently, again depending on the type of model. Whereas *conservative* algorithms were highly influenced by the amount of lookahead, *optimistic* algorithms are highly influenced by the number of stragglers that occur.

1.2.3 Rollback

Due to the possibility of straggler messages, the simulation algorithm might have to roll back to a previous time, right before the *local causality constraint* was broken. This means that if a straggler message with timestamp 100 is received, the *complete LP* will have to roll back to a time right before simulation time 100. This rolling back is thus effectively an *undo* operation on the LP, which must then:

- *reset* the state variables of all its models
- *unsend* all messages that were sent to other LPs after that timestamp
- *reprocess* all incoming messages that were already processed after the timestamp

Clearly, a lot of work that has already happened must be repeated. All computations that were done in the meantime thus become useless and must be discarded as they used old (and probably invalid) information. This isn't completely wasteful, as conservative simulation would have been blocking to make sure that the event is safe to process, whereas optimistic simulation would have been progressing, hoping that no straggler would arrive. The disadvantage is that this rollback operation itself will also have a certain cost and other LPs will also be notified of our violation and must respond accordingly.
It can be shown that progress is guaranteed, due to two observations:

1. Rollbacks always propagate into the simulated time future
2. The smallest timestamp computation will never be rolled back

Due to these two observations, it can be shown that there will always be some kind of progress, as some events can never be rolled back, thus creating progress. This means that deadlock is impossible, in contrast to conservative simulation, where deadlock could occur frequently.
In several situations, a straggler message does not necessarily have to cause a rollback, as long as the final result stays equal. A query message for example, will not have to perform a rollback of all computation that follows, because it will not perform any state change. All that is required is that the state at that time is *temporarily* put back, after which simulation can progress. This concept is defined as *lookback*[5, 6] and can be used in both optimistic as conservative simulation algorithms.

1.2.4 State saving

While rolling back doesn't seem to be that much of a problem, it requires these old states to be known. To make this possible, there is a need to save intermediate states, so that we can roll back to them. Furthermore, all messages that were received must also be stored after they were processed in order to allow them to be reused in the case that a rollback is required. Even all messages that get sent by the LP need to be saved somewhere, as we have to know where these messages must be invalidated in the case of rollback.
Clearly, all this state saving does require a lot of additional overhead due to the many memory operations that are necessary. In order to alleviate this problem for huge states, it is possible to use incremental state saving where only a delta is made between the previous version and the new version of the state. Such a delta requires more compute time to determine the difference, though it can save a lot of memory in cases where a big part of the state stays static is left unchanged in most of the transition functions. Of course, hybrid methods are also possible.
Another possibility to avoid state saving is to be able to perform reverse computation, though this requires more computational power.

1.2.5 Anti-messages

In order to *unsend* messages, the old messages that have to be unsent need to be known. These messages can be altered to signal that they are actually *anti-messages* and thus that they cancel out the corresponding message. Several situations are possible when an anti-message is received:

- **The corresponding message is not yet received**
  In this case, the anti-message effectively hopped over the actual message. The anti-message should be queued and as soon as the corresponding message arrives, both of them should be deleted.
- **The corresponding message is not yet processed**
  If the corresponding message is still in the input queue, the anti-message will cause the removal of this message from the queue, in which both of them get deleted.
- **The corresponding message is already processed**
  This is the worst case, as now a rollback to the time before processing this message is required. This kind of rollback is called *secondary rollback*, as it is caused by another *primary* rollback from receiving a straggler message. After the rollback is completed, this situation is reduced to the situation where the corresponding message is not yet processed, so both messages now get deleted.

To save some memory, the anti-message can be a stripped down version, as it does not need to contain the complete message, only the *headers* such as the timestamp, destination, ... The actual contents can be dropped.

1.2.6 Age

Repeatability has the same importance in optimistic simulation as it had in conservative simulation. As was the case in conservative simulation, this problem can easily be solved by adding an *age* field to the timestamp. Such a solution has several other advantages, like allowing zero lookahead to happen. Should no age field be used, a zero lookahead loop would make it possible to have anti-messages that go in a loop and effectively keep rollbacks going. Because in our previous observation we mentioned that no deadlock occurs if rollbacks *always* propagate into the simulated future, we need to guarantee that time will actually advance. In the case of zero lookahead, this is not always guaranteed due to the possibility of a loop.
Therefore, introducing the age field will provide a small difference between these *supposedly equal* timestamps, thus preventing an event from rolling back itself.
As was seen in the conservative algorithms, only an age field does not guarantee repeatability, as it also requires an ID to order those events with an equal timestamp and age.

1.2.7 Synchronization

Previously it was mentioned that LPs do not need to use any additional synchronization messages. While this is true, some synchronization is often desired. The main reason for synchronization would be to determine the lower bound of incoming messages. This slightly resembles the ideas in conservative algorithms, though this lower bound is not used to determine which events are safe to process. Because we need to save a lot of past data in our simulation execution, memory could become an important limitation to how long a simulation can last. The size of such queues also determine the efficiency of simulation, because these queues are still data structures of which the operations depend on its size. These are all good reasons to erase as many unneeded events as possible.
It is clear that if all LPs have progressed up to time T_y, that all data that was saved at T_x, with $T_x < T_y$, is no longer needed[2]. This is applicable for both input messages, states and output messages. Clearly, if all LPs are at or after time T_y, none of them will send a message or anti-message with a timestamp before T_y. Since only messages and anti-messages are used for communication (and determining whether or not to rollback), it is effectively impossible for a process to roll back to a time before the lower bound of all LPs. Since no rollback can occur to such a time, there is no longer a need for the saved states (they are now *certain*), the input messages (they will never have to be reused again) or the output messages (since no anti-messages of them will have to be sent anymore). This process of removing this outdated information is called *fossil collection* and is one of the main reasons why synchronization is desired. How such a lower bound can be computed is explained in section 1.2.8.
Another use of the lower bound can be used to perform irreversible actions. While simple state changes and messages can easily be undone with the previously mentioned algorithms, some operations cannot be undone, mainly I/O operations like writing to file or tracing of the simulation. As these are irreversible, they should only be executed as soon as it is *certain* that they will not need to be undone and therefore they are only executed as soon as the lower bound has passed over the time associated with these events. Not only I/O is limited in this way, also exception handling should only be executed as soon as it is certain that this exception will occur, otherwise this exception would not occur in a sequential simulation and therefore be an artifact from the parallel simulation.
Another use of synchronization would be determining whether or not to end the simulation.

1.2.8 Global Virtual Time

Definition 1.2.2. Global Virtual Time (GVT) at wall-clock time T during the execution of a Time Warp simulation is defined as the minimum time stamp among all unprocessed and partially processed messages and anti-messages in the system at wall-clock

[2] We assume the absence of transient messages

time T.

Computing the GVT is not as simple as it may sound. Many factors need to be taken into account, with transient messages being the most important problem to take into account. Furthermore, LPs are allowed to advance their clock if this algorithm happens asynchronously. The most basic solutions to compute the GVT are blocking algorithms, meaning that simulation at all LPs will halt while computation is in progress. This closely resembles the *barrier* algorithms discussed in section 1.1.3.
Sadly, these basic solutions often don't take into account the *simultaneous reporting problem*, which arises because LPs don't send their minimal times at exactly the same time, making it possible for a message to slip between the cracks and thus be undetected as was the case in figure 1.4.
An algorithm that allows for asynchronous message passing and that does not block is called *Mattern's algorithm* and is discussed next. Other algorithms include Samadi's algorithm[35] and Time Quantum GVT[7].

Mattern's algorithm

Mattern's algorithm was defined in [27] and proposes a general way of finding the GVT (or at least an approximation). It supports many different network structures, like a ring or a tree, though [27] only gives an example of a ring variant. Furthermore, it also offers several possible optimizations, depending on the amount of information that is available to the LPs (e.g. knowing which LP must still receive a transient message).

Algorithm Mattern's algorithm is presented in algorithms 4, 5, 6 and 7, where each of these algorithms is used at a specific time. The algorithm will be called as soon as a message is sent or received, though it only incurs a small overhead as it is simply updating a small counter.
The algorithm is based on two colors, with every LP having either one of them. All LPs start in color *white*, where they record the number of messages they send and receive. As soon as a control message is received, their color switches to *red*. It is clear that all *white* messages (thus messages *sent* when the LP had color *white*), will be taken into account in the minimal time calculation and in the counter. As soon as the LP became *red*, it will lower its minimal time as soon as an earlier message is received.
As soon as all *white* messages are received, no *transient* message is present *that can lower the GVT*. This is an important consideration, since the actual LPs will continue processing and will thus keep sending messages that are temporarily transient. Compared to blocking GVT algorithms, where there are actually no *transient* messages at all when the GVT is determined.
If after the first round there are no *white* messages in transit, meaning that there were no *transient* messages to start with, the algorithm is already finished and actually reduces to a simple ring algorithm. In the case where some *white* messages are still in transit, a second round is necessary to wait for these *transient* messages. A *red* message has no meaning in the algorithm because it doesn't matter whether or not they are transient. As soon as an LP is *red*, it should still take into account the actual sending of the message, as it is possible that an earlier message is sent in case a rollback happened.

Advantages The use of this algorithm has several advantages, most of them being performance related. First of all, the GVT calculation can happen asynchronously. This is due to the use of different colors, where *red* symbolizes LPs that should otherwise be blocked. As they are still allowed to progress, we should take into account the fact that they might rollback and thus update the minimal time if necessary.
Another optimization that is present in the aforementioned algorithm is the use of only 2 passes over the complete network. This is possible because it is known which LPs still have to receive messages (and how many of them). This prevents us from waiting for messages at LPs that have no more transient *white* messages destined towards them. Also, it prevents us from going over the network indefinitely until all messages have arrived. Even though this is an important optimization, this is not a necessary part of *Mattern's algorithm* because in some situations such additional data management is infeasible or requires too many changes.

Algorithm 4 Mattern's algorithm when sending a message from P_i to P_j

```
send the actual message to P_j
if color = white then
    V[j] ← V[j] + 1
else
    t_min ← min(t_min, timestamp)
end if
```

Example An example is given in figure 1.6. These messages are the only possibilities in terms of *from color* and *to color*. For example a *white* message being sent in the first white zone, will never be received in the second *white* zone, since the intermediate *red* zone will only stop as soon as all these *white* messages are received.

1. **White 1 to White 1**: an ordinary message being sent and received before the algorithm starts, clearly LP_3 will use the

Algorithm 5 Mattern's algorithm when receiving a message at P_i

if msg_color = white **then**
  $V[i] \leftarrow V[i] - 1$
end if
process the message and use the enclosed timestamp

Algorithm 6 Mattern's algorithm when receiving a control message $< m_{clock}, m_{send}, count >$ in P_i

if color = white **then**
  $t_{min} \leftarrow \infty$
  $color \leftarrow red$
end if
wait until $V[i] + count[i] \leq 0$
send $< min(m_{clock}, T), min(m_{send}, t_{min}), V + count >$ to $P_{(imodn)+1}$
$V \leftarrow 0$

message's timestamp when determining the minimal time.

2. **White 1 to Red**: this message is a typical *transient* message. LP_2 will wait for this message before being allowed to turn *white* and will thus be able to lower its minimal time.

3. **Red to White 1**: this message has no meaning in terms of the GVT calculation. Either LP_2 has already saved a lower timestamp (in case no rollback happens in between), or LP_3 will take the timestamp into account when it transitions to *red*.

4. **Red to Red**: this message has no special meaning for the calculation, though it could be possible that LP_2 got rolled back in the meantime, thus LP_1 will update its minimal timestamp if necessary.

5. **White 2 to Red**: in the original algorithm as mentioned in [16] and [27], an LP never goes back to *white* and thus effectively stays in *red*. Though this revertion to *white* should ever happen in case the algorithm should run multiple times. If the color is naively reset to *white* after the second round, this could cause problems, as such a message could then make the receiver falsely think that it has received all *white* messages, whereas it has actually received one *white* message from another *white* zone.

6. **Red to White 2**: the receiving of a *red* message will not be special and this case reduces to the situation where a *red* message is sent to a *red* LP, as only the sender takes the new time into account.

1.2.9 Fossil collection

The need for a GVT arises mainly due to memory limitations. A time warp algorithm has no idea which states could still be needed when rolling back, so it should save all of them. As simulation time increases, the amount of memory that is required to save all this data increases correspondingly. Therefore the need for *fossil collection* arises, which will perform this cleanup. Because the GVT provides a lower bound to the simulation time at all different LPs, no rollback to a time before the GVT will ever happen and thus all data from before that point becomes certain and therefore useless to save. The process of cleaning up these *fossils* is called *fossil collection* and different algorithms exist to do this, like *Pal*[46]. Performancewise, [46] states that the fossil collection overhead is between 2-10% of the total simulation time in most applications.
Together with fossil collection, an advancing GVT also means that all pending operations from before the GVT are now safe to be actually performed. This is useful for irreversible operations, like printing tracing information or writing to file.

Algorithm 7 Mattern's algorithm when receiving a control message $< m_{clock}, m_{send}, count >$ in P_{init}

wait until $V[init] + count[init] \leq 0$
if $count = 0$ **then**
  $GVT\_approx \leftarrow min(m_{clock}, m_{send})$
else
  send $< T, min(m_{send}, t_{min}), V + count >$ to $P_{initmodn+1}$
  $V \leftarrow 0$
end if

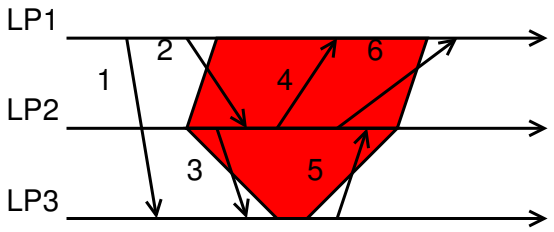

Figure 1.6: An example of all different message possibilities

1.2.10 Optimizations

In optimistic simulation, there is a high probability that some unnecessary computation is done, which will eventually be rolled back. However, there is a possibilty that the straggler message did not actually cause this computation to be incorrect. One of these cases could be a simple query message, which does not alter the state at all. Rolling back all previously done computations and redoing them is then wasteful and could be avoided. This section will discuss two such optimisations that can be used for *lazy rollbacks*. The main disadvantage of both of them is that they certainly incur an additional overhead, while they possibly don't provide any benefit to the simulation.

Lazy cancellation

As soon as a straggler is received, all messages that were sent after the timestamp to which rollback occurs will be unsent. There is a possibility though, that some of these messages are afterwards sent again, for example when the straggler message is a query. If the anti-message sending is postponed until it is certain that the message would not be sent in a sequential execution, it might be possible that some of these messages do not get unsent and that no *secondary rollback* is necessary.
This method has several disadvantages, mainly that there is no guarantee that messages will ever be sent and thus that it cannot be assumed that it will be useful in the general case. Other limitations are that it requires even more memory, as it needs to hold even more anti-messages. Furthermore, all messages that get sent need to be compared to those that should be rolled back. Finally, if such rollback actually has to happen, it comes even later than the actual rollback and the receiving LP will have to perform a much bigger rollback.

Lazy reevaluation

Lazy reevaluation is similar to lazy cancellation, but it compares the states of the models before calling the actual computation. If it detects that the states are equal, it can simply *jump forward* in simulation time. This method again has the same disadvantages of lazy cancellation: it requires more memory for all these extra states and requires a comparison between states at every simulation step. Again, no certainty can be given that this technique will ever have any positive effects.

1.2.11 Memory problems

Even though optimistic simulation seems to be the perfect way to parallellize models without any prior knowledge about the model, and thus offer seemless distribution, several situations can arise where the usage of time warp can result in very slow simulation. In the worst case, it is even possible for time warp to be slower than a normal sequential simulation[3]. Most of these problems are related to the 'work as much as possible' mentality of time warp.
Probably the worst problem that can occur with time warp is the lack of memory. Since time warp needs to save a lot of data during the simulation run, it is intuitively clear that there is a need to clean up such data. While this cleanup is provided by the fossil collection, situations can occur where fossil collection is insufficient to allow simulation to progress.
Assume a model that is distributed in an unbalanced way: a generator at machine 1 will transition very quickly and create lots of messages, which can be done without any rollbacks as there are no inputs to this model. At machine 2, a lot of complex logic is used to process the incomming messages. Further assume that both machines have equal processing power, meaning that machine 1 will advance faster in simulated time than machine 2 due to the lower amount of logic that is required. Since machine 2 advances slower, GVT will be (approximately) equal to the clock at machine 2. Because machine 1 advances at a much faster rate, additional states will be created[4], whereas they will not be cleaned up at the same speed due to the slower processing power of machine 2.
These situations are often unavoidable, as a perfect distribution of models is often impossible: there will always be a different processing rate at different machines. Our only hope is thus that a rollback will occur, allowing the machine that is far ahead to clean up a lot of memory. Sadly, this is not the case in our example. Simulation of this example model will thus make

[3] This problem is true for every distributed simulation algorithm presented here
[4] A possible solution for this specific situation is implemented in PythonPDEVS and mentioned in section 2.4

the simulator crash as soon as machine 1 runs out of memory, which can be fairly fast. To make matters worse, a sequential simulation of the same model would not cause any problems due to the fact that no state saving is needed and all parts of the simulation will progress at the same pace.
Several solutions to this problem are mentioned in [16], though they often make the assumption of shared-memory, which is not the case in this project. Other problems also exist, like *rollback echoes*, incorrect computations that spread, ... These are not mentioned here in detail and we refer to [16].

1.2.12 Conclusion

Time warp offers a general solution to parallellization of models, at the cost of possibly incorrect decisions. These decisions have to be rolled back, which causes additional overhead. The main problem with time warp is that there is no limit on the amount of memory that is necessary for some models, which requires additional algorithms and techniques to take care of this problem. However, with the current size of computer memory this should not be a problem in most situations.
While it seems that time warp is unaffected by the properties of the model that is being simulated, it is highly dependent on the distribution of the model as this is responsible for the rate at which they progress and possibly diverge. Additionally, models at different LPs should pass as few messages as possible due to each message being a potential rollback.

1.3 Conclusion

In general situations, the use of time warp is prefered over conservative as it assumes that no problems arise until they actually occur. This allows for generally efficient simulation, even in models with a small lookahead. Optimistic simulation does have the disadvantage that it requires state saving and thus requires more processing power (to copy the state) and more memory (to hold the copied data), compared to conservative simulation.
For specific models, conservative simulation could be much more efficient, but only in cases where a high enough lookahead is present. There also exist a lot of different conservative algorithms, each of them being specialised for a specific kind of model.
As the goal of our DEVS implementation is to provide a general DEVS simulator, time warp was chosen as our simulation protocol. In the future this also gives more potential for increased performance, as the distribution of the model can be changed at runtime by the simulator itself depening on the current situation.

2 Optimistic simulation in PythonPDEVS

Building upon the different possibilities from the previous chapter, we will now provide the different choices that were made in our PythonPDEVS implementation. For each such decision, the rationale will be discussed and a high-level view about the implementation of these features.
This chapter will not go to deep into the implementation details and design of the simulator, as this is provided in the next chapter.

2.1 Parallel DEVS

Some changes to the simulation kernel were necessary to support Parallel DEVS. Whereas there is no huge difference between the abstract simulators provided for Classic DEVS[47] and Parallel DEVS[8, 9], both have different possibilities for optimisations to their algorithms. Furthermore, we support distributed simulation, so there is a need to provide a step in the simulation algorithm to *inject* the incomming messages into the simulation. For these reasons, our modified simulation algorithm is discussed here as a specific decision.
In contrast to the abstract simulator, we do not use any parallellism at the LP level itself. This was done for several reasons:

1. The overhead becomes too big: when the transition function changes the state, the new state must be shared between the different cores. More importantly, there is also an overhead associated with process creation and destruction, which can be very high in Python. Python also needs to use a seperate process, as threads are unable to use multiple cores under Python.

2. The core that is used for the parallel processing is then only accessible in case at least two transition functions happen at the same time (which is often the case: an internal transition function often causes an external transition function elsewhere), though it cannot be used for anything else, like routing the messages. This only gives a relatively low utilization on the additional cores. They are thus better used as additional LPs, so that they can be used fully.

This opinion is also expressed in [20, 19]. Previous versions of the simulator had an option to allow the user to specify the number of cores that the simulation could use, which was only a few lines of additional code. This code was removed due to the previous reasons and to have some cleaner code at several components.

Rationale

First of all, the most important reason for a custom simulation algorithm is the need to *inject* external messages. Since this is an inter-node message, both simulation clocks will not be synchronised and therefore it is not possible to ommit such a step in the main simulation loop, as they will not be produced by the output generation messages.
A second reason is that the abstract simulator sometimes mentions the use of e.g. a *done* message. While this fits better in a message based simulation, it also creates an inefficiency, as this message will need to be processed seperately. For this specific problem, we took the same approach as in PythonDEVS, where the *done* message will be considered as a return value. This decision causes some specific problems in the case of distributed simulation, requiring a slightly different approach. Also, some parts of the algorithm can be written in a more efficient manner because *direct connection* is used, which is explained in section 2.5.

Implementation

Algorithm 8 The algorithm for an atomic model

```
if msg.type = 0 then
    timeLast ← (t − elapsed, 1)
    timeNext ← (t − elapsed + ta(), 1)
else if msg.type = 1 then
    imm ← abs(t − tn) ≤ EPSILON
    empty ← input = {}
    if notimmandnotempty then
        elapsed ← t − tl
        extTransition(input)
    else if immandempty then
        intTransition()
    else if immandnotempty then
        confTransition(input)
    end if
    ta ← ta()
    if ta = 0 then
        age ← age + 1
    else
        age ← 1
    end if
    tn ← (t − elapsed, 1)
    input ← {}
else if msg.type = 2 then
    output ← {}
    elapsed ← 0
    transitioning.add(self)
    returnoutputFnc()
else if msg.type = 3 then
    merge input
end if
```

Note that our implementation does not take into account any underlying coupled models, as these will be removed due to the direct connection feature presented in section 2.5.

Example

As a simple example, the message flow for a small queueing system is presented in figure 2.1. As can be seen, model *A1* and *A2* are ready to transition at time 1, while model *A3* will wait until time 2. The following steps are required:

1. Check for external incomming messages, in our example none are present
2. Find all imminent components and save their references. The earliest, and thus next, time is 1. All models with a timeNext equal to 1 will thus be notified, this set consists of model *A1* and *A2*. These models will call their *output function*.
3. The output of the *output function* is sent as a message to the connected models, which are models *A2* and *A3*. These models are also inserted into the set of imminent components.
4. After this routing is done, the processing will return to the root node, which is notified that all messages are processed. The only remaining step is to actually process the messages, so each process in the set of imminent components is sent a message.
5. Each model that has received such a message will check whether or not it is an internal, external or confluent transition function that should be called and the corresponding function is executed.
6. As soon as the sending of this message is finished, the simulation step has finished and we are ready to advance the clock.

Algorithm 9 The algorithm for a coupled model

if $msg.type = 0$ **then**
  $tl \leftarrow (0, 1)$
  $tn \leftarrow (\infty, 1)$
  for all $child$ in $children$ **do**
    $child.send(msg)$
    $tn \leftarrow min(child.timeNext, tn)$
  end for
else if $msg.type = 2$ **then**
  $immChildren \leftarrow getImminent()$
  $bag \leftarrow \{\}$
  for all $child$ in $immChildren$ **do**
    $merge(bag, child.send(msg))$
  end for
  return bag
else if $msg.type = 3$ **then**
  $merge(input, msg.content)$
end if

{Note the abscence of $msg.type = 1$ as this will never happen}

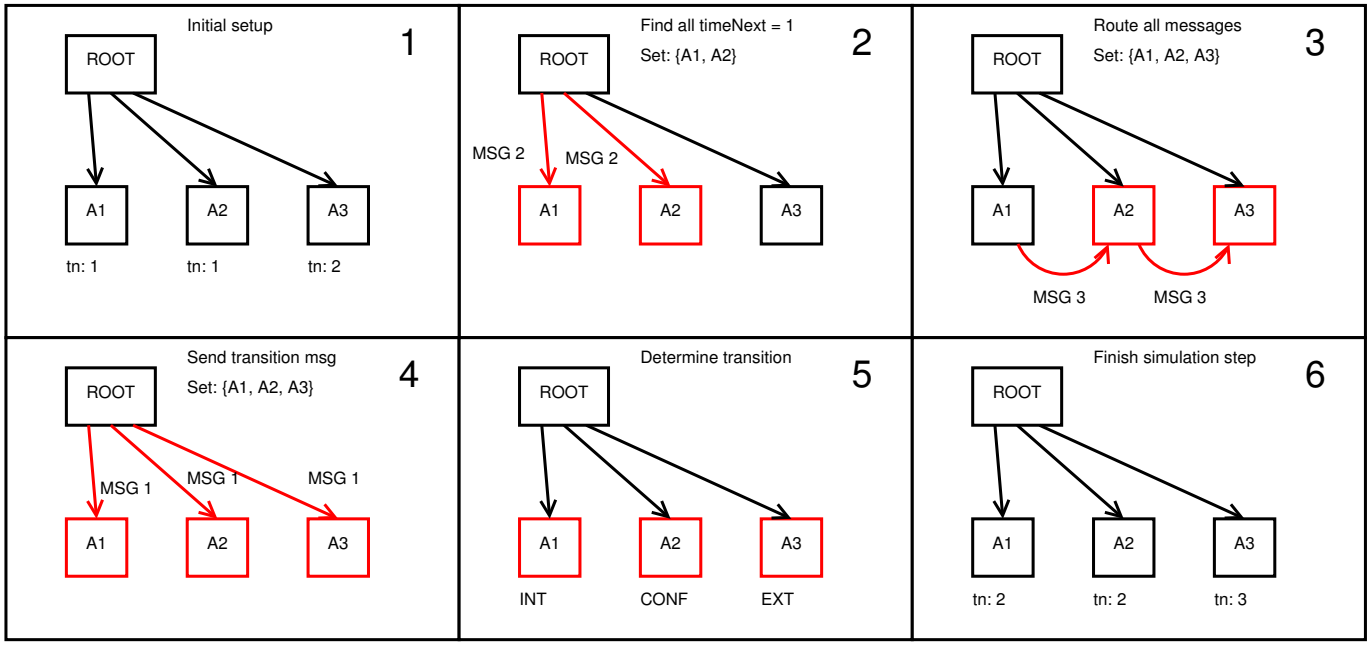

Figure 2.1: The message flow in Parallel DEVS simulation

Problems

An important problem of our ommision of the *done* message is that we would be unable to perform asynchronous simulation, because we would have to wait for the return value instead of processing the message when it arrives. This problem is solved by returning an empty bag immediately if the model is remote, but the remote model will have a connection to the local kernel. This kernel will then be notified of the actual answer and will be responsible for delivering it to the correct model. In order to allow for a clean solution for this problem, we again employ direct connection (see section 2.5) to change the model at run time.

2.2 GVT calculation

As mentioned in section 1.2.8, the GVT calculation will be done to be able to perform fossil collection and perform pending IO operations.

Rationale

The need for a GVT algorithm is fairly logical as otherwise all saved states would just 'pile up' in a simulation kernel. The choice for a specific GVT calculation algorithm on the other hand is not so simple. For PythonPDEVS, the decision was made to use Mattern's algorithm (see 1.2.8). The main advantages of this algorithm is that it allows the GVT calculation to happen asynchronously from the actual simulation and providing very little overhead on the actual simulation. Also, there is no need for acknowledgment messages as in Samadi's algorithm, reducing the number of messages that have to be passed and processed.

Implementation

The implementation of Mattern's algorithm is mostly according to the algorithm mentioned in section 1.2.8, though with a little twist. In [16] and [27], the algorithm only focusses on a *single* pass over the system. This means that at the end it cannot be guaranteed that a certain color is set at a specific kernel, neither can it guarantee that the vectors are cleared at the right time. For this reason, we slightly altered the algorithm to always take two rounds over the ring (to prevent all nodes apart from the controller from 'guessing' what has happened) and clean up all vectors after the GVT was set. This was done in such a way that multiple passes of the algorithm are still correct. One of the downsides is that it takes slightly longer and has a slightly larger overhead. Another downside is that there must be some time between successive runs, as transient messages between the reset of the algorithm are sometimes not logged. To prevent this problem from becomming a fault, we only allow the GVT algorithm to run for a maximum of once a second. If message passing takes longer than one second, there are probably serious problems in the middleware or the network.

2.3 State Saving

Due to the possibility of rollbacks, it is required to perform some kind of saving for the input, output and states. For the input and output messages, it is only required to save those that are external to the local kernel. This is because the others are perfectly reproducable by executing the transition functions. On the other hand, it is not possible to do the same thing with the states of the models. For this reason, the input and output message saving happens at the kernel level (during dispatch and receivement of messages), whereas the state saving is done by the models themselves right after a transition function.

Rationale

Saving objects costs time, so the more we don't have to save, the better. If we were to save each and every message, so also internal messages, it would become very costly because we need to 'intercept' all of them. On the other hand, if we save all local message too, we gain some more insight in message passing and we are able to use infrequent state saving. Currently this is not possible as we might have to roll back the model to a previous point in time, that doesn't 'line up' with the other models. If we were to save messages, we could send the internal messages again and thus circumvent this problem. Internal message saving however, can be as expensive if not more expensive than state saving, due to the frequency of them. Furthermore, if we save states very infrequently (to make up for the message saving), we will have to do long rollbacks, slowing down simulation even more.

Implementation

As soon as a message is send by a *Remote DEVS* model, it is appended to the outputqueue of the local kernel. If the message is received, it is also scheduled by the message scheduler, which will keep track of all incomming messages, even after they are injected in the model. State saving is done by the models themselves and is also done using a list. This list contains both the *timeLast* and *timeNext* values. The *timeLast* is necessary to search for the state to roll back to, as we want the state at the time as soon before the rollback time as possible. The *timeNext* value is necessary in case the incomming message that caused the rollback happens simultaneously with an *internal transition*. By saving this timeNext, we are able to remove the call to the

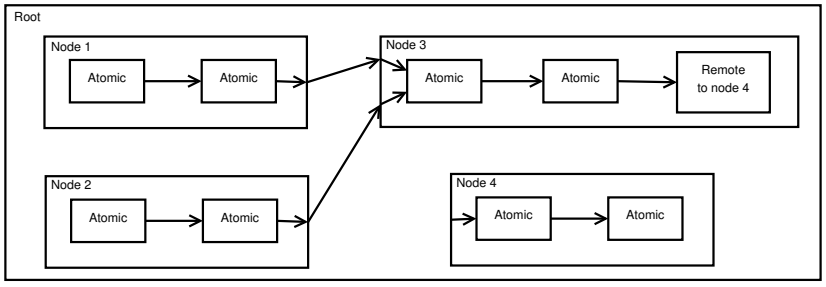

Figure 2.2: Highly optimizable model for reversability analysis

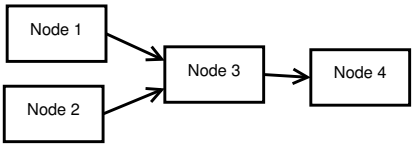

Figure 2.3: Corresponding dependency graph of figure 2.2

timeAdvance function right after a rollback.

2.4 Irreversibility

A simulation kernel (a logical process) can be called irreversible if it is impossible for this kernel to roll back or be influenced by any other logical process. Even during simulation, it is possible that an irreversible model finishes simulation before other models. As that model is irreversible, it is certain that this model will never be activated again, so it can notify all other simulation kernels that it will no longer send out messages. As soon as this happens, this kernel can be removed from the dependency graph, potentially making other kernels irreversible.

Rationale

If a kernel is irreversible, it doesn't need to save its inputs, states or outputs. This saving is one of the causes of time warp using a lot of memory. Furthermore, in Python it is very expensive to copy data. In several profiling runs, it was clear that for enormous states, this state saving can become one of the most time consuming operations. Therefore, we employ any available method to prevent us from saving the state.
Since this means that irreversible models will run faster than others (due to less work being done), they can notify other kernels before these are finished, thus speeding up the simulation of these other kernels too. However, it is not necessarily a good idea to 'overmodularize', since these finished models will still keep the processor idling because model migration is not supported.
Even though irreversibility is not a common case in more real-life models, it provides no run-time overhead. The only cost is in a slightly slower startup because all information about the complete model needs to be exchanged.

Implementation

In order to implement this concept, it was necessary to construct a dependency graph. At first sight this seems simple, as we can simply use the connections that were created by the models. However, this is not that straightforward, as *every* coupled model can be a remote model. So it is possible that a coupled model without inputs is not irreversible, because it has a remote submodel. Due to direct connection, this problem has somewhat decreased and we are able to construct such a dependency graph relatively efficient. Every model will now have a set of all its dependencies and as soon as it is notified that a kernel is finished, this set will be decreased. If the model has no more dependencies, it becomes irreversible too.

Example

An example of a model that can be optimized is given in figure 2.2. At the start, it is clear that neither *node 1* nor *node 2* can ever receive a straggler. This is shown explicitly in the dependency graph shown in figure 2.3, where both *node 1* and *node 2* are shown as having no incoming edges. As soon as *node 1* has reached its end time, it can signal this to all nodes in the simulation, so they know that *node 1* will never generate new messages (which could potentially be stragglers). If also *node 2* signals that it has finished the simulation, *node 3* will have no more dependencies and can be flagged as irreversible too (because the output of *node 4* is never used).
On the other hand, the model in figure 2.4 cannot be optimized in this way. Since *node 1* can always generate output that must be handled by the root model, the root model can only become irreversible if *node 1* is irreversible. The other way around, *node 1* depends on input from the root model, this makes it impossible for *node 1* to become irreversible without the root model being irreversible. This is shown explicitly in figure 2.5, where the cycle indicates a dependency which can never be broken, thus indicating that all models inside the cycle will never be irreversible.
Such situations do not necessarily signal a problem, as simulation will just progress at the normal pace, though an irreversible model can take a lot of shortcuts, thus increasing performance in many situations.

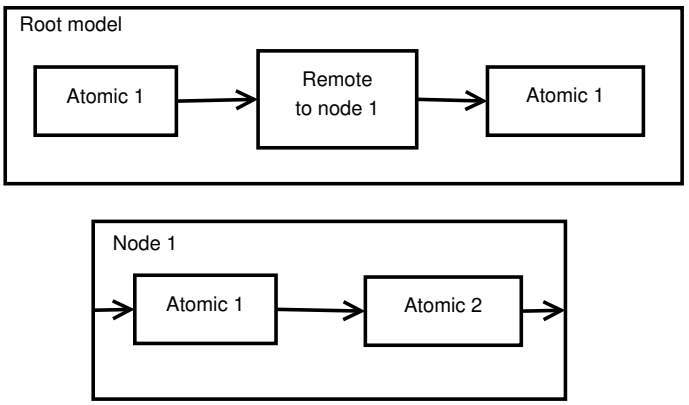

Figure 2.4: Inoptimizable model for reversability analysis

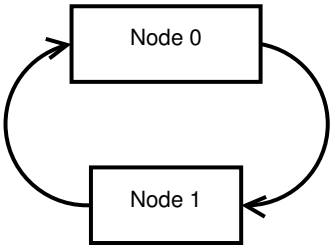

Figure 2.5: Corresponding dependency graph of figure 2.4

Trivial case

The trivial case with such analysis, is a model that is completely local. The dependency graph thus exists of exactly one node, with no edges. Clearly this node is thus irreversible and can use all optimisations as normal. Actually, this trivial case is a special case, more specifically, it is a non-distributed model. Because of *time warp* always requiring extra computation, sequential execution would also be slowed down due to all the extra work that would have to be done (state saving, message saving, ...). Since PythonPDEVS is not created for only distributed and parallel simulation, it is important to keep in mind the sequential simulation runs.
Should irreversibility not be used, we would have to check whether or not the number of kernels is exactly one for all time warp specific parts. Our irreversibility analysis is thus actually a *more general* form of such checks, because a sequential simulation will also be irreversible. The other way around, an irreversible node will also appear in distributed situations. Therefore, we actually get efficient sequential simulation for free, thanks to this concept of irreversibility.

Critique

Irreversible models offer a lot more performance by taking several shortcuts. The sole critique is that irreversibility is not frequently present in realistic models. Most of these models will require some kind of feedback loop, causing irreversibility to be a useless feature. On the other hand, sequential simulation is one of the situations where irreversibility is guaranteed. As one of the primary requirements of PythonPDEVS should be that it is still an efficient sequential simulator, because it is the successor of PythonDEVS, irreversibility will thus be a necessity for high performance simulation. The only addition is that a dependency graph must be constructed and maintained, but since this happens exactly once at the start of the simulation, this will not incur a run-time overhead and still allow these few select models to be simulated a lot faster.

2.5 Direct connection

Direct connection will reduce all atomic model connections to a one-hop connection. It furthermore reduces the need for coupled models and makes a flat hierarchy (though it should not be confused with flattening[4]) of an arbitrary DEVS model. In the distributed setting, it will also create such one-hop connections on an inter-node scale. These direct connections will allow a coupled model that is just an 'intermediate model' to be completely ommitted. In the case that this model resides on a different simulation kernel, direct connection will thus completely avoid that kernel and the associated message passing overhead and even prevent it from rolling back.

Rationale

In very deep models, direct connection is known to speed up the simulation due to very fast message passing. Such speedups were visible in [45] for a variety of models. In a distributed simulation, this connection will also happen and has the potential to speed up the simulation due to the prevention of several rollbacks. Another addition of direct connection is that is prevents the 'race' between an anti-message and the actual message. Otherwise, each intermediate coupled model would have to check for a compatible anti-message to cancel it and vice versa.

Implementation

The way that direct connection is handled in a distributed setting is very different from how it is done in a local setting. Only the bare basics of local direct connection remain the same, whereas all passing of messages over the local kernel boundaries are

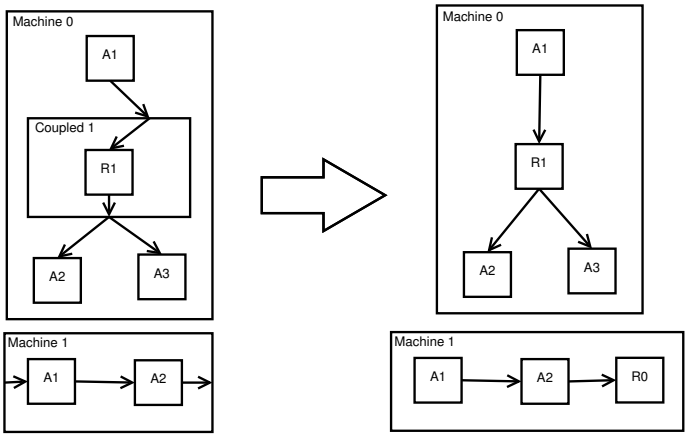

Figure 2.6: The effect of direct connection on a simple model

to be handled by communication between two kernels. Due to the way this is handled in the implementation, it also connects beyond kernel boundaries and thus models never 'create output' anymore. The remote model will then be abstracted to a *proxy model*. For the situation where a single output connects to multiple inputs, all of them on another kernel, the message will only be passed once and split up at the receiving kernel.
As was mentioned in section 2.1, direct connection will also aid in asynchronous simulation. A coupled model that has output ports to other kernels, will have its output ports removed and replaced by a remote model. These remote models then send the message to the destination kernel, which will process them as an ordinary external input message.

Example

An example of a basic model that will get somewhat easier using direct connection is shown in figure 2.6. Note the creation of the remote models, which are considered as atomic models. For some more examples we refer to [45, 4], which is still the basis of the direct connection algorithm.

2.6 Anti-message processing

Anti-messages are necessary to 'unsend' messages after a rollback took place. To be able to send anti-messages, it is required to save every outgoing message, as to have all necessary information about which message was sent when and to where. Anti-messages require only two parts of the original message, specifically the ports on which the anti-message was sent, and the ID of the message.

Rationale

Anti-messages should be sent out and cancelled as soon as possible, as otherwise the message that is being cancelled might already start being processed and the computation will have to be rolled back. Checks for whether or not a message was cancelled need to be very efficient as these checks need to happen at every step in the simulation.

Implementation

Filtering out anti-messages happens in the messageScheduler class. It will flag the ID of the message as 'cancelled' and as soon as a message is received or sent, it will be checked with these cancelled messages and if it was cancelled, it will be skipped. Such an implementation is possible due to the fact that direct connection is used, so the anti-message will not have to 'catch up' to the actual message.
The ID of these anti-messages was originally computed using the Python UUID module, though this had a very low performance compared to manual ID generation. This manual ID will be based on the name of the sending message and the counter of sent messages.
Anti-messages are exactly equal in structure to the normal messages, though their content might be empty as long as the ports are still present. The only difference between these messages is that a flag is toggled.
The structure of our message scheduler and corresponding complexities is discussed in section 3.3.

2.7 Checkpointing

Checkpointing provides the user with an automated means to restore a simulation run as soon as a problem occurs at a certain node. Such checkpointing should not significantly slow down the main simulation, as it is only required if simulation crashes.

Rationale

If a simulation runs for a long period, it is possible that failure occurs. In distributed simulation this problem becomes even bigger as failure can occur due to other kernels. Naturally, if ten nodes are used instead of a single node, the chances of a failure increased tenfold.

Implementation

For checkpointing in PythonPDEVS, the Python pickle module is used to provide easy serialisation of the kernel and all associated models. Every time the GVT is recalculated, the possibility to take a snapshot is present. It is required to 'lift' on the GVT algorithm as otherwise we would require multiple ring algorithms and it would be possible that all kernels are in a different state with correspondance to the GVT. Having different GVT's at simulation kernels would be problematic for obvious reasons.
To restore a simulation, we simply need to unpickle the provided pickles and reinitialize them. This reinitialisation also resets some variables of the kernel that could not be pickled, like the tracers and all locks that were present. After a succesful restoration, it is required that all models be rolled back to the GVT itself because all transient messages will also be lost as they couldn't be saved.

2.8 Nested simulation

Nested simulation allows the user to simulate a model inside the simulation of another model. This can happen an arbitrary number of times. It is also possible that multiple models try to start a nested simulation and so on. This requires fairly complex logic to make this work in a distributed system. To make matters worse, this nesting starts in user code, where the simulator has no control and has locked the models main lock.

Rationale

Nesting could be useful when the behaviour of a model depends on the result of other models. Such functionality was possible in the previous versions of PythonDEVS, as they were local and didn't use any global variables, therefore this functionality was also added in PythonPDEVS, albeit in a slightly different way.

Implementation

Nested simulation is fairly complex in a distributed setting. This is mainly due to the fact that there can be a race condition between multiple models all trying to start a nested simulation. Such nested simulation requires the kernel to be aware of this, as it must route incomming messages to the correct simulation run. Therefore, a kernel has a list of simulations, called a *simstack*. The user is required to annouce the start of a nested simulation to the controller of the simulation to prevent deadlock situations. Such announcements are required as otherwise it would always be necessary to halt other processes during the processing of a single event. More details on the implementation of nested simulation and the usage can be found in section 3.9.

Example

A simple example with 3 nodes is presented in figure 2.7. At the beginning, only a single simulation is running and communication between all coupled models is possible. At a certain time, *Coupled Model 3* starts another nested distributed simulation on the same nodes as can be seen in figure 2.8. In the nested simulation, models from the nested simulation have absolutely no access to the lower levels. Only the model that started the nested simulation, *Coupled Model 3* in this case, will have the ability to communicate with the nested simulation run. All other models will simply be halted and the core will simulate the higher levels first. After the nested simulation finished, that 'ring' will dissapear and the other simulation will continue.

2.9 Termination condition

PythonPDEVS offers the user two possibilities to signal the end of a simulation run. Either by specifying a termination *time* or a termination *condition*. This termination condition has access to the local model (not to the remote objects) and the current simulation time. Because every simulation kernel will have different models, they can't have exactly the same termination condition, so for simplicity we made the assumption that the termination condition will only run on a single node. This node can be chosen freely by the user.
Due to the use of time warp, all kernels will be at a different point in time, making it extremely inefficient to allow a global termination condition. Should a global termination condition be used, at each step all the states would have to be synchronized,

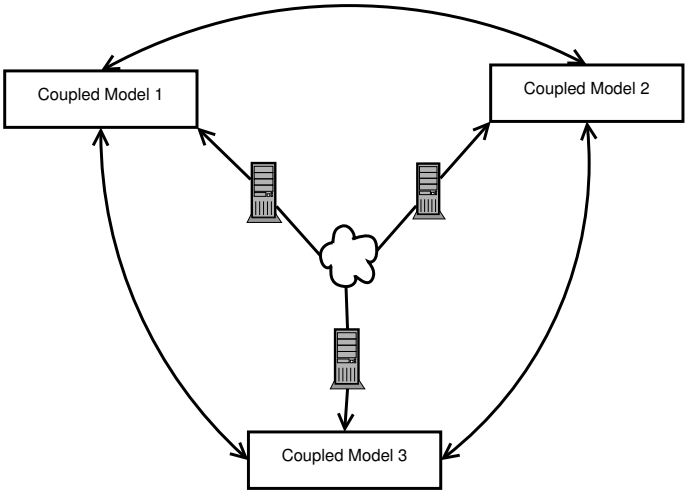

Figure 2.7: The situation in a non-nested simulation

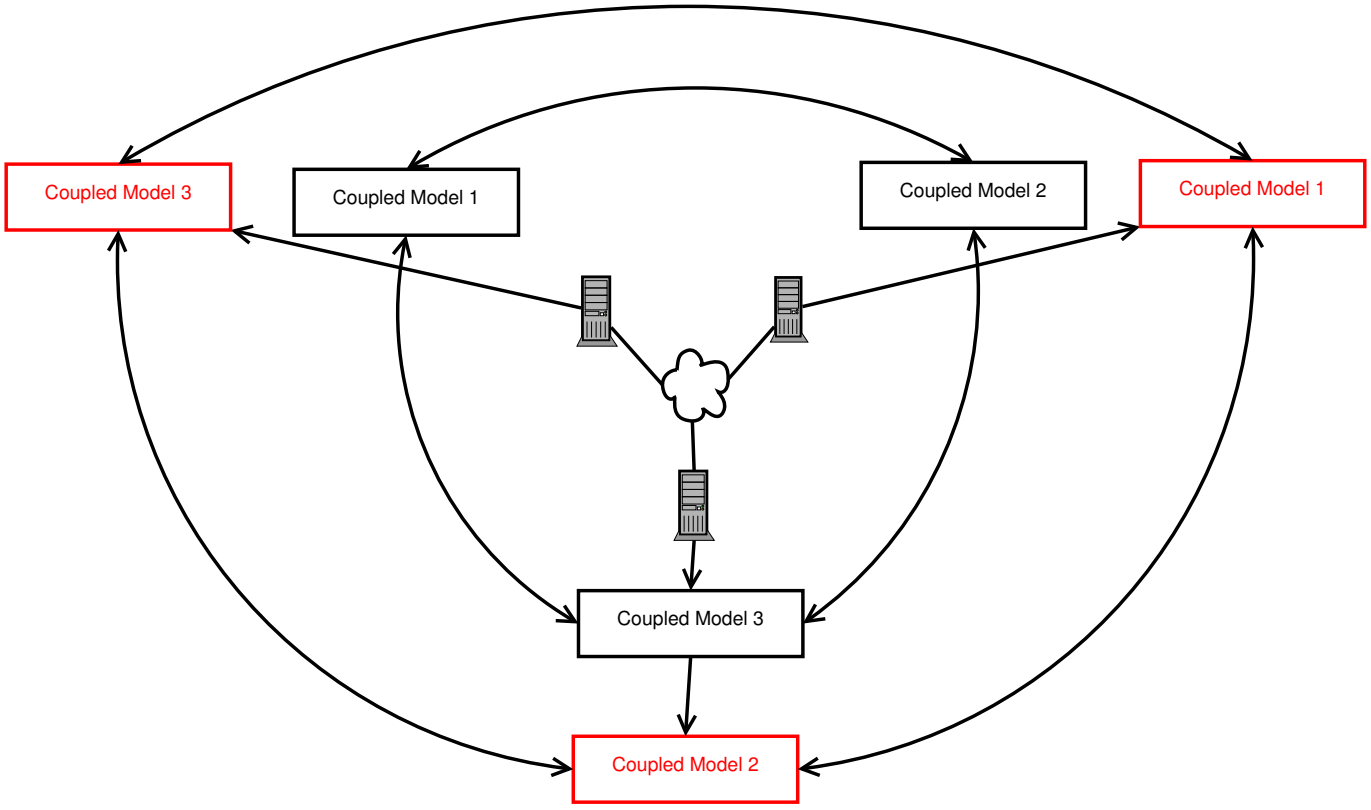

Figure 2.8: The same situation after starting a nested simulation in *Coupled Model 3*. The higher nesting level is shown in red.

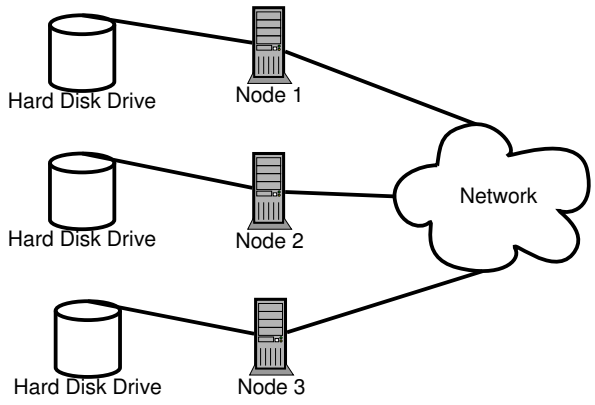

Figure 2.9: Logging without a centralized logging server

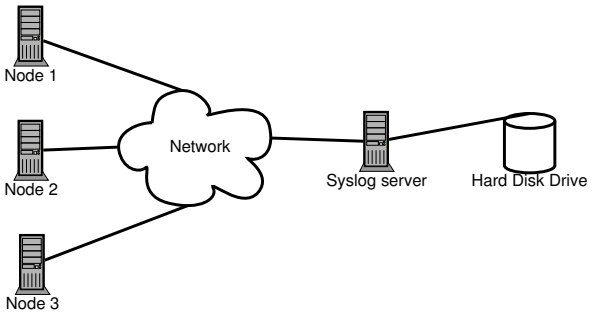

Figure 2.10: Logging with a centralized syslog server

causing both an extremely high bandwidth usage and extremely long latency. Of course, the user is always free to group all models that are required for such a termination condition on the same node.

Rationale

By allowing the user to provide a termination *condition* instead of just a termination *time*, the user gets a lot of extra control over the simulation. However, this control comes at a cost, since the function will have to be evaluated each and every time something changes in a model. This termination condition makes it possible to stop simulation as soon as an *unacceptable* state is entered, even though the termination time is not yet reached. In this aspect, the extra cost of such evaluations can be compensated by several simulation runs that can be skipped after just a few seconds of simulation.
These checks incure some overhead, as it is an additional function call within the main simulation loop. A small performance gain could be achieved by only allowing termination times in situations where this is the only requirement.

Implementation

The node that is responsible for the execution of the termination condition will always execute this after each simulation step, we will call this node the *termination node*. All other kernels will simply simulate until infinity, as they have no idea on how long they will have to simulate. As soon as the termination node detects that it must stop the simulation, it will overwrite its checking function by one that is always true. This is necessary to allow the GVT to progress. Now there are two possibilities: either the GVT progresses over the simulation time at which the termination node decided to terminate, or the termination node is being rolled back. In the first case, the termination node signals all other nodes to stop simulation as soon as possible. It is possible that the GVT has already passed over this time, since all logical processes kept running. For this reason, a final rollback takes place on the action list, to remove all such actions from the logging. In the second case, the action that the termination node had planned will be rolled back too and simulation will progress as usual.

2.10 Logging

Logging is an essential part of distributed software. To organize logging, all logs are sent to a central 'logserver', which just has to run a Syslog server. This server will then be responsible to order the logs, show the timestamp, show the contained message and so on.

Rationale

One of the biggest problems in distributed software is debugging the software itself. Most observable faults are caused by slight timing differences and are often non-repeatable. To allow such bugs to be traced to the lines of code that are responsible for the fault, it is very useful to have some kind of logging cabability. A mere print statement is often insufficient, as it doesn't allow the merging of multiple logs. Also, a print might get interrupted by another print, resulting in an unreadable mess. Saving to a file is problematic too, as the information would be scattered over the network. For this reason, a Syslog server seems the best option.

Implementation

Logging is implemented using the standard Python logging module. This module provides all necessary interfaces for a syslog server, both locally and over a network. Since logging is a relatively expensive operation, it should be removed in production code. Due to the interpreted nature of Python, this is mostly impossible to do in a clean and concise way. To resolve this issue, all logging statements are put in *assert* statements. While this is not the cleanest solution, it allows the Python bytecompiler to

remove these statements on demand. If few logging was used, this would not really cause a problem, but in PythonPDEVS most operations are logged, causing lots of function calls, which are already extremely slow in Python.

2.11 Tracing and Irreversible actions

Tracing was a feature that was present in previous versions of PythonPDEVS, though the need arose to keep them working in a distributed environment. Tracing uses IO, which is problematic in time warp as we might have to *undo* our previous work. To circument this problem, the controller will be responsible for executing all necessary functions, but only after the GVT has passed over the simulation time at which this action was requested. This way, it is possible for models to perform a rollback and notify the controller that all of its irreversible operations should no longer be executed. Since rollbacks will only happen to times after the GVT, these actions have not been executed yet, so a rollback is as simple as removing this action from the list. PythonPDEVS still supports the verbose, XML and VCD tracers that were available in PythonDEVS, though slight changes were needed to support the new *confluent transition* function.

Rationale

Many models require irreversible actions, like writing to a disk, showing output to the user, ... In time warp it is possible that this action should not actually have been executed due to the possible violation of causality. Tracing is one of these important actions and is probably one of the main reasons why a simulation is performed. What would otherwise be the reason to run a simulation, if the results are thrown away immediately because no input or output is generated?

Implementation

To keep maximal flexibility, the models have the possibility to execute a *delayedAction* method. This method will queue an action that will be executed with the Python *exec* statement. For tracing, several functions were written that are thus queued to be invoked at certain times. As soon as a rollback happens, this queue is split up in 'stale' actions and messages that should still be executed. As soon as the GVT is progressed at a node, all queued events up until the GVT will be executed.

2.12 Custom copy

Each message that is being produced by the *outputFnc* can have an optional *copy* method defined. This function is then responsible to provide a complete copy of the message itself. It doesn't necessarily need to make a complete deep copy, as long as it doesn't violate the DEVS formalism in any way. This means that if a message contains attributes of which the modeller can be sure that they will not be used for inter-model communication, it is possible to completely avoid this copy and keep using the same objects in memory.
Note that even when custom copy is enabled, the messages should still be picklable by *cPickle* as this will be used when passing the message over the network.
Another possibility is to have a custom copy function defined on the states of the models, which can then be used for copying. This is done by defining an *copy* method in the state itself. The same restrictions apply as with message copies, thus it should also still be a *picklable* state due to checkpointing.

Rationale

The default behaviour for message copy is to use the *cPickle* module that is present in Python. The problem is that this module will use a string as intermediate form, introducing a lot of extra overhead. The user might also have some specific knowledge about the message and which attributes of the message could just be skipped. From small benchmarks it became clear that message copy time is reduced by a factor 16 in most cases (of course depending on the data type that has to be copied).
Again, the same rationale holds for custom copies of states. The problem here being that multiple copies should be made of each state (make a copy to put in the state history and make a copy when fetching it from the state history), whereas pickling automatically produced an intermediate representation. Therefore it is unlikely to achieve huge speedups by defining such a custom copy function, though a certain speedup is noticable. Again, the modeller might have some additional knowledge about the state, to make copies run much faster.

Implementation

As soon as a message is copied, the simulator will check whether or not custom copy is enabled. If it is disabled, *cPickle* will be used to pickle the complete *bag* at once. Should it be enabled, the bag will be iterated and for every element this custom copy function will be called. Due to this all-or-nothing strategy, custom copy should only be enabled if all messages will have such a function.
Another possibility would have been to perform checking for every message in the bag and try calling such a function and use *cPickle* if it is not available. The main disadvantage of this approach is that it incurs a huge overhead by setting up the exception

handling and working in an elementwise manner.
The state saving will be done in a similar way, though it should be selected as the prefered way of copying the state.

2.13 State fetching

After a simulation has finished, the state of models will have changed. In local simulation it was possible to access them like any other variable, whereas this is not possible in distributed simulation. To alleviate this problem, state fetching can be called for by the user. This must be called for by the user explicitly, to prevent lots of unnecessary work.

Rationale

Just like with tracing, the user probably wants some information out of the simulation run. While tracers are very easy for debugging or analysis, some information is better obtained automatically from the states themselves. This functionality was automatically present in local simulation, since the user had the complete model available, whereas this is no longer the case in distributed simulation. Therefore it is very likely that such experiments exist and they would no longer function due to the change to a distributed simulator.

Implementation

The user will be able to *register* a state with the simulator, which causes the simulator to fetch this state from the remote model that actually holds the state. Since the state will be transfered over the network, it is required that it is picklable. After the simulation has finished, the state will be available to the user as a local variable, providing the user with the same flexibility as before.

2.14 Real Time

Real time functionality was added in the new version of PythonPDEVS, though it is only functional in local models. This functionality has nothing to do with the distributed aspect of the simulation and is therefore not really important to be mentioned in this report. On the other hand, it is a new feature for PythonPDEVS and therefore it is mentioned.
The real time component allows for several choices to be made, as the back-end was made as modular as possible. The currently implemented threading back-ends are the default *Python threading* library, but also the *Tkinter* event scheduler. Actual input processing can happen using both an input file and the command line (simultaneously).
The actual real time back-end is based on that of RT-PythonDEVS[44], which was created as part of the course *Modelling of Software Intesive Systems*. This back-end was updated, several bugs were fixed and finally integrated within PythonPDEVS. For more information about the actual implementation of this feature, we refer to the documentation of RT-PythonDEVS.

2.15 Feature interaction

As can be seen in previous sections, a lot of very different features are present. Several of these features influence each other, causing several necessary adaptations (or allows us to ommit several other necessary changes). The most important feature interactions will be discussed briefly.

2.15.1 Checkpointing and Irreversibility

The main advantage of irreversibility comes from the fact that no state saving or message saving is necessary. Our approach to checkpointing however, does rely on this, as it will create an artificial rollback to the GVT at the time of taking the checkpoint. Should checkpointing be done on such irreversible models, rollbacks would thus be required though it is impossible as the state is not saved.
To remedy this problem, our current solution is to disable irreversibility optimisations in cases where checkpointing is desired. Other solutions would require rewriting of the whole checkpointing algorithm, as an irreversible model would have to be reset completely different.

2.15.2 Anti-messages and Direct connection

One of the main problems of anti-messages is that there could occur some race conditions, where the anti-message must try to catch up to the actual message it is invalidating. In our current simulator design, this would require quite some work due to the incomming message scheduling and invalidation.
Since we use direct connection, messages and anti-messages will reach their destination in a single step, causing no problems with such race conditions. This is also the reason why direct connection is not optional in the current version, whereas it was optional in the PythonDEVS.

2.15.3 Real Time and other features

Using the real time back-end means that several features need to be disabled. For example, real time does not work with anything that has to do with distributed simulation and therefore it is only possible to perform a real time simulation on models that are completely local. Some other features are also disabled, as they have no use during real time simulation. Checkpointing and GVT calculation are some of the features that need to be disabled in order to allow real time simulation.

2.16 Conclusion

This chapter summarized the different features and decisions that were made in our implementation of PythonPDEVS. Often several of these features offer much easier and efficient simulation than if they were to be applied seperately. Whereas other features are actually necessary due to the optimistic nature of our simulator. However, these features still have several choices that must be made to give them an efficient implementation.

3 Implementation of PythonPDEVS

In this chapter, the actual design and implementation of PythonPDEVS will be explained, followed by benchmarks of this implementation on a variety of models. At the end, a reference about the new API is provided and how these can be used.
The implementation is based on the PythonDEVS[45, 3, 32] simulator written for Research Internship I, though many changes were made to nearly every aspect of the simulator.
All performance related plots, unless stated otherwise, were made on an Intel i5-2500 quad-core 3.3GHz with 4GB RAM memory, using *CPython 2.7.3* (*PyPy 2.0 beta 1* when PyPy is mentioned) and *MPICH 3.0.4*.
PythonPDEVS is focussed for running on clusters, so we cannot make assumptions about e.g. shared memory[40] or non-modularity[39]. Furthermore, our implementation is done in Python and focusses on a *standard, general* implementation, preventing us from using several advanced methods like template metaprogramming[42] or GPU programming[37].

3.1 Formalism

The new PythonPDEVS simulator does no longer support the Classic DEVS[47] formalism, as the Parallel DEVS[9] formalism was added. The Parallel DEVS formalism has many advantages over the Classic DEVS formalism, certainly when performing distributed simulation. The most important difference in terms of our simulator is the removal of the *select* function. In case a collission would occur between models in different Logical Processes (LP), the select function would have to provide inter-node communication. Furthermore, such collission resolution takes lots of time and has the potential to alter the complexity of the complete simulation run, as was seen in the comparison between ADEVS and PythonDEVS[45].
Another improvement in the Parallel DEVS formalism is the *confluent transition* function, which is triggered in case an *external transition* and *internal transition* function happen at the exact same time. In Classic DEVS, the *internal transition* function would be dropped and only the *external transition* function would be processed.
Furthermore, it is possible for multiple messages to be processed simultaneously and have them grouped in a *bag*. For a detailed comparison between both formalisms (and some others), we refer to [45].
All of these changes provide opportunities for optimisations, so the simulator will be faster for most models with only a change in formalism. However, as the formalism is changed, several changes must be made to the API of PythonPDEVS. This means that Classic DEVS models will no longer function without several (mostly minor) changes. Since these changes were inevitable, the opportunity was taken to also change other aspects of the API to allow for better performance.
Even though [43] talks about Parallel Cell-DEVS, some ideas have also been applied in PythonPDEVS, like only sending a message over the network once and splitting it up at the receiver.

3.2 Design

While the design of the new simulator was based on the previous version, many changes were needed due to the distributed nature of the new simulator. Some changes were also introduced because PythonPDEVS was also made compatible with Python3 and Cython. For example, in Python3 every exception must be a subclass of the *Exception* class. In Cython, multiple inheritance as found in the previous design, is not allowed.
A complete class diagram is shown in figure 3.1.

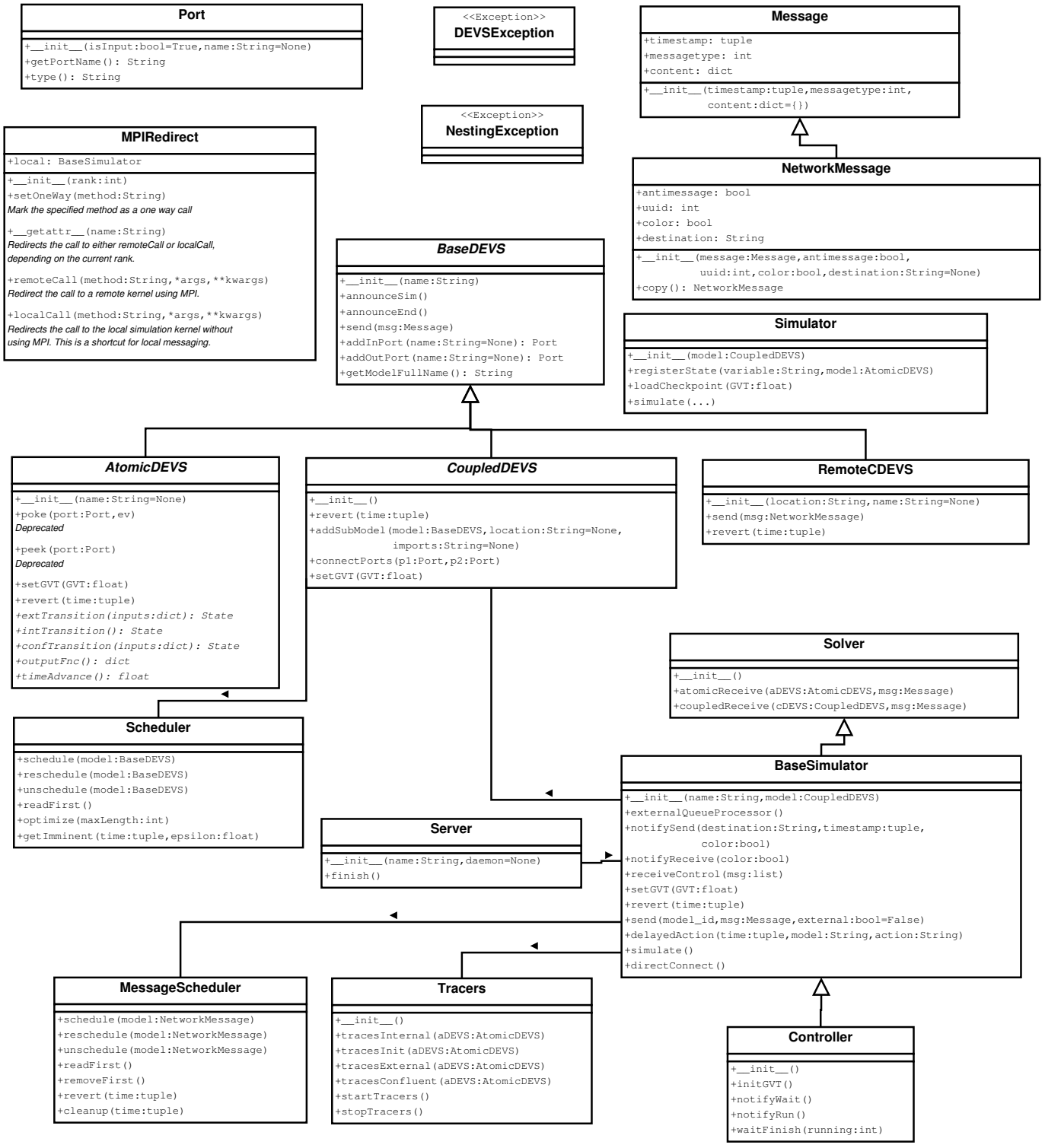

Figure 3.1: Class diagram, only the most important attributes and methods are shown for readability

3.3 Implementation details

As can be seen from the class diagram in figure 3.1, lots of classes are involved in the simulation. This section will provide some details about the responsibilities for each class and, when applicable, the algorithms and data structures will be discussed.

Port

The *Port* class is still the same as in previous versions and simply provides a means of connecting models to each other.

BaseDEVS

The *BaseDEVS* class is the superclass of each and every model that will be simulated. It offers basic functionality that is common to all models, like connecting ports, setting up hierarchies and so on. This class is itself abstract and instantiations are not allowed.

AtomicDEVS

The *AtomicDEVS* class is an abstract specialisation of the *BaseDEVS* class. It provides the functions that are only necessary for atomic models, like a default implementation of the transition functions. All atomic models must inherit from this class as they will otherwise not be recognized as atomic.

CoupledDEVS

This class is very similar to the *AtomicDEVS* class in its goals, as it provides an abstract specialisation of the *BaseDEVS* class, but specific for coupled models. It provides several important functions, like adding submodels. Again, every user-defined coupled model should inherit from this class.

Message

The *Message* class is comparable to a normal *struct* in C, as it only contains data and doesn't have any methods except for the constructor. In previous versions of the PythonDEVS, this class was not present and instead a normal list was used. While such a normal list is perfectly fine for performance, it can quickly become unmaintainable if the simulator grows. This is because a list is actually meant for *homogenous* elements and not as a drop-in replacement for a *struct*. As the content of messages grew immensly due to the distributed functionality, all different elements of the message became more and more difficult to access in a maintainable way.
Furthermore, PyPy simulation can become somewhat faster, as it allows the JIT compiler to have more insight in typing information. This class is also flagged as a class that may be compiled with Cython to an extension type class.

NetworkMessage

A *NetworkMessage* is a specialized form of a common *Message*, with the difference that it contains additional information that is necessary to make passing it over the network possible. This extension was the main reason to transition from a basic list to a class, since it contains several entries. Accessing these elements by index quickly becomes inmaintainable and problems would arise if another entry were to be added.
NetworkMessages are a special subclass, because otherwise all the additional network entries would have to be filled in at all times, even for completely local messages.
Note that this class does support a copy function, as it will be necessary to make copies of these entries to store in the output queue of the simulation kernel. This class is flagged for compilation with Cython too.

RemoteCDEVS

The *RemoteCDEVS* class is a newly introduced class and functions as a proxy between different kernels. It is a representation of a coupled model at a remote location and supports the most important features that are also required for other models. It also provides a kind of *barrier* for the simulation kernel, as all requests that branch to all children will stop at such a proxy.
This proxy provides a *send* method, which will encapsulate the incomming *Message* object in a newly created *NetworkMessage* object. Transfer to the other kernel will then take place, where it can be unwrapped to a normal *Message* object. A *RemoteCDEVS* object is often treated special, since it should not be considered in e.g. determining the *timeNext*, since transitions at other kernels are not the kernels responsibility. That way, we allow for asynchronous simulation.

Simulator

This class has changed very little since PythonDEVS and only has some extra configuration parameters and extra methods to allow for distributed simulation. For more information about the changes, we refer to section 3.9.

Solver

Due to the switch to Parallel DEVS, the PythonDEVS *Solver*s became partially obsolete. Parts of the algorithm were rewritten or slightly altered to comply to the new formalism. One of the most notable changes is that the *AtomicSolver* and *CoupledSolver* are no longer seperate classes as they were in previous versions. This design change was done to prevent the multiple inheritance that was required to make this work. Furthermore, this lowered the overhead as only a single solver is required and no longer two different solvers. The multiple inheritance seemed wasteful too, since there was no actual use of it in the code and there were static references to the actual superclass whose methods were to be invoked.

BaseSimulator

The *BaseSimulator* class is the core of the distributed simulation and actually functions as the kernel. It is a subclass of the *Solver* and provides the means to actually call these functions. The choice for subclassing was mainly due to historical reaons. This class will contain all necessary functions to perform a complete simulation run at this kernel and will be able to receive messages from other (remote) simulation kernels.
Incoming messages are first added to a temporary input queue. That queue will simply act as a buffer for the actual kernel, as messages in it will not yet be considered as *received*. A different thread will eventually be notified of this new message and will add it to the actual *MessageScheduler*.
This somewhat strange design is due to the fact that the *MessageScheduler* and notifications of receiving a message, require several locks. It is possible that these locks are already taken and that the actual message delivery must wait for these locks. Should the locks be held for a relatively long period, the threadpool would run out of threads very quickly and all simulation would stall. Therefore, the temporary buffer was conceived to hold the messages until these locks are free and so remote calls can return a lot faster, reducing on waiting time in the sender and reducing the number of active (but waiting) threads on the receiver. To guarantee fast message processing, no simulation step will be taken while there are still messages in the temporary buffer, as these messages might contain an anti-message for the message that is ready to be processed.

Controller

A *Controller* object is a specific subclass of the *BaseSimulator* and supports some extra methods that are only required at the controller. These include waiting to initialize the GVT and determining whether or not all kernels have finished simulation. It will be the main contact point for all kernels and is therefore required to run with rank 0. Since this kernel will also have other concerns, it is adviced to keep the number of models to simulate at this kernel low to provide better load balancing.

Server

A *BaseSimulator* kernel will be present at every server and the server will be responsible for the actual receiving of the message and directing it to the correct kernel. To offer nested simulation, a *Server* object can have multiple simulation kernels, of which only one is active. The actual simulation is of no concern to the *Server* itself, unless inter-simulation coordination is necessary as is the case in nested simulation or when finishing a certain simulation run.
The *Server* will process all incoming MPI messages to provide RMI-like functionality. The actual processing of such messages is done using threads which are taken from a thread pool.

Scheduler

The *Scheduler* class is a new class, though the logic is actually the same as it was in PythonDEVS. However, to increase maintainability and offer a modular event scheduler, it was seperated in a specific *Scheduler* class. It provides methods for scheduling, unscheduling and rescheduling events and returning a list of all events that must be processed at this instant. It does so using an optimized heap implementation as mentioned in [45].
Note that this scheduler should not be confused with the *MessageScheduler*.

MessageScheduler

The *MessageScheduler* class is a special kind of scheduler, which has no relation to the *Scheduler*. Whereas the *Scheduler* provides the scheduling of transitions, the *MessageScheduler* is concerned with processing all externally received messages. This must be done in a different fashion than the event scheduler, as we are required to keep all events even after processing, to accomodate possible rollbacks. The core however, is mainly the same as a heap is also used. After a message is returned and the simulator acknowledges that it will process it, it will be popped from the heap, but will be added to a list of processed events. If a rollback happens, this list of processed events is inspected and all messages that are rolled back will again be pushed on the heap. As soon as the GVT is set, the processed messages will be checked and all messages with a timestamp before the GVT can be removed.
Due to the seperation of *processed* and *to process* (and different data structures for each), the heap will never be checked in case of a rollback and no 'heapify' is ever necessary. Furthermore, the processed list is guaranteed to be sorted due to the sequential

	Average case	Best case	Worst case
schedule	$O(log(n))$	$O(log(n))$	$O(k \cdot log(n))$
unschedule	$O(1)$	$O(1)$	$O(k)$
get first	$O(1)$	$O(1)$	$O(k \cdot n \cdot log(n))$
remove	$O(log(n))$	$O(log(n))$	$O(log(n))$
rollback	$O(m \cdot log(n))$	$O(m)$	$O(m \cdot k \cdot log(n))$

Table 3.1: Complexity analysis of the message scheduler used for incomming network messages. n is the number of scheduled messages, m is the number of processed elements and k is the number of invalidated messages.

insertion of events. This allows for efficient message finding, rollbacks and setting of the GVT.
The achieved complexities can be found in table 3.1, which are based of [11]. Note that the worst case is due to the use of Python *dictionaries* throughout the scheduler. The worst case should be a rare occurence.

MPIRedirect

A *MPIRedirect* object is responsible for routing an incoming call to the correct kernel. It provides a shortcut for local calls to prevent the use of MPI altogether. There are two possibilities: a blocking and a non-blocking call. In the case of a blocking call, a python *Event* is used to wait for a response, which will be processed by the server. In the case of a non-blocking call, no such *Event* is created and after the sending control is immediately transfered. A non-blocking call is not an asynchronous call, as we wait for the actual sending to be finished.
Such an MPI command will be wrapped together with a specified *tag* that must be used to wrap the answer. These tags should be unique for the outstanding requests. Previously this was also done using the UUID module in Python, though this provided performance bottlenecks. Now, a form of queue is provided and there is a list of 'free slots' that can be used. This provides far better results than using UUID's and dictionaries to retrieve the value, also because a dictionary can have $O(n)$ amortized worst case, with n being the total number of elements that were *ever* in the dictionary. Since UUIDs don't get reused, this means that n can become arbitrarily large in huge simulations with lots of message passing.
If the call is made to the local kernel, a blocking call is simply a function call to the specified method. In case a non-blocking call was requested, this call will be performed on a seperate thread.
Together with this class, a 'fake' class *MPIFaker* is introduced. This class will act as an MPI implementation for local simulation if MPI4Py is not found and was necessary to allow python interpreters without MPI4Py or PyRO to function. Due to this class, PythonPDEVS is still perfectly usable on any system that could run the previous version of PythonDEVS (e.g. no dependencies are introduced).

Tracers

Just like the *Scheduler*, tracing previously happened within the transition functions themselves. This caused some maintainability problems as it wasn't completely obvious how to cleanly add different tracers or modify existing tracers. Due to the new *Tracers* class, all tracing (verbose, XML and VCD) happens here and the *Solver* class only makes calls to the tracer depending on which transition actually happened, together with the model that did the transition. Actual tracing happens by sending *actions* to the *Controller*, as to only execute them as soon as the GVT passed over this transition time. This is the reason why tracing seems to happen stepwise.
A further improvement to the tracers is that tracing now happens immediately to file instead of to an internal variable. By immediately writing to file, we are no longer constrained if the trace becomes to big to fit in main memory. Another advantage of this is that in the case of a failure, the finished part of the simulation will already be written to disk and will therefore not have to be checkpointed too.

Logging

Logging is done using the standard Python logging interface, which provides support for a Syslog server. Clearly, a Syslog server is far better for distributed simulation than writing to file or terminal because it allows all messages to be merged in a single file and provides some form of causality by default. Note that it is still possible for these network messages to arrive in a different order than how they were actually sent, though this is unavoidable in distributed simulation (without introducing overhead).
In production runs, logging should be unnecessary and would only incur a relatively high overhead. Even if the logging level would be set to *WARN*, a function call would still be required. In many situations, logging alone was responsible for a huge slowdown. This is mainly due to the very verbose logging that is supported (in *DEBUG* mode), but these calls are still made and thus cause an overhead. For this reason, logging statements are considered as *assertions*. While they are not actually assertions, they should require the same treatment in most situations: not being executed in production code, but executed while debugging. Of course, some logging statements (the most important) can be used without the *assert* keyword, to always use them.

NestingException

The *NestingException* class is a kind of exception that can be thrown in case nesting a new simulation failed. For more information about why this exception occurs and what to do with it, refer to section 3.9.2.

DEVSException

As in the previous version of PythonDEVS, all simulator specific exceptions are provided in the *DEVSException* exception. This can vary from bad models (e.g. negative time advance given) to problems in certain algorithms (e.g. rollbacks to a time before the GVT). If such an exception is thrown it likely signals a bug in the simulator or models.

3.4 Middleware

Our implementation supports two different kinds of middleware. These different middlewares were mainly implemented due to historical reasons, as the PyRO middleware was used at first. While an implementation using PyRO is relatively straightforward, performance was not very good. To work around this problem, MPI was also used. At the end, the modeller will now have the choice between PyRO and MPI (and 'no middleware' for local simulation). Since this research internship is mainly focussed on efficient simulation, most of our attention will be towards MPI and the user is encouraged to use MPI as well.
The format of the exchanged messages are those provided by PyRO or MPI respectively, meaning that they are incompatible with those of other simulators. This is not a problem as our efforts focus on efficiency instead of interoperability. Interoperability was the main focus in [36, 34, 23, 1].

3.4.1 PyRO

PyRO[13] (Python Remote Objects) was the initial choice due to the easy and (nearly) transparent way of accessing remote objects because it shares the same goals as RMI. As another benefit, PyRO is implemented in pure Python. This has the advantage that it is also cross-platform just like the rest of the simulator. The main disadvantage of PyRO is that it is relatively slow in comparison to different methods. PyRO also had several important problems and drawbacks during development[1], so the need for another middleware arose. Should the user want to use PyRO, we highly recommend to use version 4.11 as this is the version for which most development happened. Higher versions, like version 4.17 (currently the most recent), have major changes to (among others) the threadpool and performance dropped immensly, sometimes only using about 20% of the available processing power.
If the user wants to use PyRO even though the previous remarks, running it is fairly simple. First of all, a nameserver is required. The nameserver can be started with the command 'python -m Pyro4.naming'[2]. Afterwards, each server can be started by executing the `server.py` file with as parameter the servername. To comply with MPI and provide easy comparisons later on, it is highly recommended to use numbers starting from 1[3]. For example the command `python pypdevs/server.py 2` will start the second server.
After all servers are started, the actual experiment file can be started as usual, for example 'python experiment.py'. This will start server 0 and subsequently start the actual simulation. As soon as simulation is finished, all servers will exit.
PyRO was also used in another attempt to create a distributed version of the PythonDEVS simulator in [41].

Dependencies

The use of PyRO requires a version of PyRO installed on the system. Since PyRO received a major rewrite starting from PyRO 4, only versions above 4 are supported (they are incompatible). Futhermore, versions above 4.11 were tested and didn't show good performance, so the recommended version is 4.11.

3.4.2 MPI

The MPI (Message Passing Interface) implementation allows for much smoother and more efficient simulation. MPI is used extensively in parallel computing. Like the name suggests, it is based on message passing and thus provides no such transparency as PyRO did. To be able to offer MPI and PyRO at the same time, a small *wrapper class* called *MPIRedirect* was created. This class will implement a highly stripped down version of RMI using MPI by using the *__getattr__* function from Python. This wrapper class supports the most important features that are present in PyRO, specifically the distinction between blocking and non-blocking calls. To keep the interface lightweight, there is no such thing as true *asynchronous* calls with future results, as is present in PyRO. This means that the non-blocking calls may not have a return value (or if they have, the result is simply dropped).
As MPI itself is not a piece of software, but merely a standard, a choice had to be made for the actual MPI implementation too. We chose to use MPICH3 as it provided the best performance for our specific situation. Other possibilities include MPICH2

[1] Some of these problems might be due to PyRO 4 being relatively new, compared to the more stable PyRO 3
[2] Several extra parameters are necessary to run this nameserver on a network, for this we refer to the PyRO4 manual
[3] Actually, server 0 is also accessible, but this will run on the controller. This is also to be compliant to MPI conventions.

(which is the previous version of MPICH3 that is present in nearly all package managers of the most popular Linux distributions) and OpenMPI. Our implementation using MPI requires the MPI_THREAD_MULTIPLE[4] property, which is more efficiently supported in MPICH than it is in OpenMPI (OpenMPI provides no official support).
Another problem with MPI is that it uses busy polling by default in most implementations, causing huge performance drops as soon as oversubscription is done. This problem escalates due to the fact that there are still other operations to be performed while this busy polling is being done. Luckily, MPICH provides a specific device to use interrupts instead of this busy polling. Due to a slight misconfiguration in the official MPICH2 build script[5], it is impossible to use this device in combination with MPI_THREAD_MULTIPLE. This leaves MPICH3 as our best option. Note though that our simulator was tested on all these three different implementations.
Running the simulator with MPI is even easier than running with PyRO, as MPI will take care of the distribution of processes, e.g. through the usage of *ssh*. The user is only required to start the `mpirunner.py` script, with the experiment file to execute and the number of cores that is required.

Dependencies

The use of MPI requires an MPI binding in Python, which is not provided by default. This requires the *MPI4Py*[12] package, which has several other dependencies, like *Cython* and an *MPI implementation*. This MPI implementation must be built with MPI_THREAD_MULTIPLE, otherwise simulation might crash at random moments due to wrong assumptions in the MPI implementation.
In our benchmarks, *MPICH*[28] was used, though other standard conforming implementations (like *OpenMPI*[31]) also work.

3.4.3 None

For compatibility with previous versions, it is still possible to run the simulator as before, though it has the limitation that no distribution is possible. In this case, the simulator will use a built-in (stub) MPI implementation, thus there are no real mandatory dependencies on either PyRO or MPI except when distribution is desired. This makes it possible to use the simulator for small projects too, as it doesn't have any dependencies and doesn't require any special setup.

3.5 Cython

Due to the interpreted nature of Python, it is unlikely to achieve high performance compared to implementations in e.g. C code. For this reason, Cython[2] came into life. Cython is a Python-to-C++ translator which maps (most of) Python to equivalent C++ code. Though most of the time, this translation can be quite difficult due to the dynamic nature of Python. By default, it merely translates the Python code to Python API calls. The main advantage of this is that control structures can be rewritten in C++ code and typing information can be added. Furthermore, the C++ compiler can perform some more optimisations e.g. using the `-O2` optimisation flag.
To allow a much higher speedup, the Python language was extended to the Cython language, which allows (among others) static types to be declared. By declaring a static type, a call to the Python API is avoided and most operations can be converted to pure C. Certainly, it is worth taking a look at this 'free' speedup.

3.5.1 Translation

In an attempt to make this translation as painless as possible, a build script was created to perform the necessary operations of compiling the Cython module. Since PythonPDEVS is still in heavy development, it is required that this building happens automatically and in a flexible fashion. This means that multiple copies of the source code have to be avoided at all costs (even more than in normal situations). This flexibility is not easy, as all Cython-specific code is not compliant with default Python, so a Python interpreter would crash as soon as such an instruction is found. Keeping compliant Python code is one of the core requirements, so it was impossible to deviate from this path.
To make this compilation painless, we added a kind of *preprocessor* step in the build script. It is possible for the Python code to have different kind of comments, which will be hints towards the preprocessor. This completely automates the translation from Python to C++ code, without the need for manual type insertion each time a build is attempted. One of the disadvantages is that the code becomes slightly less readable due to the 'directive comments'. Luckily, most functions do not need specific Cython annotations, as only a few functions really benefit from the use of static types.

[4] See `http://www.mpi-forum.org/docs/mpi-20-html/node165.htm`
[5] See `https://trac.mpich.org/projects/mpich/ticket/1787`

directive	meaning
#cython	uncomment this line
#cython-remove	remove this line

Table 3.2: The different preprocessor directives

3.5.2 Pickling

A problem with Cython is that it translates classes to extension types, which are considered as built-in types by Python. This clearly has several advantages, like using a C-style struct instead of a dictionary lookup when accessing attributes. On the other hand, the Python interpreter has no knowledge about the internal structure of the object and cannot provide common utility functions. The most obvious of these is the *__reduce__* method, which provides Python with a function on how to create a pickle of this object.
Whereas normal objects can simply be pickled by calling e.g. *cPickle.dumps(obj)*, such extension objects will be unpicklable by default. Normally, this wouldn't be too much of a problem, as pickling is normally not used on e.g. a simulation kernel. However, due to our implementation of checkpointing (see section 2.7), we require such objects to be picklable. All other objects, like an Atomic DEVS model, are also required to be picklable due to dependencies on the simulation kernel and due to the fact that such models might have to be transferred over the network, introducing the need for serialization.
For this reason, most complex objects are not made into extension type classes and only the most trivial ones (Message, NetworkMessage) are.

3.5.3 Performance

Cython usually has a decent speedup, even in cases where no extra annotation is added. For example in [45], an increase of about 10% was noticable, of course depending on the models and the extent to which the compilation happened (only the simulator itself, or the models too). For local simulation, we also noticed some speedups by only compiling the simulator itself. On the other hand, distributed simulation actually seems to become slower in some situations when using Cython. This has a multitude of possible reasons, most of which are tightly linked to the overall non-determinism of the underlying network and the associated delays.
In time warp, rolling back is a very costly operation, as it effectively erases previous computation as they might be wrong due to causality. This erasure itself also takes time, making it possible that time warp is slower than sequential simulation. Therefor, it is interesting to minimize rollbacks. Should not a single rollback occur, the simulation performance will increase by a factor k, with k being the number of cores that is used[6]. Since Cython usually speeds up Python code, it is likely that it causes a different flow of events and it might actually cause a lot more rollbacks to happen. On the other hand, it could turn out the other way around and prevent even more rollbacks.
For example a generator and a processor, which live on a different LP. In the normal Python situation, only a few rollbacks happen because both run at approximately the same speed. In Cython however, the processor might be optimised a lot better than the generator, meaning that the processor will always be further in simulated time than the generator. This would cause (nearly) every incomming message from the generator to be a straggler and thus cause an immensive number of rollbacks. Since rollbacks are relatively costly, it is possible that the Cythonized simulator is actually slower than the original Python simulator, even though the Cython version does everything somewhat faster. Note that this problem is not only present when using Cython, but it is always possible that a slightly different timing or order of threads causes huge differences in actually measured performance.

3.6 PyPy

While Cython translates and compiles the Python code, PyPy[33] is an alternative interpreter that is compliant with Python. The most important feature of PyPy is the Just-In-Time (JIT) compiler, which allows it to translate hot spots in the code to assembly code, thus avoiding the interpreter. It also provides no possibility to add type declarations, as was possible in Cython, so a speed improvement is nearly always noticable without *any* code change.
Due to incompliance between our middleware (both PyRO and MPI4Py) and PyPy, we were only able to perform local simulation on a single core. Nonetheless, this is an interesting comparison as it allows us to determine the speedup that would be obtainable, simply by switching to a different interpreter.
Even though some parts become (a lot) faster in PyPy, some modules actually slow down under PyPy. This is mainly due to PyPy implementing these modules in Python instead of C as was the case in CPython. One of these slower modules is the *cPickle* module, which is one of the most time consuming parts of the distributed version (it is part of the state saving algorithm).
Furthermore, the same problem as in Cython is present, where a faster interpreter or system doesn't necessarily mean faster execution speed in a distributed setting due to the different timing and difference between rollbacks.

[6] Actually, even in this best case scenario a speedup k is impossible to reach due to the extra cost of message passing and state saving

As we are unable to use PyPy for distributed simulation, it will rarely be used in any of the following efficiency plots.

3.7 Recomparing to ADEVS

In [45], a comparison between both ADEVS[30] and PythonDEVS was performed in terms of performance on a (slightly altered) DEVStone[17] benchmark model. However, there were some differences between both simulators that didn't allow a perfectly fair comparison. For example PythonDEVS using the Classic DEVS formalism, whereas ADEVS uses the Parallel DEVS formalism[7]. Other differences were in the interface that was provided to the user, termination conditions and so on.
Now that PythonPDEVS was created, we also support the Parallel DEVS formalism, have a more similar interface and also have the possibility for a faster termination check. This way it is possible to provide a somewhat more fair comparison between the two. The only major difference between both simulators is the implementation language and the corresponding freedom to the modeller.
It should be noted that all simulation runs were performed on a single core and thus the results are compatible with the previously obtained results. Also, PythonPDEVS didn't have any more resources available than ADEVS or vice versa.
Figures 3.2 and 3.3 show the difference between PythonPDEVS using both CPython and PyPy, and ADEVS which is compiled using the `-O2 -march=native` GCC compiler flags. The test is ran for both wide and deep models, both of them try to avoid collisions as much as possible[8]. For reference, the sequential version of PythonDEVS is also shown.

3.7.1 Discussion

From figures 3.2 and 3.3 it becomes clear that PythonPDEVS became a lot faster than PythonDEVS, even though PythonPDEVS still has to handle a slight locking overhead because it also allows the modeller to simulate distributed models. The main reason for this huge difference is the changed API (see section 3.9) and the possibility to define a *termination time* instead of a *termination condition*. The use of parallel DEVS is not that important in these test cases, as collisions rarely happen and the avoidance of this collision check introduced a check for *confluent* transitions.
PyPy also offers a nice speedup compared to the ordinary CPython version, which didn't require any code change at all and is simply another interpreter. It should be noted that PyPy requires a warmup period for its JIT, which explains the shakey behaviour at the start.
At the end, ADEVS is still the fastest, though our PythonPDEVS implementation is closing in very fast. Even though PythonPDEVS is somewhat slower (approximately a factor 4 in this case), it allows the modeller to use Python as a modelling language. The main advantages and disadvantages of Python versus C++ are well known and they still apply here. The relevant advantages of Python are that it does not require compilation and that it allows dynamic typing.
Note that there is nearly no difference between wide and deep models, because all tested implementations employ *direct connection* (or a variant of it) which reduces deep models to wide models.

3.8 Performance

First of all, the used algorithms were profiled using *cProfile*[10] and *Line Profiler*[22] to make sure that no local algorithm in itself is responsible for a bottleneck. As can be seen in section 3.7, our algorithms are on par with those employed in ADEVS (up to a constant factor), so the main reason for further performance issues would be due to the actual distributed aspects, such as network latency or jitter, or the behaviour of our *time warp* implementation.
The main performance bottleneck in a *time warp* implementation is the amount of rollbacks that happen. First and foremost, a rollback requires some computation as to which state is the one exactly before the straggler message. If the rollback is from the distant past, a lot of states will have to be checked before the correct one is found. To make matters worse, every model at the LP that is being rolled back should be checked whether or not a rollback is necessary. Secondly, rolling back a model requires it to be rescheduled on the new time. This rescheduling is again proportional to the amount of models at the LP, though this time it is even $O(n \cdot log(n))$ in the average case. Thirdly, the messages that were sent in the meantime need to be unset. This incurs network messages and possibly causes even further rollbacks. Finally, the input messages also need to be rolled back, as some might have already been sent into the model.
Clearly, *time warp* is an optimistic protocol in the sense that it assumes that this worst case will rarely happen. Whether or not this is the case, is highly dependent on both the model and the heterogenous nature of the different machines (including the intermediate network). These aspects will be discussed next, though first a general remark is made about how all further plots should be analyzed.

[7] Actually, ADEVS uses the DynDEVS formalism, though the dynamic structure features are never used in the benchmarks, therefore reducing it to Parallel DEVS.

[8] This is due to the quadratic behaviour of PythonDEVS when collisions occured. While not exactly necessary, this is done to keep consistency with the previous comparison in [45]

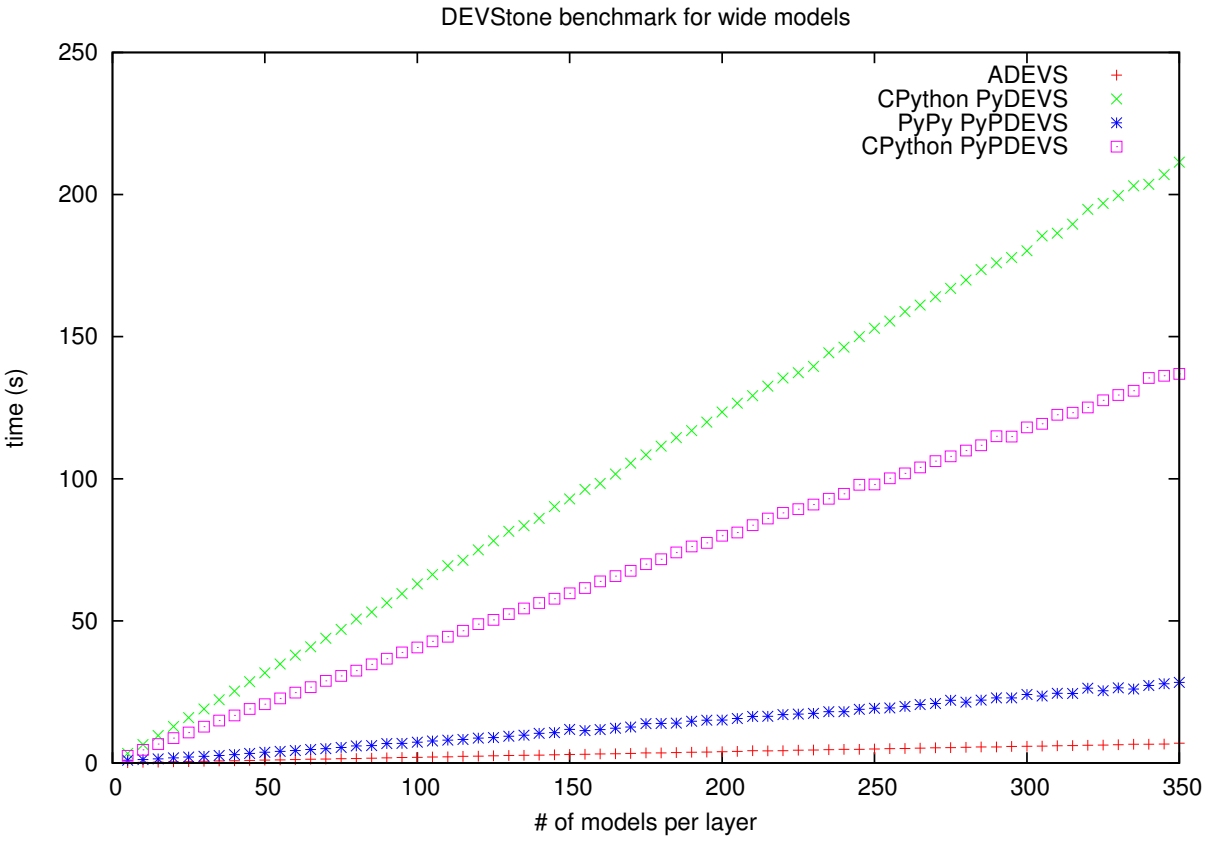

Figure 3.2: DEVStone benchmark for wide models

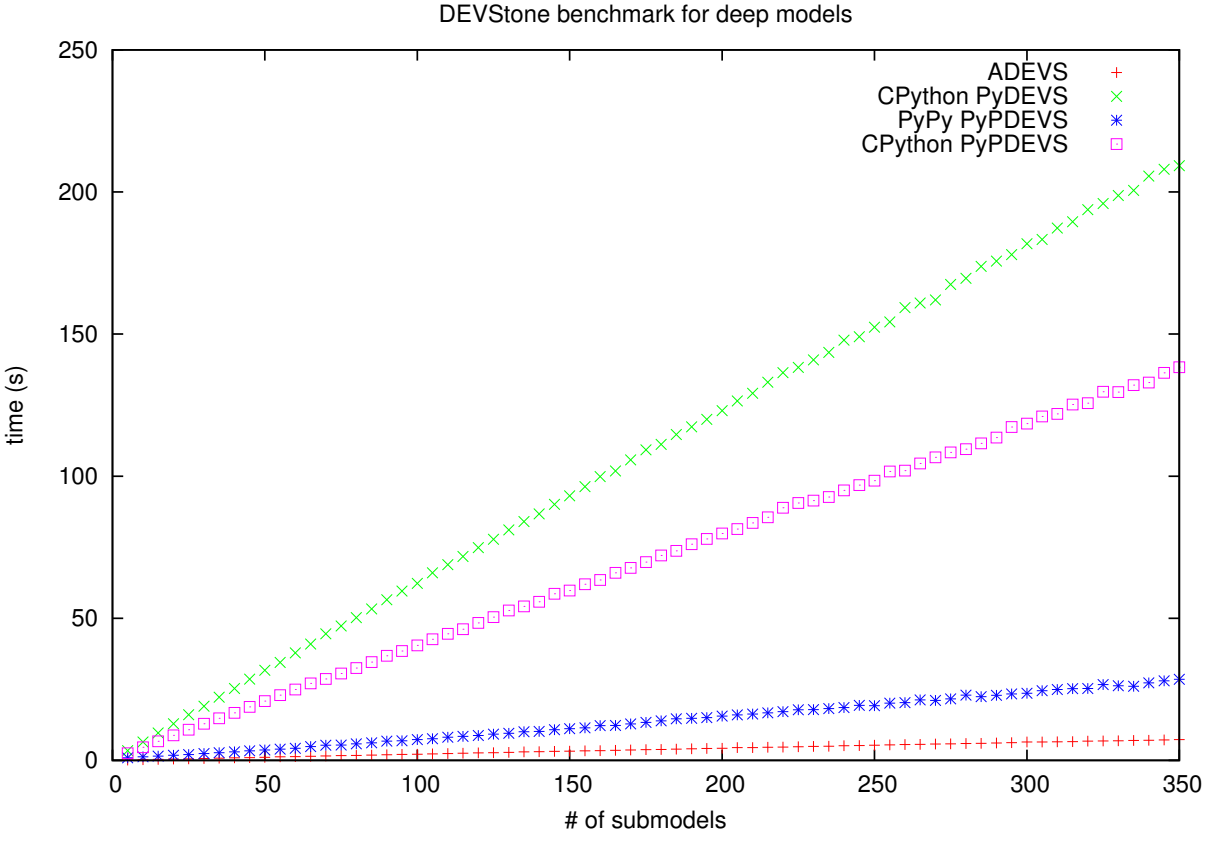

Figure 3.3: DEVStone benchmark for deep models

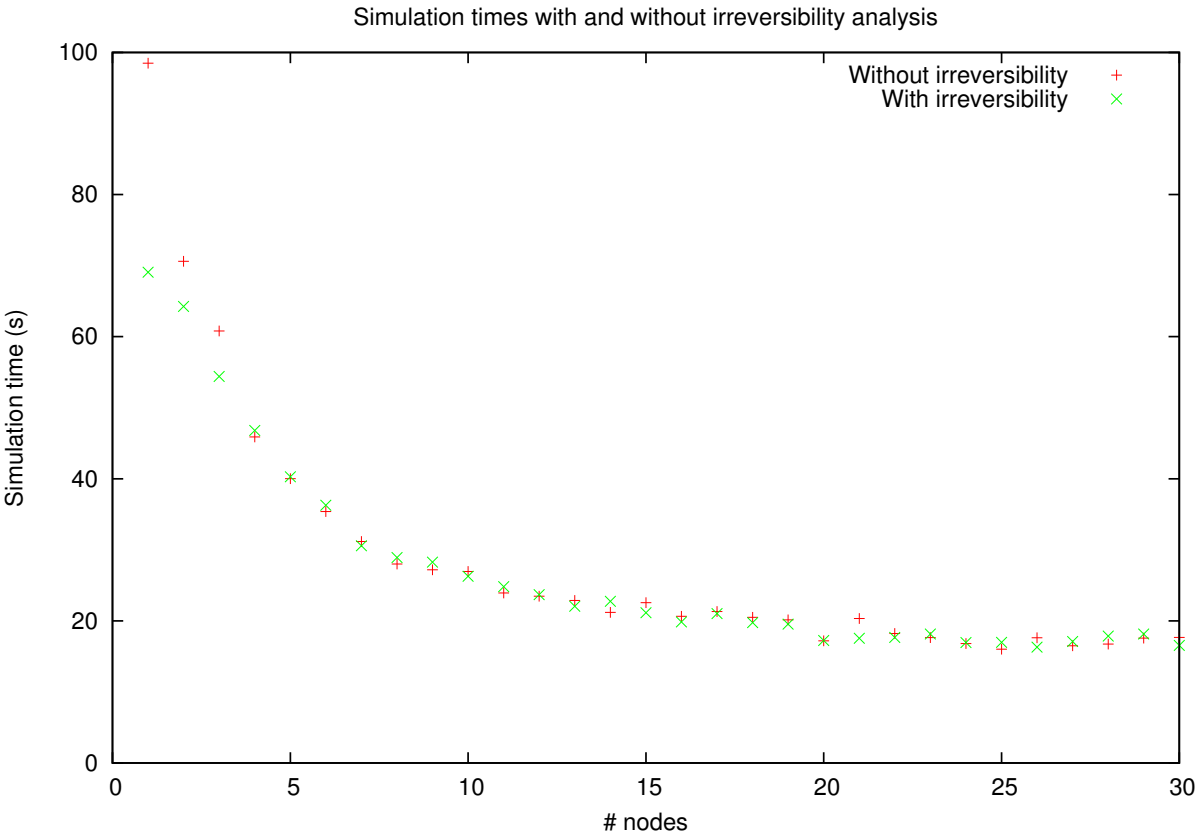

Figure 3.4: Comparison between with and without irreversibility

3.8.1 Irreversibility

As mentioned in section 2.4, PythonPDEVS uses an optimisation called *irreversibility*. Due to this, efficiency plots can become rather skewed, as the sequential case is always able to exploit this irreversibility. Though it is part of the definition of *efficiency* to compare with the *best* sequential implementation, efficiency plots without this irreversibility will also be shown. This offers more insight in where the 'problem' lies and what is the actual impact of this small optimisation.
In some cases, irreversibility can also be achieved with distributed models, which then gives very high efficiencies. Models that are unable to achieve irreversibility will have to do a lot of additional work (all the overhead of time warp) and will thus be inefficient compared to the sequential case in nearly all situations.
This does not mean that adding more nodes necessarily slows down the simulation, it only means that adding them might not speed it up enough to compensate for the additional time warp overhead. Therefore it would be unwise to compare a sequential simulation against a distributed simulation with only two nodes and expect a major speedup, without keeping these remarks in mind.
Figure 3.4 clearly shows the difference between with and without irreversibility. Notably, an actual difference is only noticable if only a few nodes are used, because only the atomic models at the first node will be irreversible. If the number of nodes increases, the number of atomic models at a certain node decreases. There is some slight jitter when more nodes are added, which is due to the usual non-determinism of delays when sending and receiving messages, but also slightly due to timing differences caused by irreversibility.
The model that was simulated for this plot has very light work during the transition functions to clearly show the difference. For heavy work models, the difference is approximatly equal in absolute time, though the relative impact decreases.
Due to this analysis, efficiency will always be close to 1 if only a single node is used. Normally the overhead from time warp would be added, causing efficiencies that are somewhat lower. Such a situation could be observed in [40], where the speedup dropped to 0.5 in the worst case.

3.8.2 Model

The model is one of the primary causes of performance (or the lack thereof) in a distributed simulation. Since *time warp* is an optimistic protocol, it will perform very well in situations where nearly no synchronization is necessary, but it will perform very

bad (or even worse than a sequential simulation) in cases where a lot of synchronization is necessary. Several different factors influence the actual performance that is achievable by using *time warp*, though most of them boil down to minimizing the amount of rollbacks.
In case a conservative protocol would be used, the performance would be even more dependent on the model[15], to the extent that slight changes to the model can give drastic changes to the performance.

Lookahead

The amount of lookahead is an important consideration for performance, even though this is not stated that explicitly as it is in *conservative simulation*, where several algorithms even fail when the lookahead becomes zero. For *time warp*, a loop with a lookahead of zero indicates that a model *could* be indirectly dependent on the message it just sent, and that this dependency occurs at the 'same' time[9]. The algorithm will progress the simulation time after the message was sent, but after some time, the zero lookahead message will be received, causing a rollback to the time right after the message was sent. So this means that all computations that happened in between were useless and should be removed. This causes delays due to the computations that are ran in the meantime (they aren't force quit as soon as a message is received) and mainly due to the rollback algorithm.
So while a zero lookahead is not problematic for the algorithm itself (as it is in e.g. the *null message algorithm*), performance is likely to suffer in situations where such loops are frequently used. Of course, if a zero lookahead loop exists, but is rarely used, it will have nearly no impact on the performance due to the optimistic nature. In the *null message algorithm*, this special case would have to be taken into account and performance would suffer, even if the link is only used sporadically[10].

Message passing

It is well known that network latency and the associated inter-node message passing causes high overhead. As soon as a message is passed, it has to be converted internally in the simulator, afterwards the middleware will need to pickle the message before handing it off to the middleware (MPI or PyRO). As soon as it is received at the receiving end, the message will be depickled again and converted by the simulator again. Finally the message needs to be queued for processing. Furthermore, the message also needs to be taken into account by the GVT algorithm. The conclusion is that message passing is relatively expensive and can even become very expensive in case the message is a straggler.
In other cases, where nearly no message passing is done, the efficiency will be close to 1. Although this should be common sense, a model as represented in figure 3.6 is constructed and simulated. On the horizontal axis, very few messages are passed, whereas a lot of messages are passed on the vertical axis. To demonstrate the massive influence message passing has, this model is simulated in three different setups:

1. **Completely sequential**: only a single LP, making no use of time warp or parallellisation at all. Note that this situation can thus profit from *irreversibility*.
2. **High message passing**: two LPs are constructed, LP_1 has models 1 and 3, LP_2 has models 2 and 4. Every time step, messages will be exchanged between both LPs, meaning a massive amount of rollbacks and corresponding anti-messages will be necessary, whereas there is nearly no local computation or message passing.
3. **Low message passing**: two LPs are constructed, LP_1 has models 1 and 2, LP_2 has models 3 and 4. Only a few messages are exchanged, so rollbacks will be very infrequent. Even if a rollback occurs, (nearly) no anti-messages will be needed.

The run time plot in figure 3.7 compares these three different setups. Clearly, the high message passing model is extremely inefficient due to very frequent rollbacks and message passing overhead. As the number of messages on the low message link increases, both distributed models converge, which is logical since the low message link will then become as frequently used as the high message link. Note that performance of the distributed model is actually of much better efficiency in the (close to) *ideal* situations, which occurs when only a few messages are being passed. The efficiency in this close to ideal situation is shown in figure 3.8. In these ideal circumstances, a relatively large portion of time is spent in initialization of models over the network, which is somewhat slower then simple local initialization. Despite this, no longer simulations were used to keep a clear distinction between the high and low message passing situation.

Transition function workload

Another aspect of the model that influences performance, is the amount of work that is done in the models. Clearly, if a model does nearly no processing, the main bottleneck will be in the message passing and processing. To highlight this situation, figure 3.5 shows a plot of the performance that was achieved for models that have an increasingly big amount of processing. This

[9] Remember that the time will be extended with an *age* field to prevent two times being exactly equal
[10] Several other conservative algorithms do no suffer from this problem to the same extent, or some solutions to this problem exist, like using adaptive lookaheads

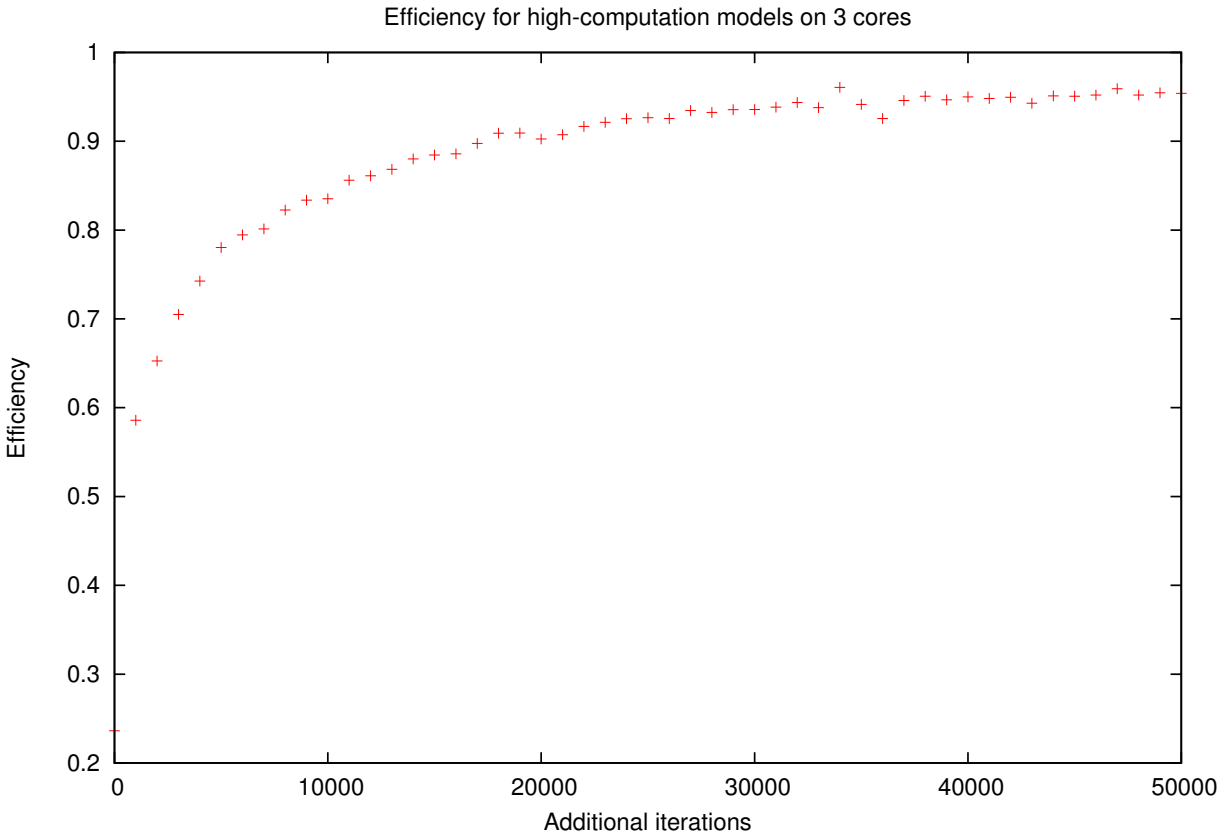

Figure 3.5: Efficiency for increasing amounts of computation in the model

processing is simply implemented as an empty loop with a growing number of iterations. As long as the interpreter is the (naive) CPython interpreter, this will actually do this number of iterations and thus do some computation. Even though it is mentioned that these are '*heavy*' models, 70000 iterations take only a microsecond on the machine that ran this benchmark. Due to this higher amount of work that needs to be done, the overhead from message passing drops drastically and correspondingly, the efficiency approaches 1. Furthermore, the number of stragglers will also decrease, preventing a lot of rollbacks.

Different simulation pace

Another reason for varying performance is not that straigthforward, though it is typical of *time warp*. At the start, both LPs run at about the same pace (they all have the same number of iterations) and they will be at approximatly the same simulated time. After some processing, a straggler arrives (caused by the message passing delay) and a rollback occurs at the second LP. In the first LP, processing can simply continue as no causality error happens. In the second LP, the rollback will undo a lot of computation, which will afterwards have to be redone. Because the work that must be redone is bigger, the second LP will be unable to progress in simulated time before the first LP (which is still generating events at maximal speed). Since the second LP can't catch up, it will rarely need rollbacks, simply because it has rarely done too much computation. In the cases where no or nearly no additional computation is done, the difference between both LPs becomes much smaller, thus it becomes possible for the second LP to catch up. Note that this argument is only applicable in models without feedback loops.
The figure clearly shows that the actual efficiency is influenced by slight details in the model, such as time it takes in the models functions. Though this influence is not as strong as it was in conservative simulation.
Note that this model has the possibility to use *irreversibility* analysis, as mentioned in section 2.4, providing an even higher performance.

3.8.3 Heterogeneity

Even though the model does not uses a lot of message passing, it is still possible for one LP to run on a faster machine than another. If this is the case, the fast machine might already be in the far future compared to the slow machine. While such situations would be problematic in *conservative* simulation too, the fast machine will now have to check a lot more states, resulting in slower rollbacks. Additionaly, all messages that were sent in the meantime will have to be rolled back, which possibly went to

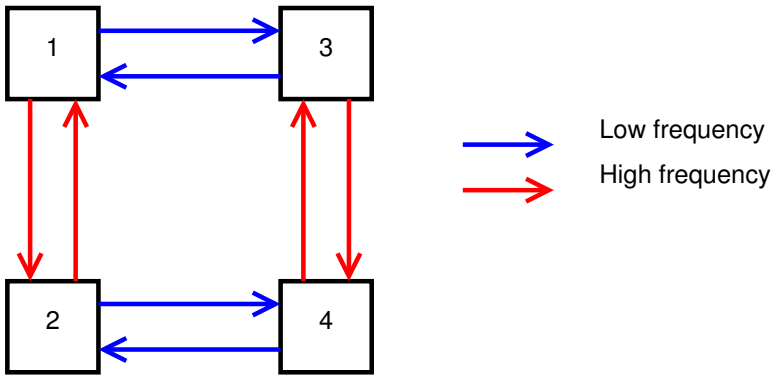

Figure 3.6: The model for the message passing test

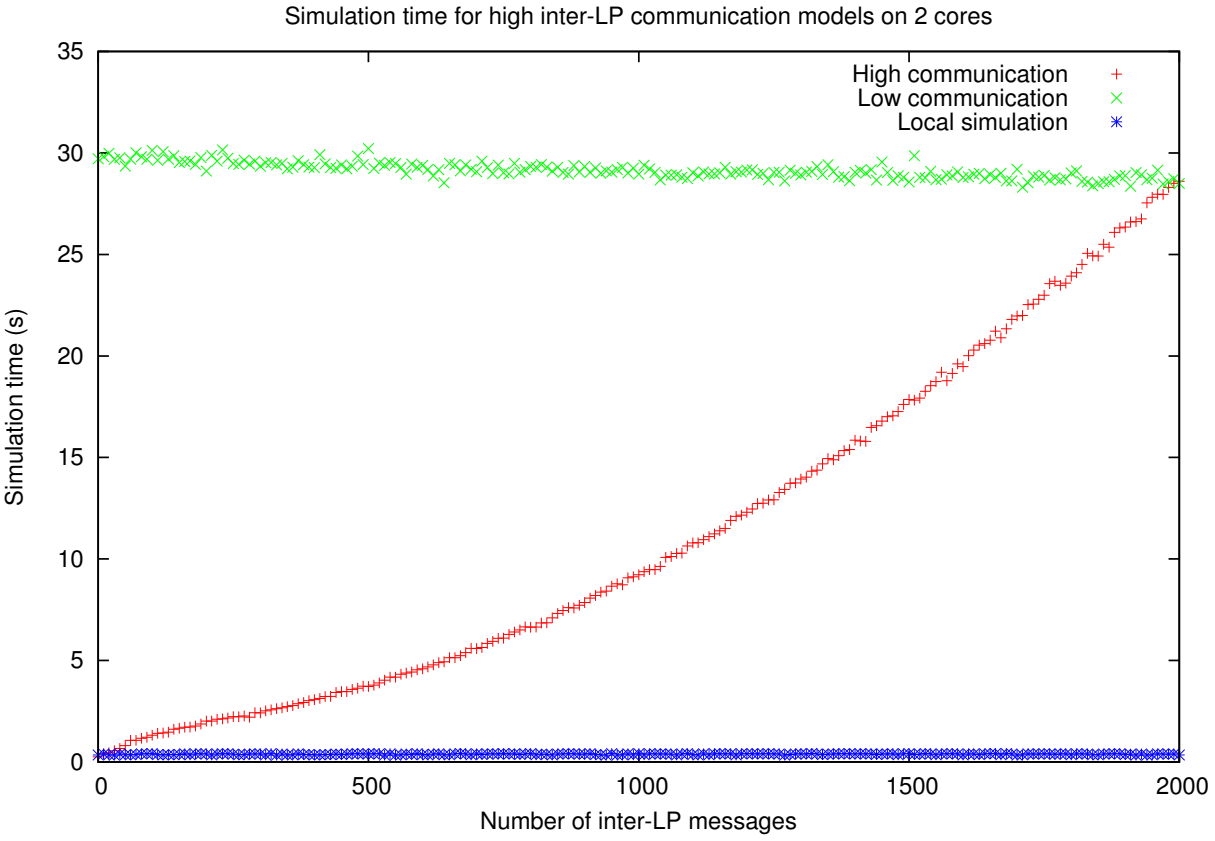

Figure 3.7: Simulation time for increasing amount of inter-LP message passing, results from model 3.6

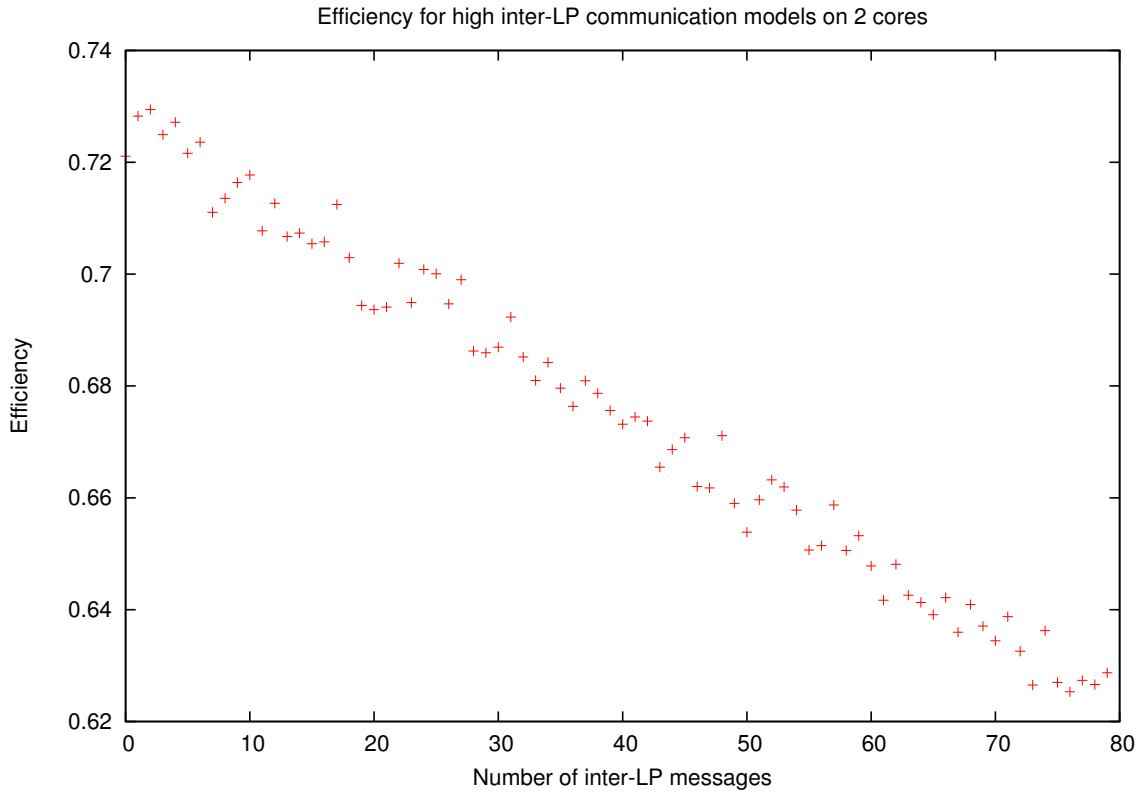

Figure 3.8: Efficiency for increasing amount of inter-LP message passing, results from model 3.6

the slower LP. In that case, the slow LP will have to do additional processing to invalidate all previously received (though not yet processed) messages.
The load on the machine on which the LP runs is also of utmost importance. It is very counterintuitive to have a simulation that actually runs *faster* as the load from other processes on the same core increases. This is due to the fast LP being throttled and thus requiring a lot less rollbacks. In situations where the difference is huge, the amount of rollbacks can become immense.
This effect is closely linked to the effect shown in figure 3.5, as it is comparable to a higher load.

3.8.4 Efficiency with increasing number of nodes

To show the performance with an increasing number of nodes that take part in the simulation, a small model is created that balances the atomic models over the number of available nodes. This model is again based on a simple buffer, which passes the message after some processing. In total, 100 atomic models are present, which are equally distributed over the available nodes. For example: when using 20 nodes, each node will have a coupled model with 5 atomic models in it. The efficiency that was obtained in these simulation runs can be seen in figure 3.9 and 3.10.
A distinction is made between the version where irreversibility (as mentioned in section 2.4) is used and another where it is not used. Irreversibility can speed up this kind of models, though the amount of speedup that can be achieved is dependend on the number of models that is a submodel of the irreversible kernel. In this situation, it didn't make a real difference, because custom copy functions were defined that are several factors faster than the default *pickle* library.
This simulation was run on a 30 node cluster, each containing an *Intel Core 2 6700 @ 2.66GHz*, with each process ran on a physically different system. On this machine, the number of iterations correspond to 0, 60, 125 and 200 μs respectively.

3.8.5 PHOLD benchmark

Due to the previous considerations about the model and the high impact that this can have on performance, benchmarking and comparing time warp implementations becomes extremely hard. In the literature, the PHOLD[14] model is often used as a benchmark, for example in [7, 29]. Other benchmarks, which are often more suitable for the implementation being benchmarked, include models for fire spreading[40, 26] or the game of life[18].
Our implementation of the PHOLD model places a single atomic model on each different node, as shown in figure 3.11. Each model starts with a single event that is has to process and after it was processed, the event will be sent to another, randomly

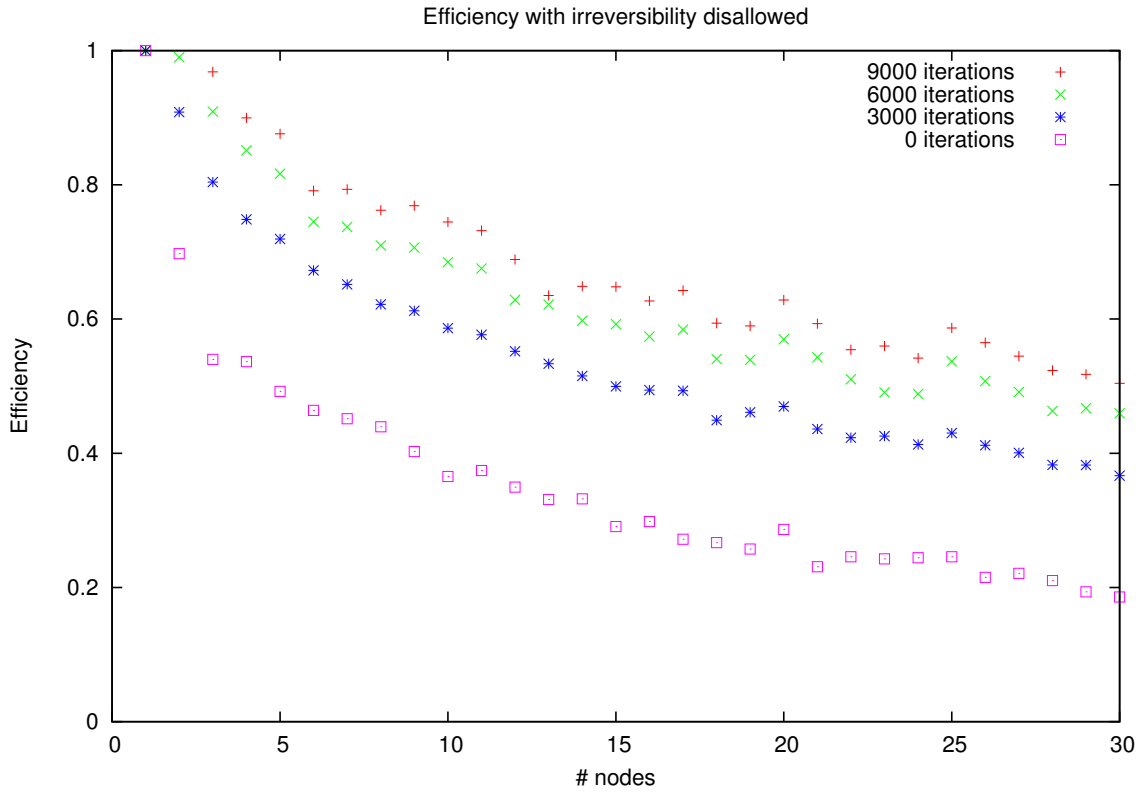

Figure 3.9: Efficiency with irreversibility disabled

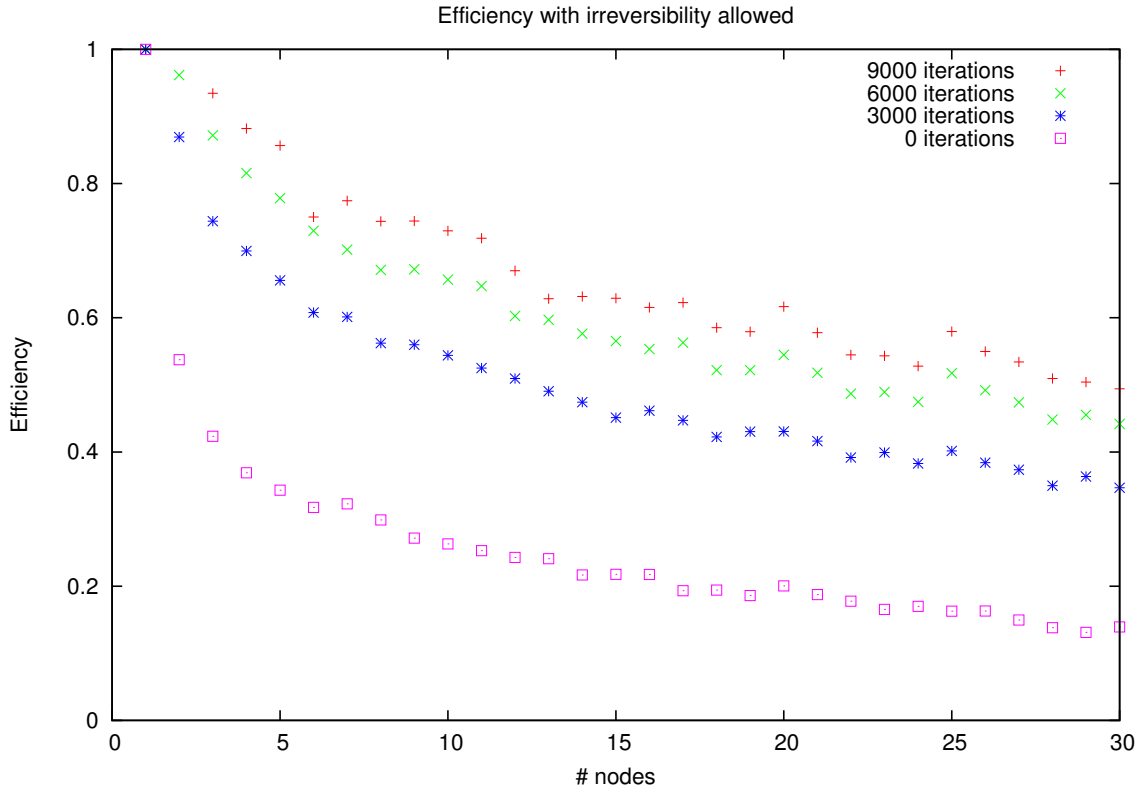

Figure 3.10: Efficiency with irreversibility enabled

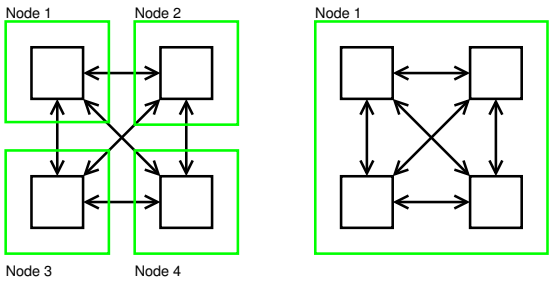

Figure 3.11: The PHOLD model, both the distributed version and the local version

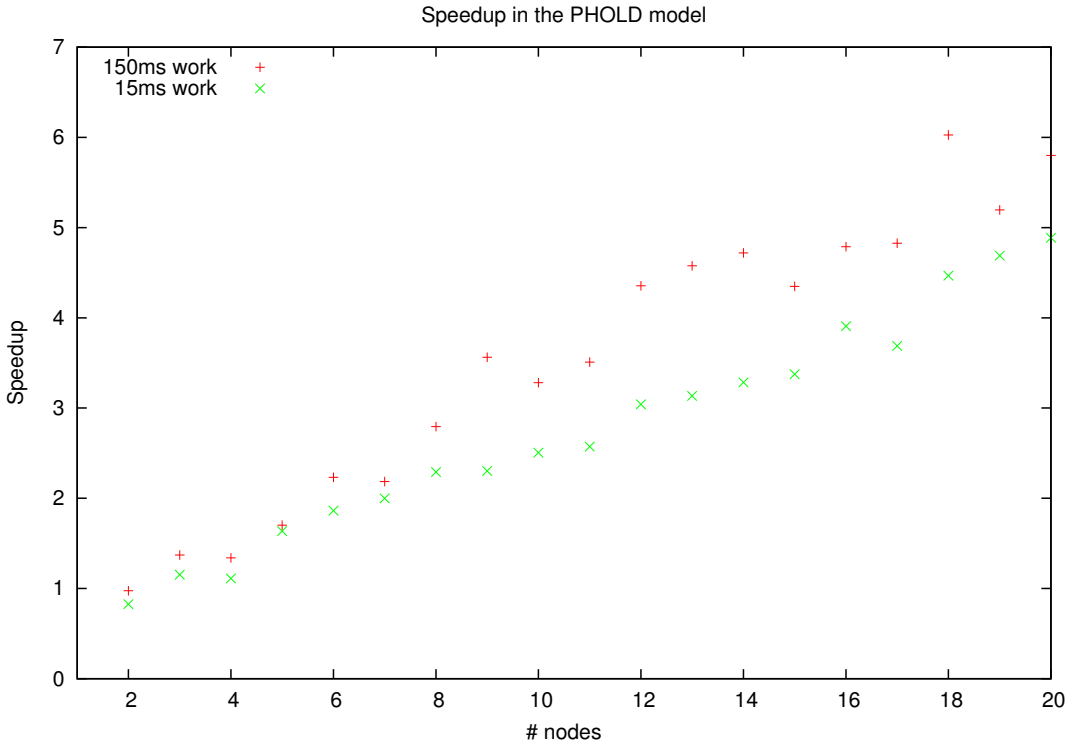

Figure 3.12: PHOLD benchmark for varying amounts of work in the external transition function

chosen, atomic model. The special case where a node sends an event to itself is not allowed. The processing time of each event is determined at random (in simulated time), whereas the actual wall clock time to process events is a variable parameter and will be varied to show different situations. A note about the 'random' numbers is that they are determined based on the value of the event that is being processed. This is done to introduce some determinism in the actual simulation trace, since actual random numbers use some kind of a *global state*, more specifically the seed.
Our simulations were run with 100% remote events, which means that every event will be passed to another node and thus will have to pass through the complete message passing routines.
Even with this default benchmark model, wrong comparisons could still happen due to a simulator optimizing some specific parts of the simulation. Furthermore, factors apart from the simulation algorithms influence the performance, such as the network delay, implementation language, processing speed of different nodes, ... The actual results of several simulation runs are visible in figure 3.12, each of these series use a different amount of work that is required in the external transition function[11]. This way, it is possible to compare different kind of models at once. In [40] it can be seen that the speedup can be immensly influenced by the amount of work that is being done, though this is done in a different kind of model. This same relation can be seen in our PHOLD benchmarks, though with a somewhat smaller difference[12].
Some fluctuations are visible in the figure, mainly due to the overal non-determinism of distributed simulation, but also due to the random aspect of the model: it might very well be that some nodes are not used for a long time due to some *coincidence*.

[11] We choose workloads in the order of tens of microseconds due to the choice for 100ms in [18]
[12] This could be partially due to the different type of model, but also due to the *vague* description of how much work is actually being done.

However, the global trend is visible.
It should also be noted that PHOLD is a relatively bad model to parallellize, mainly due to the high amount of message passing that is required during simulation. So while performance can be relatively bad in this benchmark model, this does not necessarily mean that some (better to parallellize) model will perform at the same level of efficiency. As was seen previously in section 3.8.2, efficiency can be relatively high in better situations.

3.9 Programmers reference

Due to the new formalism, the possibility to distribute models and several other enhancements, the API has changed drastically since [45] (which was still completely compliant with the original specifications mentioned in [3]). Most old models will no longer function, mainly due to the switch to Parallel DEVS. Several API changes were needed due to the distribution of models, though they stay compliant with the previous version where possible. E.g. accessing a state of a remote model is not as simple as it was in the sequential situation.
The appendix contains several examples of models wich use this new API, with the updated lines of code marked in red. So we avoid giving real code examples in this section.

3.9.1 Simulator

The core simulator is the object that is created in the experiment file and is the object on which *simulate()* is called to start the actual simulation. The interface was rather slim, so a few additions were necessary.

registerState(variable, modelname)

Parameters:

- *variable*: The name of the variable to initialize with the state
- *modelname*: The fully qualified name of the model whose state should be fetched

As previously mentioned, the state of a remote object cannot be accessed from the experiment file. Sending all states to the machine running the experiment would be inefficient as the state is often unneeded, introducing unnecessary delays. Furthermore, if the states were to be sent, it could be possible that the controller runs out of memory due to a very large model being simulated. The lack of memory on a single system might have been the primary reason to distribute the model in the first place. This caused the creation of the *registerState(variable, modelname)* method. This method has no return value, the actual state will only be available after the *simulate* call. Note that this registration must happen *before* starting the simulation, as all models will be destroyed as soon as simulation finishes.

simulate(...)

The *simulate* method starts the simulation and can be used to provide configuration parameters. This configuration is not provided in a configuration file, as different (subsequent) simulation runs might require very different configurations. The following options are supported:

- *termination_condition*: A function which takes two parameters (a float containing the current simulation time and a model containing the root model at the specified node) and returns a boolean. If this boolean is True, simulation will stop, otherwise, simulation will progress. If this function is defined, it will override the *termination_time* variable. Note that this function should only access states **local** to the node running this check, as other states are not guaranteed to be correct due to time warp. (default: *None*)
- *termination_time*: The global time at which simulation must halt. Providing a time is much more efficient than providing a termination condition and should be used if possible. (default: *INFINITY*)
- *verbose*: Whether or not a human-readable verbose trace should be generated (default: *False*)
- *xml*: Whether or not an XML trace for the XMLTracePlotter[38] should be generated. The trace will be saved to `devstrace.xml`. (default: *False*)
- *vcd*: Whether or not a VCD trace should be generated. The trace will be saved to `devstrace.vcd` (default: *False*)
- *termination_location*: The node at which to run the *termination_condition*. If the *termination_condition* is *None*, this value will be ignored. (default: *None*, only needed when a termination_condition is given)
- *logaddress*: The location of the Syslog server for logging, in the format of a tuple containing the location, followed by the port number. UDP is used for communication. (default: *('localhost', 514)*)
- *loglevel*: The level of detail at which logging should happen. Should be one of the logging levels defined in the *logging* Python library. (default: *WARN*)
- *outfile*: A file to save the verbose trace to, if *None* is provided, this output will go to stdout. (default: *None*)

- *GVT_freq*: The interval in seconds between successive GVT calculations. This value is only an approximation, as this is measured between the end of the previous run and the start of the next run. This is to prevent multiple simultaneous GVT calculations, as there is no bound on how long one run can take. Value must be at least 1. This parameter can be important for performance, as it also determines the fossil collection frequency. (default: *1*)
- *CHK_freq*: The number of GVT calculations to wait before creating a checkpoint. Setting this to a negative number will disable checkpointing. (default: *-1*)
- *state_saving*: The way to save the state of a simulation run. Several options are available, of which 2 is the fastest that is guaranteed to be safe. For high performance users, option 5 is recommended, though it requires some additional code. Other options are present for specific cases, which shouldn't be necessary most of the time. (default: *2*):
  0. Use *deepcopy*: allows all kinds of states to be copied, though it is very slow
  1. Use *pickle* with protocol 0: faster state saving in compatibility mode (if the state must be transferred to an old Python interpreter). This uses cPickle if available.
  2. Use *pickle* with highest available protocol: fastest *guaranteed to be safe* state saving. This uses cPickle if available.
  3. Use *copy*: this creates a shallow copy of the data, not adviced unless you know what you are doing.
  4. No copy: Just reassign the value. This is only considered safe if the state is a primary type.
  5. Manual copy: A user-defined *copy* method on the state will be called, use this for the fastest (safe) state saving, though safety is dependent on the users definition. Some shortcuts can be taken if the user knows that some parts of the state will not be altered. Even in cases where no such shortcuts exist, a manual copy function is almost always several factors faster than using *pickle*.
- *seed*: The value to use to seed the random number generator on each kernel. Note that using random numbers in time warp is not deterministic at all, since the seed will not be reset during a rollback. (default: *1*)
- *manualCopy*: Whether or not a manual copy function is defined for all events that get passed. An exception will be thrown if this parameter is *True* but no such function is found. The function is the same as in the previous version of PythonDEVS and is just a *copy()* method on the message that gets passed. This is merely a 'speed hack' as it can save a lot of time, though it must be enabled on **all** messages being passed. If primary types are being passed, they should either be encapsulated in a class to allow the definition of a *copy* method, or otherwise this parameter should be set to *False*. If set to *False*, *pickle* will be used. (default: *False*)
- *allowNested*: Set to *False* to prevent all necessary checks for nesting. Allowing nesting is relatively expensive as it requires some extra logic at *every* call to a user function. Setting this to *False* will disable all such checks, this is also a 'speed hack' as it allows to take some shortcuts. (default: *True*)
- *realtime*: Whether or not realtime simulation should happen or not. Realtime simulation cannot be used in distributed simulations, trying to do so will result in an exception to be thrown. (default: *False*)
- *realTimeInputPortReferences*: Provides a mapping between a string that can be used to identify an input port and an actual reference to this port in the simulation. Both of them have no limitatinos to them, so even internal ports can be used. It is in the form of a dictionary, where the key is the string that can be used and the value is the actual port. (default: {}, only needed when realtime is *True*)
- *generatorfile*: The file to use as input for the realtime simulation. It should contain text in the format *TIME PORT MESSAGE*, where *TIME* is the real time at which the message should be generated, *PORT* is the port mentioned in the *realTimeInputPortReferences* parameter and *MESSAGE* is the actual message to be sent. This message will be sent as a string. Can be *None* to indicate that no input file should be used, only command line input is then possible (default: *None*, only needed when realtime is *True*)
- *subsystem*: The subsystem to use to actually perform the waiting. The Python subsystem uses the Python *threading* library, whereas the Tkinter subsystem uses the event list of Tk. Options are either "python" or "tkinter". (default: "python", only needed when realtime is *True*)

execute(model, action)

Due to time warp possibly executing several functions multiple times, irreversible operations cannot simply be done as usual by the modeller. Therefore, the *execute* command (found in the *util* package) was introduced. This function will guarantee processing of the *action* as soon as it is certain that this action will never need to be reversed. All tracing performed by the simulator is also done using this function.
Note that the action should be a string with a python command that is executable without any other context, as this string will be executed using the python *exec* instruction.

3.9.2 Model

The introduction of Parallel DEVS requires a different interface from the models, so these changes are highly likely to be necessary in every model that was created for PythonDEVS. Apart from these changes, the *peek* and *poke* functions were removed due

to performance reasons. Profiling showed that these function calls were one of the biggest bottlenecks in the current implementation. Their removal is one of the primary reasons for the performance increase mentioned in section 3.7. Furthermore, removing *peek* and *poke* allows the user to write much more efficient models, as it is no longer necessary to *peek*/*poke* every port, but it is possible to iterate over the provided dictionary.
Due to the distributed simulation, it is also required to *announce* the fact that a new simulation starts within the current simulation. This makes nested distributed simulation possible.

extTransition(inputs)

The main change to the external transition function is that it no longer uses *peek*, but it takes a parameter with an input dictionary. This dictionary is indexed with the port on which the data is available, if a port did not receive any data, it will not be present as a key. To comply to the Parallel DEVS formalism, the values in the dictionary are lists instead of simple elements, as it is now possible to receive multiple events on a single port. Note that the formalism states that these lists are actually bags, so the user should not count on any determinism in the order of these elements. As previously, the function returns the new state.

confTransition(inputs)

This function is new in the Parallel DEVS formalism and is called if an *external* and *internal transition* happen at the same time. It also takes a dictionary as input, with the same semantics as in the *extTransition* function. The default behaviour is to call the *internal transition* function first, followed by the *external transition*. This function also returns the new state to use.

outputFnc()

The output function can no longer use the *poke* function, but must simply return a dictionary similar to the dictionary that is received as input in the *extTransition* function. This allows the modeller to be much more flexible in constructing this dictionary and prevents the simulator from making specific assumptions about the outputted value. Note that the values in the dictionary **must** be lists, even if it is just a single element.

announceSim()

A call to this function must happen before the user starts to initialize new models as part of a nested simulation. If such initialisation happens before this call, they are probably denied and simulation will crash. Whereas the methodname makes it sound as if it is just an announcement, it is actually rather a request, as it is possible that multiple kernels request this simultaneously. For such situations, refer to section 3.9.2. If the call from *announceSim()* returned, the user if free to initialize new models in a different simulation. A call to *announceSim()* should eventually always be followed by a call to *announceEnd()* as soon as the nested simulation has finished.

announceEnd()

This call will indicate to the other kernels that they can continue processing all other models that they were stalling due to the nested simulation. As long as this method is not called, all kernels will still be reserved and thus performance will be lower if *announceEnd()* is not called as soon as possible.

NestingException

As soon as a nested simulation is attempted, it is possible that another model is already asking for such a nested simulation. This makes it possible that both kernels are waiting for each other, as they are both in user code and the kernels have no idea how to exit this code in case of trouble. Such a highly undesirable situation would normally cause a deadlock, since both kernels are blocking on the *announceSim()* method.
To prevent that situation, we introcude the *NestingException*, which can be thrown by the call to *announceSim()*. If this exception is received, it means that the kernel was unable to start a nested simulation and that it should try to start it again after some time, though first the simulation locks should be released for some time. This exception will cause an exit from the user code and will notify the kernel that it should reexecute this function as soon as the other nested simulation is finished. Note that this requires the calls to functions with an *announceSim()* call to be idempotent, unless the nested simulation is granted, as it might be restarted an arbitrary number of times. Only the code up to the *announceSim()* call should be idempotent, code after that will only be executed once unless a rollback occurs.

3.10 Conclusion

We explored the design and architecture of the new version of the PythonPDEVS simulator. As was to be expected from the title of this research internship, several performance plots were shown. We also compared with the previous, sequential version of PythonDEVS and ADEVS. To conclude, a list of all changes in the interface to and from the simulator was given.

A Example programs

In this chapter, several actual implementations using the new interface are shown with their major changes marked in red. These examples are also provided in the *examples/* subfolder in the source directory.
These examples are not meant to show real-life models, but rather to provide a small working example about how a model can be modelled using the new interface.
Since real time simulation is not a actual goal of this research internship, no example will be shown here, though an example is present in the `src/examples` folder.
For completeness, these examples are accompagnied with a quick reminder on how to run the different backends. In all cases, it is necessary to configure the desired backend in the `middleware.py` file. Only one of these should be set to `True`, the other to `False`. In case only local simulation is used, the MPI version should be set (this is due to the local version being a special case of the MPI version).

MPI

Starting MPI applications partially depends on the implementation, though the basic invocations are mostly identical. This can be done using the `mpirun` command, which should call a small wrapper script that sets up all required servers. An example `mpirunner.py` script is provided, which can be used in most situations, though sometimes some straightforward modifications could be necessary (like providing the model to be simulated).
A simple invocation could be `mpirun -n X mpirunner.py`, where X is the number of nodes and the `mpirunner.py` file is modified where necessary.

PyRO

In order to use PyRO, it is necessary to first start a nameserver. Detailed instructions in doing so can be found online at `http://pythonhosted.org/Pyro4/nameserver.html`. The default way[1] of setting up such a nameserver locally is using the command `python -m Pyro4.naming`.
After the nameserver is setup, all used servers should be started using the command `python server.py X`, with `server.py` being the file in the PythonPDEVS package and X being the desired server number. At the end, the experiment file can simply be called as usual, for example `python my_experiment.py`.

Local

To stay compatible with the previous version of PythonDEVS, it is also possible to start the simulation by simply executing the experiment file without any additional setup. The only limitation to this setup is that no distribution or parallellization is possible, though the Parallel DEVS formalism is still used. An example invocation would be `python my_experiment.py`.
This is the only method that is supported in case neither PyRO or MPI4Py is available.

[1] This will not use any interface except the loopback interface, other options allow for NAT traversal and hostname configuration

A.1 Example 1: Simple buffer

```
class Generator(AtomicDEVS):
    def __init__(self, name = "Generator"):
        AtomicDEVS.__init__(self, name)
        self.state = ModelState()
        self.outport = self.addOutPort("outport")
        self.inport = self.addInPort("inport")

    def timeAdvance(self):
        # Generate an event after a fixed delay
        return 1

    def intTransition(self):
        # Don't do anything
        return self.state

    def extTransition(self, inputs):
        # Never happens
        return self.state

    def outputFnc(self):
        # Output a new generated event on the output port
        return self.outport:  [Event(1)]

class Processor(AtomicDEVS):
    def __init__(self, name = "Processor"):
        AtomicDEVS.__init__(self, name)
        self.state = ModelState()
        self.inport = self.addInPort("inport")
        self.outport = self.addOutPort("outport")

    def timeAdvance(self):
        # Generate an event after a fixed delay
        return 1

    def intTransition(self):
        # Mark the state as having processed everything
        self.state.counter = INFINITY
        self.state.event = None
        return self.state

    def extTransition(self, inputs):
        # Process the input, don't take into account previous entries or multiple entries
        self.state.event = inputs[self.inport][0]
        self.state.counter = self.t_event1
        return self.state

    def outputFnc(self):
        return self.outport:  [self.state.event]

class Chain(CoupledDEVS):
    def __init__(self, ta):
        CoupledDEVS.__init__(self, "Chain")
        # Load all necessary files to make the constructor executable at the remote side
        imports = "from includes import *"
        # Put the Coupled Generator on node 1
```

```
        self.generator = self.addSubModel("CoupledGenerator(1.0)", 1, imports)
        # The following is also possible, though slightly less efficient
        # self.generator = self.addSubModel(CoupledGenerator(1.0), 1)
        # Put the first Coupled Processor on node 2
        self.processor1 = self.addSubModel("CoupledProcessor(5, 2)", 2, imports)
        # This is a local model (as in: at the root)
        self.processor2 = self.addSubModel(CoupledProcessor(0.30, 3))

        # Now connect the ports
        self.connectPorts(self.generator.outport, self.processor1.inport)
        self.connectPorts(self.processor1.outport, self.processor2.inport)

class CoupledGenerator(CoupledDEVS):
    def __init__(self, t_gen_event1 = 1):
        CoupledDEVS.__init__(self, "CoupledGenerator")
        self.generator = self.addSubModel(Generator("Generator"))
        self.inport = self.addInPort("inport")
        self.outport = self.addOutPort("outport")

        self.connectPorts(self.inport, self.generator.inport)
        self.connectPorts(self.generator.outport, self.outport)

class CoupledProcessor(CoupledDEVS):
    def __init__(self, levels):
        CoupledDEVS.__init__(self, "CoupledProcessor_" + str(levels))
        self.inport = self.addInPort("inport")
        self.outport = self.addOutPort("outport")

        self.coupled = []
        for i in range(levels):
            self.coupled.append(self.addSubModel(Processor("Processor" + str(i))))
        for i in range(levels-1):
            self.connectPorts(self.coupled[i].outport, self.coupled[i+1].inport)
        self.connectPorts(self.inport, self.coupled[0].inport)
        self.connectPorts(self.coupled[-1].outport, self.outport)
```

A.2 Example 2: Traffic lights

```
# Import code for DEVS model representation:
import pypdevs
from pypdevs.DEVS import *
from pypdevs.infinity import INFINITY

class TrafficLightMode:
  def __init__(self, current="red"):
    self.set(current)

  def set(self, value="red"):
    self.__colour=value

  def get(self):
    return self.__colour

  def __str__(self):
    return self.get()

class TrafficLight(AtomicDEVS):
```

```
def __init__(self, name=None):
  AtomicDEVS.__init__(self, name)
  self.state = TrafficLightMode("red")
  self.elapsed = 1.5
  self.INTERRUPT = self.addInPort(name="INTERRUPT")
  self.OBSERVED = self.addOutPort(name="OBSERVED")

def extTransition(self, inputs):
  input = inputs.get(self.INTERRUPT)[0]
  state = self.state.get()
  if input == "toManual":
    if state == "manual":
     return TrafficLightMode("manual")
    if state in ("red", "green", "yellow"):
     return TrafficLightMode("manual")
    else:
     raise DEVSException("unknown state")
  if input == "toAutonomous":
    if state == "manual":
      return TrafficLightMode("red")
    else:
     raise DEVSException("unknown state")
  raise DEVSException("unknown input")

def intTransition(self):
  state = self.state.get()
  if state == "red":
    return TrafficLightMode("green")
  elif state == "green":
    return TrafficLightMode("yellow")
  elif state == "yellow":
    return TrafficLightMode("red")
  else:
    raise DEVSException("unknown state")

def outputFnc(self):
  state = self.state.get()
  if state == "red":
    return self.OBSERVED: ["grey"]
  elif state == "green":
    return self.OBSERVED: ["yellow"]
  elif state == "yellow":
    return self.OBSERVED: ["grey"]
  else:
    raise DEVSException("unknown state")

def timeAdvance(self):
  state = self.state.get()
  if state == "red":
    return 60
  elif state == "green":
    return 50
  elif state == "yellow":
    return 10
  elif state == "manual":
    return INFINITY
  else:
```

```
      raise DEVSException("unknown state")

class PolicemanMode:
  def __init__(self, current="idle"):
    self.set(current)

  def set(self, value="idle"):
    self.__mode=value

  def get(self):
    return self.__mode

  def __str__(self):
    return self.get()

class Policeman(AtomicDEVS):
  def __init__(self, name=None):
    AtomicDEVS.__init__(self, name)
    self.state = PolicemanMode("idle")
    self.elapsed = 0
    self.OUT = self.addOutPort(name="OUT")

  def intTransition(self):
    state = self.state.get()
    if state == "idle":
      return PolicemanMode("working")
    elif state == "working":
      return PolicemanMode("idle")
    else:
      raise DEVSException("unknown state")

  def outputFnc(self):
    state = self.state.get()
    if state == "idle":
      return self.OUT: ["toManual"]
    elif state == "working":
      return self.OUT: ["toAutonomous"]
    else:
      raise DEVSException("unknown state")

  def timeAdvance(self):
    state = self.state.get()
    if state == "idle":
      return 200
    elif state == "working":
      return 100
    else:
      raise DEVSException("unknown state")

class TrafficSystem(CoupledDEVS):
  def __init__(self, name=None):
    CoupledDEVS.__init__(self, name)
    self.policeman = self.addSubModel(Policeman(name="policeman"))
    self.trafficLight = self.addSubModel(TrafficLight(name="trafficLight"))
    self.connectPorts(self.policeman.OUT, self.trafficLight.INTERRUPT)
```

A.3 Example 3: State Fetching and Nested simulation

```
class MultiNestedProcessor(Processor):
    ...
    def timeAdvance(self):
        self.announceSim()
        from simulator import Simulator
        model = Chain(0.66)
        sim = Simulator(model)
        sim.registerState("fetch_state", "Chain.CoupledGenerator.Generator")
        sim.simulate(termination_time = self.state.processed, verbose=False)
        result = max(sim.fetch_state.generated, 1)
        self.announceEnd()
        return result
    ...
```

Bibliography

[1] Khaldoon Al-Zoubi and Gabriel Wainer. Interfacing and coordination for a devs simulation protocol standard. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, pages 300–307, Washington, DC, USA, 2008. IEEE Computer Society.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[3] Jean-Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package pythondevs for classical hierarchical devs. Technical report, MSDL Technical Report, 2001.

[4] Bin Chen and Hans Vangheluwe. Symbolic flattening of devs models. In *2010 Summer Simulation Multiconference*, SummerSim '10, pages 209–218, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[5] Gilbert Chen and Boleslaw K. Szymanski. Lookback: a new way of exploiting parallelism in discrete event simulation. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation*, PADS '02, pages 153–162, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Gilbert Chen and Boleslaw K. Szymanski. Four types of lookback. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation*, PADS '03, pages 3–, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Gilbert Chen and Boleslaw K. Szymanski. Dsim: scaling time warp to 1,033 processors. In *Proceedings of the 37th conference on Winter simulation*, WSC '05, pages 346–355. Winter Simulation Conference, 2005.

[8] A.C. Chow. Abstract simulator for the parallel devs formalism. *AI. Simulation, and Planning in High Autonomy Systems*, 1994.

[9] 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. Society for Computer Simulation International.

[10] CPython. Python cprofile. `http://docs.python.org/library/profile.html#module-cProfile`, 2013.

[11] CPython. Python time complexity. `http://wiki.python.org/moin/TimeComplexity`, 2013.

[12] Lisandro Dalcin. Mpi4py. `http://mpi4py.scipy.org/`, 2013.

[13] Irmen de Jong. Pyro. `http://pythonhosted.org/Pyro4/`, 2013.

[14] R. M. Fujimoto. Performance of time warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

[15] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

[16] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[17] Ezequiel Glinsky and Gabriel Wainer. Devstone: a benchmarking technique for studying performance of devs modeling and simulation environments.

[18] Ezequiel Glinsky and Gabriel Wainer. New parallel simulation techniques of devs and cell-devs in cd++. In *Proceedings of the 39th annual Symposium on Simulation*, ANSS '06, pages 244–251, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Jan Himmelspach, Roland Ewald, Stefan Leye, and Adelinde M. Uhrmacher. Parallel and distributed simulation of parallel devs models. In *Proceedings of the 2007 spring simulation multiconference - Volume 2*, SpringSim '07, pages 249–256, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[20] Jan Himmelspach and Adelinde M. Uhrmacher. Sequential processing of pdevs models. In *In Proceedings of the 3rd EMSS*, pages 239–244, 2006.

[21] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, 1985.

[22] Robert Kern. Python line profiler. `http://packages.python.org/line_profiler/`, 2013.

[23] Jasna Kuljis and Ray J. Paul. A review of web based simulation: whither we wander? In *Proceedings of the 32nd conference on Winter simulation*, WSC '00, pages 1872–1881, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[24] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[25] Qi Liu and Gabriel Wainer. Lightweight time warp- a novel protocol for parallel optimistic simulation of large-scale devs and cell-devs models. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, pages 131–138, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Qi Liu and Gabriel Wainer. A performance evaluation of the lightweight time warp protocol in optimistic parallel simulation of devs-based environmental models. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 27–34, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.*, 18(4):423–434, August 1993.

[28] Mpich3. `http://www.mpich.org`, 2013.

[29] James Nutaro. On constructing optimistic simulation algorithms for the discrete event system specification. *ACM Trans. Model. Comput. Simul.*, 19(1):1:1–1:21, January 2009.

[30] James J. Nutaro. Adevs. `http://www.ornl.gov/~1qn/adevs/`, 2013.

[31] Openmpi. `http://www.open-mpi.org`, 2013.

[32] Pythondevs. `http://msdl.cs.mcgill.ca/projects/projects/DEVS/`, 2013.

[33] Pypy. `http://pypy.org/`, 2013.

[34] José L. Risco-Martín, Alejandro Moreno, J. M. Cruz, and Joaquín Aranda. Interoperability between devs and non-devs models using devs/soa. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, pages 147:1–147:9, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[35] Behrokh Samadi. *Distributed simulation, algorithms and performance analysis*. PhD thesis, University of California, 1985.

[36] Hessam S. Sarjoughian and Yu Chen. Standardizing devs models: an endogenous standpoint. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 266–273, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[37] Moon Gi Seok and Tag Gon Kim. Parallel discrete event simulation for devs cellular models using a gpu. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 11:1–11:7, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[38] Hongyan Song. Infrastructure for devs modelling and experimentation. Master's thesis, School of Computer Science, McGill University, 2006.

[39] Yi Sun and Xiaolin Hu. Partial-modular devs for improving performance of cellular space wildfire spread simulation. In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pages 1038–1046. Winter Simulation Conference, 2008.

[40] Yi Sun and James Nutaro. Performance improvement using parallel simulation protocol and time warp for devs based applications. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[41] Eugene Syriani, Hans Vangheluwe, and Amr Al Mallah. Modelling and simulation-based design of a distributed devs simulator. In *Proceedings of the Winter Simulation Conference*, WSC '11, pages 3007–3021. Winter Simulation Conference, 2011.

[42] Luc Touraille, Mamadou K. Traoré, and David R. C. Hill. Enhancing devs simulation through template metaprogramming: Devs-metasimulator. In *2010 Summer Simulation Multiconference*, SummerSim '10, pages 394–402, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[43] Alejandro Troccoli and Gabriel Wainer. Implementing parallel cell-devs. In *Proceedings of the 36th annual symposium on Simulation*, ANSS '03, pages 273–, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Yentl Van Tendeloo. Personal homepage. `http://msdl.cs.mcgill.ca/people/yentl/`, 2013.

[45] Yentl Van Tendeloo. Research internship 1: Optimizing pydevs. Technical report, MSDL, 2013.

[46] Voon-Yee Vee and Wen-Jing Hsu. Pal: a new fossil collector for time warp. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation*, PADS '02, pages 35–42, Washington, DC, USA, 2002. IEEE Computer Society.

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