TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Formalizing and Verifying TimeWarp with FOCUS

Max Breitling

TUM-I9744
SFB-Bericht Nr. 342/27/97 A
Oktober 97

TUM–INFO–10-I9744-150/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Formalizing and Verifying TimeWarp with Focus[*]

Max Breitling

Institut für Informatik
Technische Universität München
D-80290 München

E-mail: Max.Breitling@informatik.tu-muenchen.de
http://www4.informatik.tu-muenchen.de/

October 27, 1997

Abstract

The TimeWarp mechanism accomplishes an efficient synchronization between the components of a distributed, discrete event-driven simulator. Using an optimistic simulation strategy, the components of the simulator may calculate ahead locally, sending results to other components without waiting for any events produced by those components, ignoring possible causality problems. In case of an incorrect calculation caused by messages received too late, a component must perform a rollback and cancel some messages already sent, possibly initiating further rollbacks in other components. Nevertheless the distributed TimeWarp algorithm returns a correct result.

In this paper this technique is modelled with the development methodology Focus and the correctness is formally investigated. Starting from a simple, centralized simulator three development steps are performed, reaching a distributed simulator using TimeWarp. The simulators on various abstraction levels are formally specified, and the development steps are verified using the techniques of Focus.

[*]This paper originates from a cooperation of the projects A6 and B4 of the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Recherarchitekturen", which is sponsored by the DFG.

Contents

List of Figures

List of Tables

1 Introduction

Simulations are used in several areas as e.g engineering, computer science and economic applications. Since simulation of complex systems is in general a task costing a lot of calculation effort, parallelization of the calculation is a promising way to reach a substantial speedup. If the simulation is carried out by several computers simulating different parts, these parts are not totally independent from each other and therefore the computers have to communicate and exchange information. So a kind of loose synchronization is necessary to assure that all information is available at the right place when it is needed there.

In this paper only simulators for a specific type of models are considered: The models are described adequately through *state transitions* at *discrete points in time* triggered by *events* that contain information *what* change of states happens *when*. A non-conservative, optimistic and efficient synchronization mechanism which can be used for that kind of model is the TimeWarp mechanism that is tackled formally in this paper. The formal treatment of the TimeWarp mainly aims at two different aspects of correctness. One is the inspection of the correctness of the algorithm calculating the value for the *global virtual time* (*gvt*). This *gvt* contains information about the global progress of the simulation. The other aspect of correctness deals with the distribution over several simulators, the communication between them, and the handling of stragglers, cancellations, and the rollbacks that can occur when errors have to be corrected. This aspect plays a crucial part concerning the behaviour of the components of the simulators, and is essential to make the distributed simulator work. This paper is focusing on the second aspect that is not yet formally investigated. The first one is already treated formally in literature ([Gaf85], [KRF$^+$96]) and necessary prerequisites for the correctness of the *gvt*-algorithm are simply assumed in this paper.

The object of this paper is to present a formal development of a distributed, correct simulator for event-driven models that uses TimeWarp for the synchronization between the components. The meaning of "correct" can only be stated relative to another formal description of a simulator, which may be very abstract and is not required to be efficient, but defines the functionality of a simulator in a way that is near to intuition, easy to understand and therefore suitable for validation. This is done in a first step. Combining three intermediate steps altogether four versions of simulators are specified, and each one is proved to have essentially the same functionality with only slight differences concerning the behaviour in time.

As formal foundation the methodology Focus ([BS97]) is used. It offers mathematical, well-founded methods and techniques for specifying distributed systems on various abstraction layers and for relating the different levels through a provided notion for *refinement*. Since Focus is mainly concentrating on reflecting the communication between components, it is well suited for the purpose of this paper.

Two main motivations for this work can be mentioned. One is the demand of users of simulators implementing TimeWarp for being sure that the results are correct. One cooperation partner who implemented the TimeWarp for simulating digital circuits expressed this desire. Another motivation is the chance to demonstrate and test the abilities of Focus (and formal methods in general) in larger case studies and to get new impulse for further improvements.

The paper is structured in a way that the formal specification of TimeWarp is getting more and more detailed so that the reader can follow to that detail level he is interested in. The development is shown starting from an informal description of TimeWarp through the formal specification of simulators at different abstraction levels up to the formal verification with the proofs in the appendix. This paper is intended to be readable stand-alone; all concepts of Focus that are needed for understanding are briefly explained.

Section 2 of this paper contains an informal description of the TimeWarp algorithm. In section 3 a short introduction to the concepts of Focus is provided. An outline of the stepwise development of the algorithm together with some aspects of how the algorithm is modelled is given in section 4. The formal specifications of all four abstraction layers are described in section 5. In section 6 the relation between the different abstraction levels is investigated, and the idea of the formal proofs is outlined. A summary of the paper together with some conclusions is contained in section 7. A summary of all definitions and the proofs are found in the appendix.

This work originates from a cooperation of the projects A6 and B4 of the "Sonderforschungsbereich 342" of the Technical University of Munich, having the aim to develop tools and methods for the efficient use of parallel and distributed computer architectures. With this cooperation project, A6 was able to test the suitability and benefit of the use of the developed formal methodology (namely Focus) in this case study supplied by B4, whose main focus lays in parallel simulation of digital circuits and communication networks. The results of this work show the profit that can be gained from a cooperation of partners from theory- and application-oriented projects.

2 Simulating with TimeWarp

In this section the basic concepts of the simulation of event-driven and discrete simulation models are described, valid for both distributed and non-distributed simulators. Then the idea of the TimeWarp mechanism is explained, offering an efficient method for the synchronization in a distributed simulator.

If something of the real world is to be simulated, this world has to be described by an adequate *model* that is an appropriate abstraction. TimeWarp cannot be used for all kind of models but only for those that have certain properties stated now. A lot of examples for models that fulfill these requirements exist, e.g. the simulation of digital circuits and traffic flow simulations.

- The *time* in such a model must be adequately described by discrete points of time (comparable with natural numbers) in contrast to a continuous time (real numbers).

According to the *virtual time paradigm* ([Jef85]) there is a clear distinction between the virtual time of the simulation model and the physical time of the simulator itself.

- At each single moment of the (physical) time the model is in a specific *state*, containing a description of the state of all constituents of the model, e.g. the (virtual) time to be simulated next in the simulator.
- The changes between one state and the next can be described by a set of *events*, containing information about *what* change in the model is happening *when* (concerning virtual time).

The way how a non-distributed simulator for a model like this is working is quite simple: It is started with an initial state of this model, and an initial set of triggering events. These events contain a description (d) of *what* is changing in the state of the model and *when* this change has to happen ($t_{execute}$ or t_e). When the time vt is simulated, all events with $vt = t_{execute}$ have to be considered, causing a change in the state of the model and creating resulting events. A simple simulator starts with simulating time 0, calculates all events that derive from this simulation step and inserts them in the event set, proceeds with simulating time 1, then 2, and so on. The creation-time ($t_{generate}$ or t_g) of events is also noticed. Due to causality, it can be assumed that $t_e \geq t_g + 1$, stating that the consequences of the simulation of a certain time only concern the future. Events can be represented (together with the symbol "+" for reasons explained later) by a quadruple of the form

$$[+, t_{generate}, t_{execute}, d]$$

To achieve a faster simulation of complex systems, it is promising to use parallel computation in a distributed system of interacting components. For that, the model to be simulated must be split into several partitions, one for every available simulating component. Every component should be able to simulate as independently as possible, using a *local virtual time* (lvt) describing the time which is to be simulated next in its partition. Since some events are created in one partition but have to be executed in another, events must be exchanged between the components of the simulator. To preserve causality it must be assured that a component simulating the time lvt must have *all* events available that have an execution time equal to lvt.

A distributed simulator with *conservative* synchronization uses waiting to avoid causality errors, i.e. a component does wait until it is sure that all required events have been received from other components. The behaviour of a simulator using TimeWarp is different: instead of wasting time by waiting, the components assume optimistically that no further events will come and just proceed in computing. Due to that behaviour, there is a need to care about *stragglers*. These are messages that arrive too late, i.e. with an execution time t_e smaller than the actual lvt. Since such an event was not yet available when simulating the time t_e, the receipt of this straggler signals that probably the computation steps from simulating t_e until now were wrong. This error has to be corrected by a *rollback*, undoing all activities of the local simulator back to the situation before t_e was simulated.

To make the components capable to do rollbacks, additional mechanisms and data structures must be provided. The components of the simulator must be able to react on stragglers. To do a rollback, older states of the model must be saved, and the components must be able to cancel messages that have been already sent during a computation that turned out to be possibly wrong. Cancelling is done by re-sending the message that has got invalid, but marked with a negative sign. Two messages of the form $[+, t_g, t_e, d]$ and $[-, t_g, t_e, d]$ are called *antimessages* and annihilate each other when they "meet". There are two different cancellation strategies: Using *aggressive* cancellation all messages already sent are cancelled at once when doing a rollback. *Lazy* cancellation delays sending antimessages until it is sure that sent messages are indeed wrong. Sometimes stragglers have an impact only to some sent messages, so that some cancellations can be avoided. This last strategy needs more effort for the implementation, but leads to simulations that can take a short-cut of the critical path (given by the causality graph of all events).

As now also cancellations are sent between the components, it must be defined how they react on these messages. Fortunately, this mechanism is quite simple. If a cancellation was received for an event whose execution time lies in the future (relative to *lvt*), then this event is just deleted and no further action is needed. If there occurs a cancellation of an event that was already processed, an ordinary rollback must be performed, and the computation is done again without this event.

The rollbacks can lead to a kind of snowball effect. If one component sends cancelling messages causing rollbacks in other components, these could cause further rollbacks, and so on. It will not happen that all simulators rollback to the beginning (with $lvt = 0$) and start all over, what can be concluded from inspecting the algorithm for the *global virtual time*, that is an essential part of the TimeWarp.

The *global virtual time* (*gvt*) contains the information about the general progress of the distributed simulator. This value is a kind of minimum of all *lvt* values together with the execution time of messages still on the way between components. So if this value is known, it can be assured that there are no longer any events in the system that will cause a rollback to a time before this *gvt*. An approximation of this value can be calculated by a central instance when it is supplied with enough information from the components of the simulator. This algorithm together with its properties is described in literature as e.g. [Bau94] and [JS85], and was proved formally to be correct in [Gaf85] and [KRF+96].

The effects of the rollbacks with their cancellation messages should be made invisible from the outside of the simulator. So the results returned to the user should be free from these messages. To achieve this, the simulators firstly keep back the results that are meant for the environment. When a new value for *gvt* is calculated by the central instance, it is broadcasted to all components. As reaction, those send all results that are now safe from being cancelled to the central instance, and they clean up their store of old states to which no rollback will ever occur again.

For more detailed descriptions of the TimeWarp see e.g. [JS85], [Jef85], [Fuj90] and [Bau94]. In [Bau94] it is implemented for simulating digital circuits with a measurable speedup.

The above description should give an idea of the functionality of the TimeWarp mechanism. Since the interaction between the components is quite complex, it is not obvious that the returned results of such a simulator are correct. Thus, a formal specification and verification of the TimeWarp as done in the next sections can lead to interesting insights.

3 The Methodology Focus

Focus is a powerful methodology for the development of distributed reactive systems. It offers methods with a formal foundation for specifying and refining systems through several abstraction layers in a top-down manner. Since Focus contains a variety of techniques, specification formalisms and semantic choices, it is not possible to give an extensive introduction in this paper. The interested reader is referred to literature as [BS97] and [BDD+93] for an introduction, to [BBSS97] for an overview on case studies done with Focus, and [HSSS96] for the description of the supporting tool AutoFocus. In this section only a very short and specific introduction to some aspects relevant for this case study is presented.

According to the concept of Focus a distributed system consists of a number of components that can be partially connected with each other or with the environment via asynchronous one-way communication channels, comparable with unbounded FIFO-buffers. By defining the behaviours of the components and the topology of the connecting network of channels the system is sufficiently defined. The behaviour of this system can be deducted from the behaviour of its constituents.

To describe the topology of a distributed system, a graphical notation is sufficient. The components are depicted as named boxes, and the channels as named arrows pointing from components that are allowed to write messages on that channel to components that read these messages. Arrows coming from or pointing to the outside symbolize connections with the environment.

The basic data structure needed for the definition of components are *timed streams*, i.e. infinite sequences of messages including the special message $\surd$ (say *tick*) denoting that one time intervall had passed. In the so-called *synchronous model* used here, a global and discrete time is assumed, and in every time intervall at most one message can be transported between two components. The situation that no message has been sent during an intervall is denoted by $\surd$[1]. With these streams the whole communication history is modelled: a specific stream that is associated with a channel between two components contains all information *what* message is sent *when* between these components. Other

[1]This definition is slightly different from the usual definition, but this can be ignored for the purpose of this paper.

spec Voter: $(I^{\underline{\infty}})^N \rightarrow O^{\underline{\infty}}$

data $m \ : \ Integer \ = \ 0$

I_j	PRE	O	POST
$\forall j \in N : \surd$		$\surd$	$m' = m$
$\forall j \in N : i_j$	$\forall k, l \in N$ $i_k = i_l \neq \surd$	ok	$m' = m + 1$
$\forall j \in N : i_j$	$\exists k, l \in N$ $i_k \neq i_l$	fail	$m' = m - 1$

Table 1: A specification by a table

semantic variants common in Focus ignore aspects of time totally (*untimed model*) or admit a finite sequence of messages (instead of just one message) to be sent during one time intervall (general *timed model*).

To define a component, first the *interface* must be declared. This contains a description of its input and output channels as well as the type of messages that can be received or sent via these channels. The *behaviour* of a component can be described precisely by defining a relation between its input streams and its output streams, containing the set of communication histories that are valid for this component. One way to describe this relation is to define a stream-processing function that maps input streams to output streams. This function reads an input stream message by message, and writes - as reaction - some output messages onto the output channels. Stream-processing functions have to fulfill further semantic properties as continuity, realizability, time-guardedness and more, as explained in literature. It is possible to use state parameters to store control states or additional data that can be helpful for easier modelling.

One way to specify a stream-processing function is using a tabular notion as explained now with the example in Table 1. This component represents a kind of voter that compares all incoming messages and yields "ok" if they are the same and "fail" otherwise, and calculates internally the difference between the number of occurences of these two messages. The interface of the component called *Voter* is given in the first line. It reads input from N (with N as an arbitrary natural number) input channels containing messages of a type I (not necessarily specified more detailed at this abstraction layer), and writes to an output channel with type $O = \{\text{ok, fail}\}$. The index $\underline{\infty}$ denotes an infinite timed stream with messages of the appropriate type, extended by the additional symbol $\surd$. *Voter* has a state variable m of type $Integer$ that is initialized by 0.

The table itself contains one column for every input channel, one (optional) column for a precondition, one column for every output channel and one (optional) column for the postcondition. The number of the input channels I_j is parameterized here. Nevertheless, the according columns for the input channels can be represented by one column, superscribed by an indexed name as shown in the example. The distinction between different behaviours for a given input is now defined by the entries in the columns for the input and the precondition in a way that resembles pattern-matching in functional programming languages. The first line in the example describes the case that on all input lines no message (i.e. a $\surd$) is received. The second and third line instantiate the arriving messages with the variables i_j. The second line describes the case that all read messages are the same, while in the last case at least two messages are different. In all cases from all input channels exactly one message (maybe $\surd$) is read and "removed" from the channel. Note that it would be allowed to use the data state m in defining the precondition so that the behaviour could depend on the actual value of m. In this example exactly one of the three cases will be true. If more or none of the cases could occur at the same time, the behaviour is *underspecified.*

The output is now defined for all cases by the according columns. Changes of the state can be described by a predicate in the postcondition, using the convention that variables without primes denote the original values and variables with primes the new ones. So in the first case of this example, no input results in no output (denoted by $\surd$), in the other cases an "ok" or "fail" message is written to O, and the value of m is increased resp. decreased[2].

The semantic (i.e. the relation between the input and output streams) of such a table can be derived in a uniform way suitable for automatic treatment. Two variants of notions are given here. The first one defines the behaviour of *Voter* by a function $Voter$ calculating an output stream for an arbitrary input stream tuple i. It is defined by

$$\begin{array}{l} Voter : (I^{\underline{\infty}})^N \rightarrow O^{\underline{\infty}} \\ Voter(i) = \surd \ \& \ f[0](i) \end{array}$$

Since due to semantic reasons all streams have to start with a $\surd$, this $\surd$ is appended by the operator & in front of the rest of the output that is calculated by the state-based auxiliary function f, whose state is initialized by 0, as specified in the table. The definition of f derives from a simple translation of the lines of the table into conditional equations:

$$\begin{array}{lcrcl} true & \Rightarrow & f[m]((\surd,\ldots,\surd)\&i) & = & \surd \quad \& \quad f[m](i) \wedge \\ i_1 = i_2 = \ldots = i_N \neq \surd & \Rightarrow & f[m]((i_1,\ldots,i_n)\&i) & = & ok \quad \& \quad f[m+1](i) \wedge \\ \exists k,l \in N : i_k \neq i_l & \Rightarrow & f[m]((i_1,\ldots,i_n)\&i) & = & fail \quad \& \quad f[m-1](i) \end{array}$$

In the proofs of this paper the following, equivalent way to formulate the semantic turned out to be suitable. This notion uses a relation with the name VOTER and the dot-notation

[2]Note that the variable m is used neither in the precondition nor for the output, and could therefore be ommited without changing the behaviour. So this specification does not make too much sense, but is still suitable to show the expressiveness of tables.

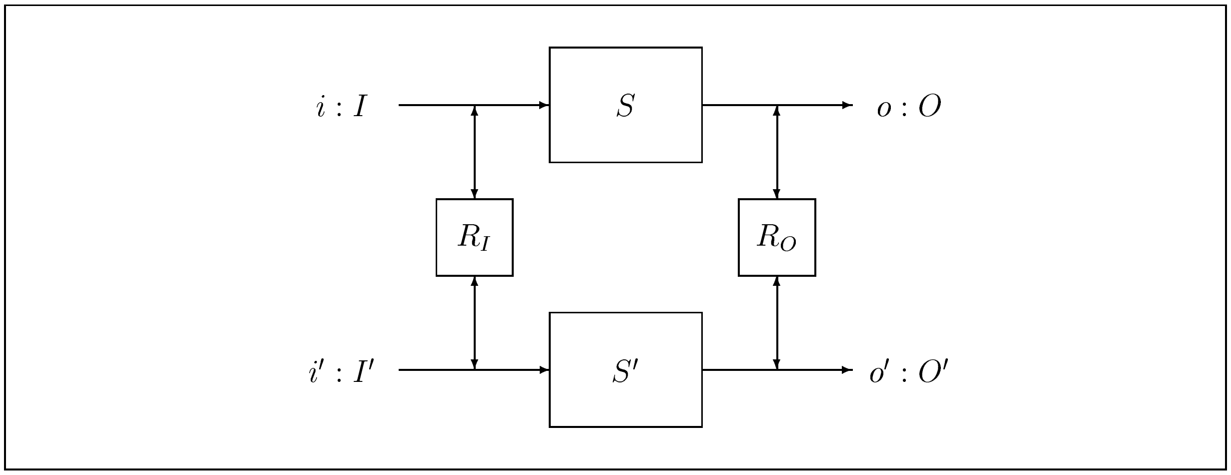

Figure 1: Interaction-Refinement

by which single elements of streams can be addressed and the value of data states Z can be described at a specific time t by $Z.t$. The relation states that streams i_j $(j \in N)$ and o are related by VOTER, if all streams start with a $\surd$ and if messages in the input stream at time t are related to messages in the output stream at time $t + 1$ [3] according to the table:

$$
\begin{array}{l}
\text{VOTER}[i_1, \ldots, i_n; o] \Leftrightarrow \\
\quad \forall k \in N : i_k.0 = \surd \ \wedge \ o.0 = \surd \ \wedge \\
\quad \forall t \in TIME, m : INTEGER. \\
\quad \begin{array}{llll}
 & Z.t = m \ \wedge \ \forall k \in N : i_k.t = \surd & \Rightarrow \ Z.(t+1) = m & \wedge \ o.(t+1) = \surd \\
\wedge & Z.t = m \ \wedge \ \forall k, l \in N : i_k.t = i_l.t \neq \surd & \Rightarrow \ Z.(t+1) = m + 1 & \wedge \ o.(t+1) = ok \\
\wedge & Z.t = m \ \wedge \ \exists k, l \in N : i_k.t \neq i_l.t & \Rightarrow \ Z.(t+1) = m - 1 & \wedge \ o.(t+1) = fail
\end{array}
\end{array}
$$

To support the stepwise top-down development of a system, a notion of *refinement* is supplied with Focus. A valid refinement relation between two components states that these two are doing essentially the same, but on different abstraction levels. The connection between the different layers stating this similarity has to be formulated by so-called *InterAction-Refinement-relations* (IAR-relations). The situation is demonstrated by Figure 1. The abstract component S should have a similar functionality as the more concrete component S', when the input and output streams are related through IAR-relations R_I resp. R_O. A behaviour of the component S', denoted by a pair of streams i' and o', must be an "implementation" of a behaviour that is specified by the abstract component S, meaning that S' must show essentially the same behaviour as S, but on a more concrete level. More formally stated there must be abstract streams i and o that denote a valid behaviour of S and whose representations are i' and o'. This[4] is formalized by (with $R[i; o]$

[3]The choice of the *strongly time guarded* model is reflected here.

[4]This form of interaction refinement is called *upward simulation*. Other notions of simulation can be found in [Bro93]

denoting that i and o are related by the relation R):

$$\forall i', o' : S'[i'; o'] \;\;\Rightarrow\;\; \exists i, o : R_I[i; i'] \;\wedge\; R_O[o; o'] \;\wedge\; S[i; o]$$

An important aspect for the top-down-development is *compositionality.* It assures that components can be refined independently. To gain compositionality the two IAR-relations must fulfill the *IAR property.* This property is assured if a concrete stream has exactly one corresponding stream on the abstract level (i.e. $R[x; z] \wedge R[y; z] \Rightarrow x = y$). Weaker conditions for IAR relations can be found in literature mentioned below.

If all these necessary conditions are fulfilled, the refinement (stating that S and S' are doing the same with respect to R_I and R_O) is written as

$$S \overset{(R_I;R_O)}{\rightsquigarrow} S'$$

The interaction-refinement has some specialized variants, namely *behaviour* refinement (restricting underspecification), *structural* refinement (splitting a component into a network of interacting subcomponents) and *interface* refinement (changing the interface). These are further investigated in [Bro92] and [Bro93], and summarized in [BS97], together with all formal definitions and further properties of '$\rightsquigarrow$' omitted in this short introduction.

4 Modelling TimeWarp with Focus

This section outlines the stepwise development of the distributed, TimeWarp-synchronized simulator through four steps.

When modelling with Focus, a model for time must be chosen. For the purpose of this paper, the *synchronous* time model of Focus is used, for the following reasons: To be able to model that messages are *not* arriving at a certain time, it is inevitable to use timed streams. In the *synchronous time model* there is at most one message at a time, or none (symbolized by $\surd$). This time-model turned out to be suitable for the treatment of the TimeWarp as done in this paper, as for example the proofs turn out to be easier formulated by knowing that messages that are sent at the same time occur at the same positions in the streams. In addition, all stream processing functions are meant to be strongly time-guarded, i.e. the reaction of all components is not instaneous, but needs at least one tick. This choice avoids problems that can occur in feedback-loops, and has no other drawbacks.

When arguing about the correctness of the TimeWarp-based, distributed simulator it is obvious that there is the need for a clear definition of what *correct* means. Correctness can be defined only in relation to a first formal specification of an event-driven simulator. This should be kept as easy as possible and will be modelled in this case by one single and abstract component that calculates the results of a simulation in a simple and

classification of simulator	centralized	distributed
event-driven	CED	DED
single-step	CSS	DSS

Table 2: The four abstraction layers

straightforward way. Then more and more complex variants of the simulator are given, whose correctness can then be described relatively to the preceding versions in a formal way. During development two major steps have to be made:

- Distribution from a centralized component to several communicating components. Note that some (possibly simple and unefficient) synchronization is needed to ensure correct results.
- Optimizing the synchronization through the TimeWarp mechanism.

To reduce the complexity of the system development, it is useful to keep these two major steps separated in different development steps. For that reason, we propose four different abstraction layers, identified as CED, CSS, DSS and DED, whose properties are summarized in Table 2. CED represents the most abstract, DED the most concrete layer investigated in this paper.

On all levels a slight simplification is now introduced: the initial phase is left out, i.e. all concerned components are considered to know the information about the simulation model with all initial events. The way how the components are getting this information is not modelled. This simplification reduces some technical overhead, and is not essential to the problem to be tackled in this paper.

The most abstract simulator, **CED**, is quite simple, and represents the basis for all further considerations. Since all information is available in this component from the beginning, no input channels are necessary. On the output channel all events that result from the

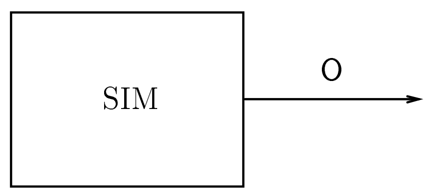

Figure 2: Centralized Simulation (CED, CSS)

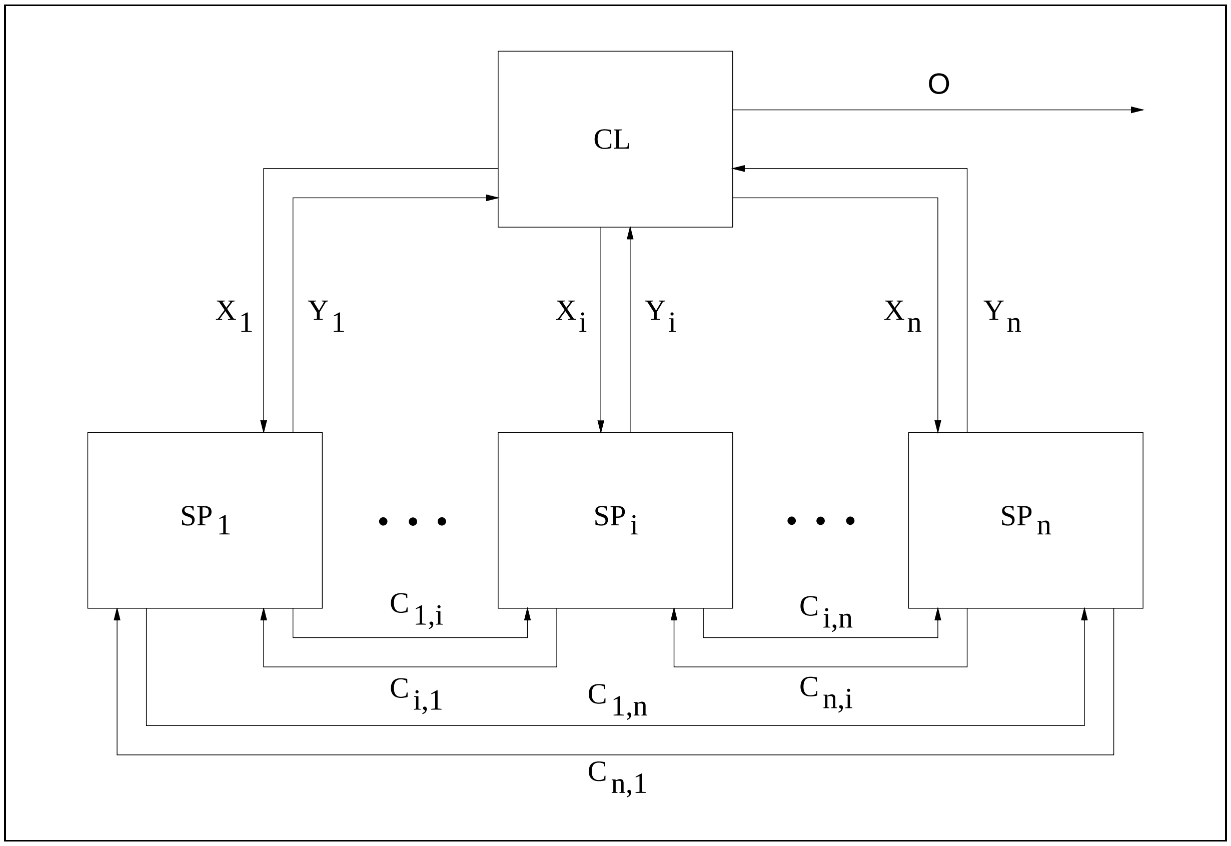

Figure 3: Distributed Simulation (DSS, DED)

simulation are sent. Since this simulation is event-driven, only the points of time are considered for which events are existent. During the time between two succeeding events "nothing is happening" in the model, meaning it is sufficient that the local virtual time jumps in steps. A graphical representation of this component is given in Figure 2.

The next simulator, **CSS** plays an intermediate role just to make the verification step between CED and DSS easier by splitting it up into two smaller steps. The only difference is that *all* points of time are considered, even those without corresponding events that describe something to happen at that time. Thus, this version of the simulator makes the same simulations steps as CED, but additionally "empty" steps in-between. The graphical representation (Figure 2) is the same as the one of CED.

The simulator called **DSS** is a distributed simulator, meaning that the actual simulation is performed by several, similar components SP_i that simulate disjoint parts of the model. For a correct simulation preserving causality, it is necessary that all relevant events are available in a component when this component is performing a simulation step. In DSS a very easy synchronization mechanism is realized by making all components stepping through all points of time synchronously. All components make a calculation step for a specific time, then new events are exchanged via channels C_{ij} (connecting SP_i with SP_j)

and all components continue with the calculation step for the next point of time. Since this stepwise proceeding through time was chosen already in CSS, the output of DSS is the same as the one of CSS. In Figure 3 a further central component CL can be recognized. This component collects the events of all simulating components and produces the overall output by just merging these events. The interface of the system to the environment keeps therefore the same, i.e. only one output channel. Note that the channels named X_j (for all j) are not really used in this simulator, i.e. no messages at all are sent over these channels. They are already existent in this simulator to make the refinement to the next, more concrete simulator easier. Alternatively it would be possible to insert a further development step in which these channels are introduced by an interface refinement step.

The timing model of the last two simulators resembles the so-called *cycle-based simulators* if they would be extended by a specific management of the events, allowing a delay of the passing of messages different from one.

The TimeWarp mechanism itself is realized in **DED**, that can be depicted also by Figure 3. The difference to DSS is the synchronization mechanism, i.e. the way the components proceed with their local virtual time and talk to each other. The components now simulate different points of time, and have to use the more complex protocol including stragglers and cancelling messages. In this simulator, the central component CL has not only to collect and merge the results, but it has also to implement the *gvt*-algorithm. It can do this by getting information about the local virtual times of the other components and sending the new *gvt* to the components if a new value has been calculated.

These four abstraction layers are specified formally in section 5, and the validity of the corresponding refinement relations is shown in section 6. The output of the different simulators is not precisely the same, since there are slight differences concerning the behaviour in time. But when the outputs are viewed with an appropriate abstraction concerning time, the results are identical.

During the modelling of TimeWarp as done in this paper, some simplifications were made to keep the specifications simpler. They are summarized in the following.

- The termination of the simulation is not taken into account. The simulators just compute without terminating, sending empty messages once the event set gets empty.
- The input of the initial events is not modelled as already mentioned. It is assumed that all components already have their start-up information.
- Lazy cancellation is not used, since the mechanism for rollbacks would be even more complicated.
- The storage of the components is not limited, so all states could be stored and no optimization is necessary.
- The algorithm for calculating *gvt* is assumed to be correct.

- The effort for one simulation step is assumed to be constant (i.e. one tick) for all components. If different costs or calculations speeds for the different components should be modelled, this could be done by allowing a component to delay its output by sending a (finite) sequence of ticks first.
- The medium for communication (i.e. a physical network) is not modelled, but it is assumed that all components are connected directly with each other. The communication is assumed to be totally free from any faults, what can be achieved in reality by using appropriate communication protocols.

Despite these simplifications the model is still an adequate abstraction of TimeWarp for investigating the essential concepts.

5 The Specifications

In this section the four views for the different abstraction layers already introduced in section 2 are formally specified. The necessary sets and functions are introduced step by step. All definitions are summarized in appendix A. The refinement relations between the different simulators are investigated in the next section 6.

5.1 Centralized, Event-Driven Simulation (CED)

This first view provides a specification of a centralized simulator. Since this is the first formalization, this specification describes the behaviour as simple and abstract as possible. A lot of the details is hidden in the auxiliary functions. The simulator called CED is specified by Table 3.

CED does not receive any input, and delivers a stream of the type $O = EVENTS^{\underline{\infty}}$, an infinite, timed stream containing sets of events as messages. Thus, CED has the type $\{\} \to O$. Note that at this abstract level there is no further knowledge necessary about the structure of the elements of $EVENTS$. CED contains a triple as internal data state:

- In the variable s the current state of the simulation model is stored, e.g. the state of all (technical) internals of the circuit whose run is to be simulated. The value $START$ contains the initial state of the simulation model.
- The variable ev stores the current set of events, containing events still to be executed together with events already executed. The triggering events that are assumed to be already available at the beginning of the simulation are contained in the constant EV.
- In vt the next time that is to be simulated is stored. Through the appropriate call of the function nxt it is initialized with the first time for which a simulation step has to be executed.

spec CED : $\{\} \rightarrow O$

data	s	:	$STATES$	$=$	$START$
	ev	:	$EVENTS$	$=$	EV
	vt	:	$TIME^{\infty}$	$=$	$nxt(EV, 0)$

O	POST
$\Pi_0(sim(s, ev, vt))$	$ev' = ev \cup sim(s, ev, vt)$
	$s' = next(s, ev, vt)$
	$vt' = nxt(ev', vt)$

Table 3: Specification of simulator CED

The function *nxt* and other functions only mentioned shortly in this chapter are specified in detail in appendix A.

The behaviour is specified by a simple, slightly degenerated table. Since there is no input to this component at all, this component outputs a stream of events autonomously. Consider CED to be in the data state described by the values of (s, ev, vt). Then for every step the function call $sim(s, ev, vt)$ is performed. This yields all events that arise from simulating the model described by s for the time vt considering all events out of ev with timestamps equal to vt. The resulting events are all stored in the set ev' (the new ev), and some of them, namely those that are relevant for the overall result, are selected by the function Π_0 and sent as output to O. For the case that there are no resulting events, $\Pi_0(\emptyset)$ delivers an empty output message, i.e. a tick $\surd$. The following state of the simulation is calculated by the function *next*. The next point of time that has to be simulated can be concluded from the new event set ev'. The search for the next relevant event after vt is made by the appropriate function call of *nxt*.

5.2 Centralized, Single-Step Simulation (CSS)

The Simulator CSS differs from Simulator CED only in its stepwise proceeding of the local virtual time. It is specified in Table 4.

This simulator performs calculations for all points of time, even when there are no events existent for that time. So vt is initialized by 0, and incremented by 1 at every step. This has the effect that many function calls of *sim* will deliver the empty set as result and *next* will yield the same s as it was supplied with if there are no events that describe any change in the model at the respective point of time.

spec CSS: $\{\} \to O$					
data	s	:	$STATES$	=	$START$
	ev	:	$EVENTS$	=	EV
	vt	:	$TIME^{\infty}$	=	0

O	POST
$\Pi_0(sim(s, ev, vt))$	$ev' = ev \cup sim(s, ev, vt)$ $s' = next(s, ev, vt)$ $vt' = vt + 1$

Table 4: Specification of simulator CSS

5.3 Distributed, Single-Step Simulation (DSS)

Simulator DSS is a distributed simulator. It is assumed that the model to be simulated is split into an arbitrary number of n partitions. DSS is specified to consist of one central component CLSS and n similar components $\mathrm{SP}_1\mathrm{SS}$ to $\mathrm{SP}_n\mathrm{SS}$. Each of them performs the simulation of one partition. The two kinds of components are specified formally in this section.

The controller CLSS is specified by Table 5. This table is no longer a degenerated one, since the behaviour is dependent from the input. CLSS collects all the events that are received via the channels Y_j (for all $j \in N$)[5] and outputs the union of these sets directly (i.e. one tick later, due to the selected time-model of Focus) to the channel O. The expression $m_N =_{df} \bigcup_{j \in N} m_j$ is defined as abbreviation. The union-operator for sets has to be extended to be defined over $\surd$ to make it formally correct, so $A \cup \surd = \surd \cup A = A$ has to be valid for all sets A of events. Since there is no storage of any information necessary, this component does not have any data state. As already mentioned, the channels X_j are not needed, so no messages are sent on these channels, denoted by $\surd$.

The components $\mathrm{SP}_i\mathrm{SS}$ (for all $i \in N$) work quite similarly to the simulator CSS. The main difference is that every component takes over the simulation of only a part of the model. The communication is more complex: In order to achieve correct results, every simulator has to exchange events that are not interesting for the overall result but needed in other simulators. For this the channels C_{ij} are used, connecting $\mathrm{SP}_i\mathrm{SS}$ with $\mathrm{SP}_j\mathrm{SS}$. Since the channels C_{ii} (connecting components with itself) prove to be useful in the next level of abstraction, they are not excluded. The events that have to be sent to the environment are put on the channel Y_i connected to CLSS.

Two additional functions are required to be able to specify in an abstract way that $\mathrm{SP}_i\mathrm{SS}$ simulates only a certain partition of the model:

[5]Note that the indices i and j are meant to range over $N = \{1, \ldots, n\}$ for the rest of this paper.

spec CLSS: $Y^n \rightarrow O \times X^n$		
Y_j	O	X_j
$\forall j \in N$: m_j	m_N	$\surd$

Table 5: Specification of controller CLSS

- The function $part_i$, applied to a description s of a simulation model, delivers all information of partition i of the model that is necessary to simulate this partition.
- The function Π_i selects the subset of a given set of events that have to be considered when simulating partition i.

On this abstract level, there is no need to define these functions in detail. A loose algebraic specification of some properties is sufficient, as for example the following requirement:

$$part_i(next(s, ev, vt)) = next(part_i(s), \Pi_i(ev), vt)$$

It denotes the property that if a simulation step is made for the timestamp vt with only the information about partition i of s and about the events that have to be considered in partition i (right-hand side of the formula), then the same state is reached in partition i of the model when the whole model is simulated, and then only partition i is inspected (left-hand side of the formula). Requirements like this were found during the process of proving the refinement relations, and are summarized in appendix A. When these functions are implemented in future stages of the development, they just have to fulfill these requirements, and none further.

In the specification of SP$_i$SS in Table 6 the variables s and ev are initialized in a way that SP$_i$SS will simulate partition i. At every step, this component reads all event-sets on the channels C_{ij} from its "colleagues". When the simulating functions (i.e. sim and $next$) are called, all these received events together with the events stored locally in ev are considered. By the function Π_0 and Π_j the resulting events are distributed to the colleagues and to the central controller, so that all components get the information they need.

Some of the calculated events are needed in the same partition where they have been generated. Instead of putting them in ev directly, the component SP$_i$SS sends these messages to itself over the channel C_{ii}, and therefore ev' results from a union with the value ev_N and not with the results from the call of function sim. This makes the specification and the proofs simpler, since the quantification covers the index set $1 \ldots n$ in a uniform way. In case of "real" messages being sent on one of the channels X_i (i.e. a message different from

spec SP$_i$SS: $X \times C^n \rightarrow Y \times C^n$

data				
s	:	$STATES$	$=$	$part_i(START)$
ev	:	$EVENTS$	$=$	$\Pi_i(EV)$
vt	:	$TIME$	$=$	0

X_i	C_{ji}	Y_i	C_{ij}	POST
$\surd$	$\forall j \in N$: ev_j	$\Pi_0(sim(s, ev \cup ev_N, vt))$	$\forall j \in N$: $\Pi_j(sim(s, ev \cup ev_N, vt))$	$ev' = ev \cup ev_N$ $s' = next(s, ev \cup ev_N, vt)$ $vt' = vt + 1$

Table 6: Specification of simulator component SP$_i$SS

$\surd$), the behaviour of SP$_i$SS is unspecified. But since CLSS is obviously never sending a message except $\surd$, this case will never arise.

Simulator DSS itself can now be defined by the compositon of all participating $n+1$ components according to Figure 3 by

$$\text{DSS} = \text{CLSS} \otimes \text{SP}_1\text{SS} \otimes \ldots \otimes \text{SP}_n\text{SS}$$

The operator $\otimes$, suitable for combining components to networks, is used here in a very unprecise way. A more formal definition is omitted here, since the graphical notation is much more intuitive. More formal arguments are used in appendix B.

5.4 Distributed, Event-Driven Simulation (DED)

The simulator DED, at last, realizes the TimeWarp mechanism that is already described in previous chapters. There are again two kinds of components, one central controlling instance and N simulating components, each dealing with a specific partition of the model. The functionality of these components is more complex compared to the previous DSS, and specified in this section.

The component CLED, specified in Table 7, still has to collect events from the simulating components, but has in addition the task to implement the *gvt*-algorithm. Since this algorithm and its correctness are already investigated in literature, the algorithm is modelled here in a very abstract way, and its correctness is postulated by properties about functions defined now.

CLED contains one very abstract data element called z of the unspecified set Z. It is simply assumed that in this variable all necessary information needed for the algorithm can be coded. A more detailed description of the contents of this variable can be reached by data refinement in future development steps. The initialization of z is done by using a specific element *init*. When messages are received on the channels Y_j, all received events are sent directly to O. Since these messages m_j can also contain information about the status

spec CLED : $Y^n \to O \times X^n$			
data $z : Z = init$			
Y_j	O	X_j	POST
$\forall j \in N :$ m_j	m_N	$\forall j.$ **if** $triggered(z')$ **then** $gvt(z')$ **else** $\surd$	$z' = update(z, m_N)$

Table 7: Specification of controller CLED

of the simulators (called $info$), the union-operator has to be generalized again to ignore $info$ messages when sending the set union m_N. In this abstract specification, there is only one unique value $info$, which of course has to be refined to more specific data through data-refinment in future development steps. The $info$-messages themselves are used to calculate the new state z' using the function *update*. If a new value for gvt is available, this is indicated by the boolean function *triggered*. Then all simulating components are informed by sending to them the appropiate value, calculated by the function gvt. If there is no new value for gvt, no messages are sent to the components SP$_i$ED. The correctness of the gvt-algorithm implemented by the functions *update*, *triggered* and gvt is easily postulated by affirming the liveness condition that there is indeed a real proceeding of the *global virtual time*.

The component SP$_i$ED specified in Table 8 differs from the preceding components in complexity. This component has to deal with stragglers and cancellations in the event sets that are exchanged between the components, and with gvt messages received from the controller. These new requirements make new, appropriate data elements necessary to store states for eventual rollbacks and for cancelling messages that have already been sent but turn out to be wrong.

The elements s and ev have the same task and initialization as in SP$_i$SS. The variable vt for the local time is initialized by a call of nxt with the first point of time for which an event with an appropriate execution time is contained in ev. In *outQ* all events resulting from the calls of *sim* are stored for two reasons: The events that have to be sent outside the simulator via CLED must be kept back in *outQ* until an appropriate gvt-message is received, assuring that no relevant rollback will happen anymore. Additionally *outQ* also contains the messages already sent to the colleagues, so that the component knows which messages are to be cancelled (i.e. sent again with a negative sign) when a rollback occurs. The variable *hist* is also needed to allow rollbacks: a list of the states s together with a timestamp (saying when this state was valid) are stored in this set. Again sets are used for specifying these two new elements at this abstract level. In later steps, the sets can be implemented through more efficient data structures.

spec SP_i ED: $X \times C^n \rightarrow Y \times C^n$

data	s	:	$STATES$	=	$part_i(START)$
	ev	:	$EVENTS$	=	$\Pi_i(EV)$
	vt	:	$TIME$	=	$nxt(ev, 0)$
	$outQ$	:	$EVENTS$	=	$\emptyset$
	$hist$	:	$HIST$	=	$\emptyset$

X_i	C_{ji}	PRE	Y_i	C_{ij}	POST
$\surd$	$\forall j :$ ev_j	$nxt(ev_N, 0) \geq vt$	$info$	$\forall j :$ $\Pi_j(sim(s, ev + ev_N, vt))$	$ev' = ev + ev_N$ $s' = next(s, ev + ev_N, vt)$ $vt' = nxt(ev', vt)$ $outQ' = outQ \cup sim(s, ev + ev_N, vt)$ $hist' = hist \cup \{(s, vt)\}$
$\surd$	$\forall j :$ ev_j	$nxt(ev_N, 0) < vt$	$info$	$\forall j :$ $\Pi_j(cancel(outQ, vt', vt))$	$ev' = ev + ev_N$ $s' = get(hist, vt')$ $vt' = nxt(ev_N, 0)$ $outQ' = outQ + cancel(outQ, vt', vt)$ $hist' = hist - \{(s, t) \| t \geq vt'\}$
gvt	$\forall j :$ ev_j		$\Pi_0(select(outQ, gvt))$	$\forall j.$ ev_N iff $j = i$ $\surd$ iff $j \neq i$	$outQ' = outQ - select(outQ, gvt)$ $hist' = hist - \{(s, t) \| t \leq gvt\}$

Table 8: Specification of simulator component SP_iED

The specification of $\mathrm{SP}_i\mathrm{ED}$ in Table 8 contains three lines, distinguishing three different kinds of messages on the input channels. The first two cases are discriminated by a precondition. These cases contain the specification of the behaviour if no gvt-message is received from the controller.

In the first case no rollback is triggered. This case can be recognized by the component by inspecting the received events in ev_N: If all of their execution times are not earlier than the point of time that is to be executed next (stored in vt), the component can perform a usual simulation step, which is similar to the steps CED is doing. The function sim is called, supplied with all information needed, and the resulting events are sent to the colleagues and stored in $outQ$ for further reference. The old state s together with vt is stored in $hist$. To the controller some information $info$ is sent, but no resulting events. Note that the operator $+$ is used (instead of $\cup$) when merging the new received events with the events already stored in ev. This operator returns a kind of set union, but dissolves messages with their antimessages. If, for example, there was an event already received that will have to be taken into account when simulating some future point of time, and now a cancellation is received, these two messages just dissolve, get invisible and have no more influence on future calculations. The function $+$ is specified in section A. The new vt' is set to the next time when an event has to be considered, so the simulation-time is jumping from one relevant point of time to the next.

The second line describes the behaviour when a rollback is initiated by an event with a time stamp referring to a point of time that was already executed. It is not important if this message is a positive message (event) or negative message (cancellation of an event). Since the simulation step of the past time was done without this event, this step together with all the following could be wrong and must be corrected by a rollback. The time to which vt has to be set back is described by vt'. The function $cancel(outQ, vt', vt)$ selects all events out of $outQ$ that have been generated between vt' and vt and signs them negative. They are sent to the colleagues (and itself) again, so that all sent messages (including internal events) are cancelled. Then $outQ$ can be reduced by all events that are now cancelled. Note that the operator $+$ together with the negative signs introduced by $cancel$ has exactly the desired effect. The state s which was valid when executing the time vt' is seeked in $hist$ by the function get, and is assigned to s', meaning the state is reset. Then all states in $hist$ that are no longer needed are deleted.

The third line deals with the case that a message was received containing a new value for the gvt. In this case it is assured that no events with an execution time before gvt will ever occur again. This means that all events generated before gvt can be sent to the central component that forwards them to the environment. The variables $outQ$ and $hist$ can then be cleaned up since information concerning the time before gvt is no longer needed. When the gvt-message is received, there are in general events arriving on the C_{ij}-channels in the same moment. These messages may not be lost. For that reason a very simple way to buffer those messages is chosen for this specification: The component sends all received events again to itself, so that it will receive them again with the next tick. The colleagues do not get any message in this case.

6 The Verification

In this section the refinements between the different abstraction levels are described in order to make it plausible for the reader unfamiliar with formal verification why the TimeWarp-simulator DED yields the correct result as defined by the simple simulator CED.

The relation between two levels of abstraction is described by an *Interaction-Refinement* (IAR). All of the following three subsections contain

- a specification of the necessary IAR-relation,
- a statement of the proof obligation, and
- a description of the idea of the proof.

The formal proofs are not given here, but can partially be found in appendix B.

6.1 Refinement of CED to CSS

The difference between CED and CSS is the different behaviour concerning time. While CED performs calculation steps only for the relevant points of time when events with an appropriate execution time are existent, CSS makes simulation steps for all $t \in TIME$, thus including steps with no production of output. So the output of CSS contains just more $\surd$-messages than the output of CED, and the output of both simulators is the same when a suitable abstraction concerning this difference is made.

This abstraction is modelled as IAR by defining a relation between the output streams of CED and CSS, as illustrated in Figure 4. Since these components do not have input channels, no IAR-relation must be defined for the left-hand side of the diagram.

The relation RT (for *ReTime*) between the output streams for both simulators states that they are the same when all $\surd$s are left out and just the order of messages is considered in the streams. The formalization is easy since the standard operator $\bar{.}$ can be used:

$$RT[o^1; o^2] \equiv \quad \overline{o^1} = \overline{o^2}$$

Note that the relation RT does not satisfy the condition for compositionality mentioned in section 3. For a stream o^2 on the concrete level arbitrary many streams o^1 of the abstract level (fulfilling the relation RT) exist. This contradicts the requirement for IAR-relations. Since for the aim of this paper only the overall result of the simulation is interesting, the compositionality of this refinement step can be dropped. If this component should be integrated in an environment, further considerations would be necessary and another, compositional refinement relation should be given. When relating abstract with concrete streams, this relation would insert $\surd$-messages exactly at these points in the stream where CSS would do "empty" steps, i.e. steps with no output of any events. Due to the higher

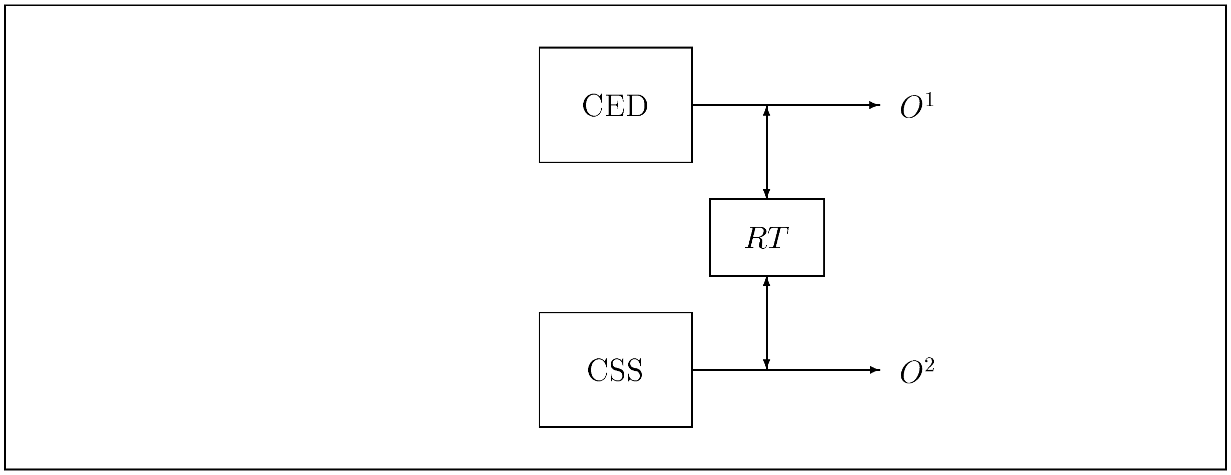

Figure 4: Refinement of CSS to CED

complexity of the formal treatment and since there is no need concerning the purpose of this paper, this is not further realized here.

The formal notion of the refinement statement, saying that the output of both CED and CSS is the same relative to RT, is given by (with † denoting the empty relation)

$$\text{CED} \overset{(\dagger;RT)}{\rightsquigarrow} \text{CSS}.$$

The proof is based on a comparison of the calculations of both simulators. CED performs only *efficient* steps of the form $z \xrightarrow{o} z'$ (with z and z' as internal states), while CSS executes several *empty* steps until its vt reaches the value when the same efficient step has to be performed. Thus, the corresponding calculation has the form

$$z \xrightarrow{\surd} z_* \xrightarrow{\surd} \ldots \xrightarrow{\surd} z_{**} \xrightarrow{o} z'$$

with the efficient step at the end. In the proof, the corresponding internal states of both simulators are related by the refinement relation r, defined by

$$r((s, ev, vt)) = (s, ev, nxt(ev, vt))$$

The proof with all details can be found in appendix B.1.

6.2 Refinement of CSS to DSS

When comparing the simulators CSS and DSS one can observe that their behaviour visible from outside is nearly the same, since both simulators work in a stepwise manner and do simulation steps for all points of time. Since in DSS all results are passed through the

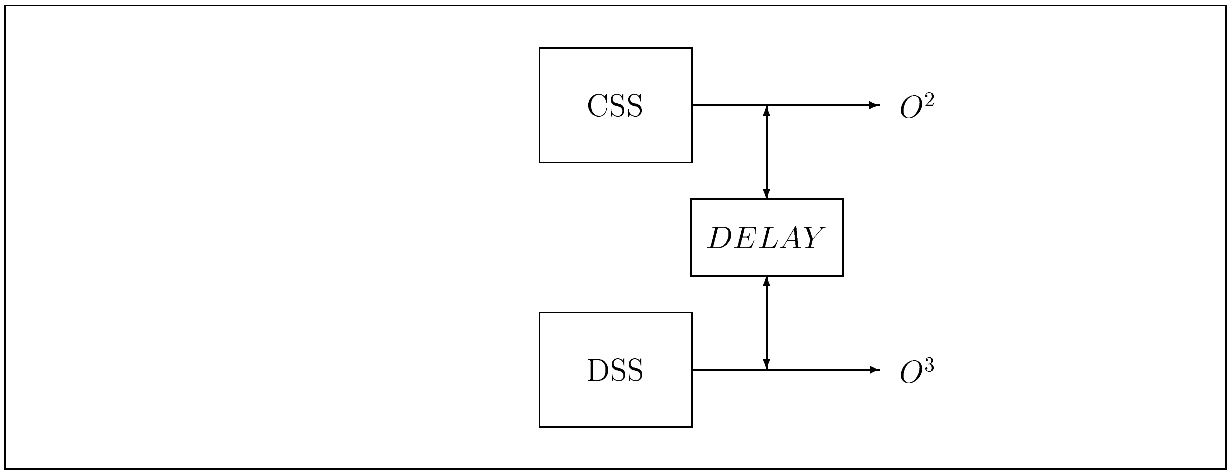

Figure 5: Refinement of CSS to DSS

controller CLSS the output is sent to channel O one tick later. This interaction refinement can be depicted as done in Figure 5 with the relation $DELAY$ easily defined by

$$DELAY[o^2; o^3] \equiv \quad o^3 = \surd \; \& \; o^2$$

The main aspect of this refinement step is to show that the distribution of the computation to several components still leads to the same results as delivered from the centralized CSS. To show that the outputs of both simulators are the same (with the time delay considered), the states of both systems can be related as described now. The state Z^2 of CSS is simply given by the values of its data elements. The state Z^3 of DSS in a specific moment is given by the product of the states of all components SP$_i$SS *and* the messages that are just being transmitted on the internal channels. Since the synchronized model is chosen, on all channels there is at most one message at a time. Between the states of both simulators a relation R can be defined stating that both simulators are in *equivalent states*, meaning they will produce the same output (modulo the mentioned delay) from now on:

$$\begin{aligned} R[Z^2, Z^3] \equiv \forall i \in N : \; & part_i(s) = s_i \wedge \\ & \Pi_i(ev) = ev_i \cup c_{j \in N,i} \wedge \\ & vt_i = vt \end{aligned}$$

The second line, for example, states what is necessary for the event sets occurring in both simulators to make the states Z^2 and Z^3 be in relation R: the part of event set of CSS that belongs to partition i must be equal to the event set stored already in SP$_i$SS together with the events that are just on their way to SP_iSS and received by it in the next step.

With use of the algebraic specifications of the auxiliary functions it can be proved that $\forall t \in TIME : R[Z^2.t; Z^3.t]$. From that it can be concluded that $\forall t \in TIME : o^2.t = o^3.(t+1)$, which implies the proof obligation that is proved in detail in appendix B.2:

$$\text{CSS} \overset{(\dagger; DELAY)}{\rightsquigarrow} \text{DSS}$$

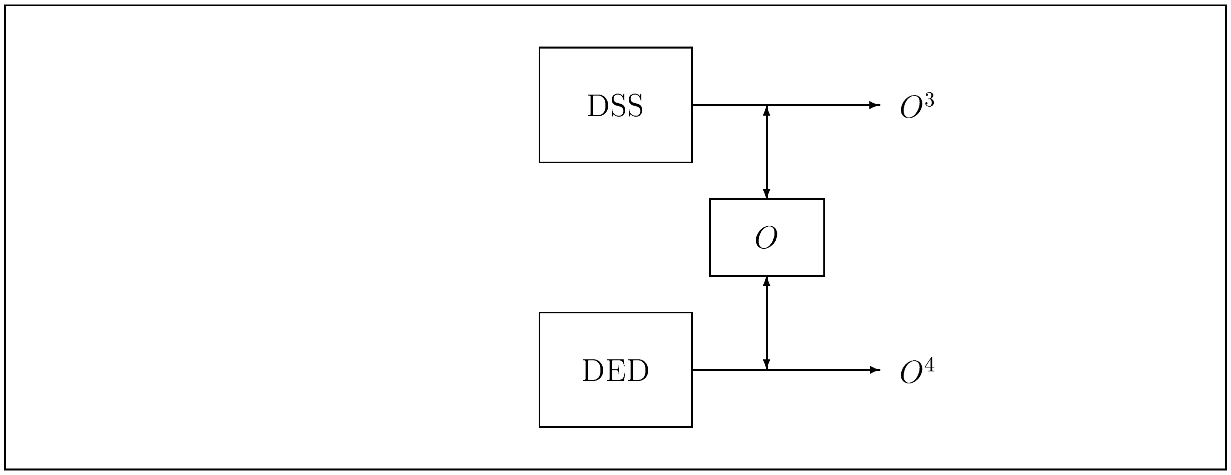

Figure 6: Refinement of DSS to DED

6.3 Refinement of DSS to DED

To show that DSS and DED calculate the same simulation results demands a precise specification what "same results" means in this context, since the way they calculate their results is quite different. While DSS follows a very predictable and easy manner of calculation, the method of DED is much more complex. Results of DSS are sent directly after calculation via the controller. The simulators of DED keep the results back for a while until an according *gvt*-message is arriving. So the IAR-relation called O in Figure 6 has to reflect this idea. When defining this relation, the detailed contents of the events have to be considered.

The TimeWarp-mechanism is made invisible to the environment concerning the cancellation mechanism. So all occurring events are (positive) events, antimessages do not appear in the resulting stream. Only the position in the stream, i.e. the Focus-time when they appear, is different. This can be formulated by the relation O through

$$\begin{aligned} O[o^3; o^4] \quad \equiv \quad & \forall t, t_g, t_e \in TIME : \\ & [+, t_g, t_e, d] \in o^4.t \Rightarrow [+, t_g, t_e, d] \in o^3.(t_g + 2) \quad \wedge \\ & [+, t_g, t_e, d] \in o^3.t \Rightarrow t = t_g + 2 \wedge \exists t' \in TIME : [+, t_g, t_e, d] \in o^4.t' \end{aligned}$$

If an event occurs in the result stream of DED, then it occurs also in the stream of DSS exactly two ticks after being generated in that simulator, and, if there is an event in the stream of DSS, then it is located at the right position (namely two ticks after being generated) and this event is also appearing at *some* time in the output of DED. This relation (together with all the following) complies the condition of an IAR-relation, as shown in appendix B.4.

On the basis of this relation the following refinement step can be proved:

$$\text{DSS} \overset{(\dagger;O)}{\rightsquigarrow} \text{DED}$$

The complexity of the proof can be reduced by taking advantage of the compositionality. If suitable IAR-relations can be defined for all internal channels X_i, Y_i and C_{ij}, and appropriate refinement relations can be shown for all components, the validity of the statement above can be concluded. These relations can be defined (using the same letter for both the channels and the relations) through

$$
\begin{array}{lcl}
Y[\vec{y}^3; \vec{y}^4] & \equiv & \forall i \in N : \\
& & \quad [+, t_g, t_e, d] \in y_i^4.t \Rightarrow [+, t_g, t_e, d] \in y_i^3.(t_g + 1) \\
& & \wedge \quad [+, t_g, t_e, d] \in y_i^3.t \Rightarrow t = t_g + 1 \wedge \exists t' \in TIME : [+, t_g, t_e, d] \in y_i^4.t' \\
X[\vec{x}^3; \vec{x}^4] & \equiv & \forall i \in N : \forall t \in Time : \\
& & \quad x_i^3.t = \surd \\
& & \wedge \quad (x_i^4.t = \surd \ \vee \ \exists gvt : x_i^4.t = gvt) \\
& & \wedge \quad \forall l \in TIME : \exists k, gvt \in TIME : \quad gvt \geq l \ \wedge \ x_i^4.k = gvt \\
C[c^3; c^4] & \equiv & \forall i, j \in N : \\
& & \quad [+, t_g, t_e, d] \in c_{ij}^4.t \wedge \forall k \in TIME, k > t : [-, t_g, t_e, d] \notin c_{ij}^4.k \Rightarrow \\
& & \qquad\qquad [+, t_g, t_e, d] \in c_{ij}^3.(t_g + 1) \\
& & \qquad \vee \quad \big(i = j \wedge \exists l \in N : [+, t_g, t_e, d] \in c_{li}.(t - 1)\big) \\
& & \wedge \quad [+, t_g, t_e, d] \in c_{ij}^3.t \Rightarrow \\
& & \qquad\qquad t = t_g + 1 \\
& & \qquad \wedge \quad \exists t' \in TIME : [+, t_g, t_e, d] \in c_{ij}^4.t' \\
& & \qquad \wedge \quad \forall k \in TIME, k > t' : [-, t_g, t_e, d] \notin c_{ij}^4.k
\end{array}
$$

These relations describe the connection between the "simple" streams of DSS and the "complex" streams (with another order of the received messages and including antimessages and stragglers) of DED. For example the relation C states the following concerning the streams on the channels between SP$_i$SS resp. SP$_i$ED. If there is an event in a stream of DED, and there comes no corresponding cancelling antimessage later on, then this event is also occurring at the right position in the stream of DSS, or this event was just received one tick before and re-sent by a component to itself. In addition, for the other way round, if there is an event in a stream of DSS, then it is at the right position (one tick after being generated) and it is also occurring in the stream of DED with no cancelling message coming later. So the idea why all streams of DSS and DED contain the same information from an abstract point of view must be "coded" in these relations. This is a very important part of the verifying process that cannot be automated.

With these relations the proof obligation can be formulated by two local refinements (one for the controller and one for the simulating components) that can be proved separately.

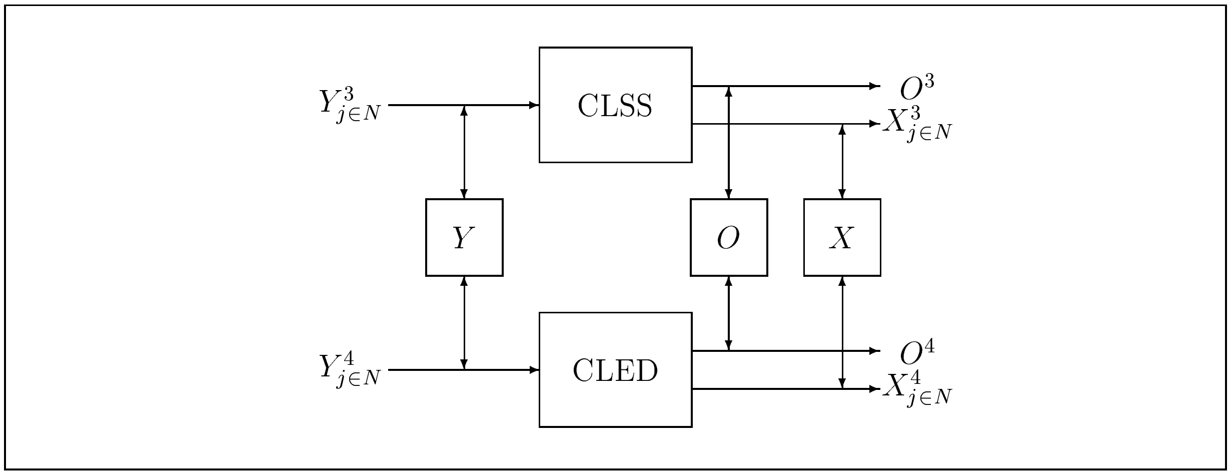

Figure 7: Refinement of CLSS to CLED

Refinement from CLSS to CLED

The statement of the interaction refinement

$$\text{CLSS} \overset{(Y;O\|X)}{\rightsquigarrow} \text{CLED}$$

is represented in Figure 7. The proof is straightforward and done by just expanding all involved relations. It is shown in appendix B.3.1.

Refinement from SP$_i$SS to SP$_i$ED

The statement

$$\text{SP}_i\text{SS} \overset{(X\|C;Y\|C)}{\rightsquigarrow} \text{SP}_i\text{ED}$$

which is illustrated by Figure 8 is more interesting. To prove it, the single computation steps of both simulators must be compared. Since the communication mechanism of SP$_i$ED is much more complicated than the one for SP$_i$SS and rollbacks occur at this abstraction layer, this comparison is not possible in a direct way. Hence the following steps have to be carried out:

- First it is shown that the proof obligation is valid for special streams of SP$_i$ED, namely streams *without* rollbacks and *without* interfering *gvt*-messages that interrupt the normal computation for one step. For this specific subset of streams a direct comparison of the calculation steps is possible. On the additional assumption that all events in *outQ* are eventually sent via channel Y (this can be concluded from the assumed correctness of the *gvt*-algorithm) the resulting outputs of the components of both abstraction layers can be shown to be identical.

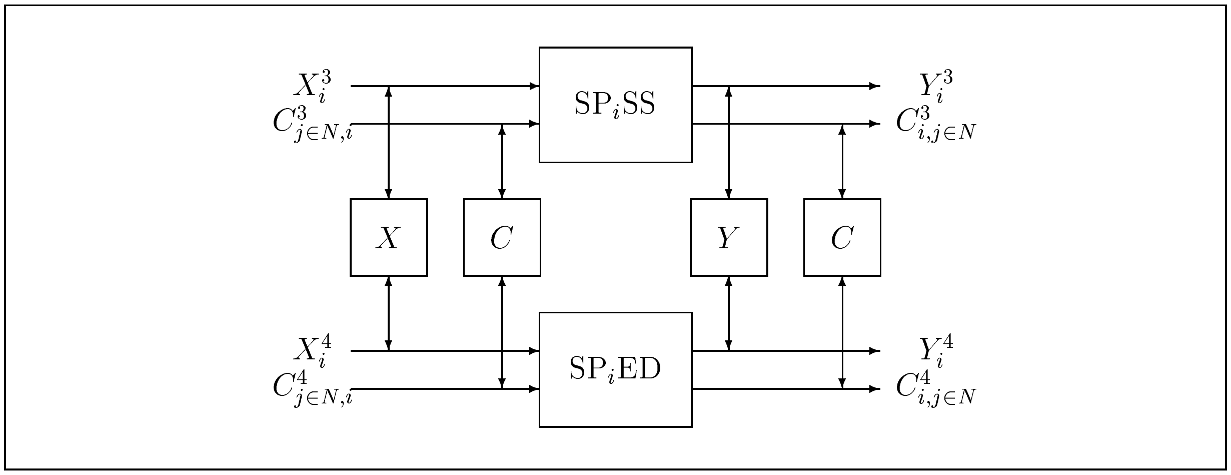

Figure 8: Refinement of SP$_i$SS to SP$_i$ED

- It must be proven that for streams containing rollbacks (i.e. input streams that trigger rollbacks and output streams resulting from computations with rollbacks) and containing *gvt*-messages similar streams without rollbacks and with delayed *gvt*-messages exist that belong to the same abstract streams of layer DSS. The general refinement relation can then be concluded.

A proof sketch can be found in appendix B.3.2. For the first step, the states of both simulators are compared by a relation, and one computation step of SP$_i$ED is related to several steps of SP$_i$SS with a similar input/output behaviour. The equality of the abstract counterparts of these two different kinds of streams (with and without rollbacks) can be demonstrated intuitively by using diagrams.

6.4 Overall Refinement

The stepwise refinements in the previous sections can now be combined using the transitivity of the refinement relation $\rightsquigarrow$. It can be directly concluded that

$$\text{CED} \overset{(\dagger;RT\succ DELAY\succ O)}{\rightsquigarrow} \text{DED}$$

with $\succ$ as a composing operator for relations. To make this result more readable, a function *result* is defined (in appendix A) that abstracts from any information about time and causality in the resulting streams, and just delivers the set of all occurring events in a stream. With this function the equality

$$result(\text{CED}()) = result(\text{DED}())$$

is valid (proved in appendix B.5), stating that DED returns the same resulting events as CED does - so the TIMEWARP-mechanism is correct!

7 Conclusion

In this paper a distributed, event-driven simulator was developed step by step. The development started with a specification of a simulator that is as simple as possible since this step must be validated and cannot be verified. Then two intermediate steps are performed, leading finally to a complex specification of a system forming a distributed simulator that implements TimeWarp. The development steps are formally described and verified using the refinement concept of Focus.

While modelling the simulators, some extensions of the description methods have been proposed that turned out to be useful. For instance, an extensive multiple use of similar components occured, revealing the need for appropriate notions. A suggestion for tabular specifications was given that uses a variable number of communication channels, described by parametrized columns. In addition, a tabular notion for describing the communication behaviour for synchronous communication was suggested. Some ideas that could lead to an improved support of the treatment of refinement relations can be found in the detailed proofs, e.g. proofs reflecting the *states* of systems by using the concept of *refinement mappings*.

During this case study some experiences were gained which are summarized in the following: Since Focus offers a wide variety of techniques and different variants of the semantic model, it turned out to be difficult to decide in favour of one possibility and choose the appropriate option. But once this obstacle is managed, the techniques of Focus make a specification of TimeWarp possible that is much more compact than informal descriptions, and states its behaviour in a clear and unambigious way suitable for further investigation. So the specification meets the requirements of a formal description.

The notion of refinement offered by Focus turned out to be quite powerful. All refinement relations could be expressed as *interaction refinements*. The property of compositionality and modularity was quite useful to structure the development and the proofs. So the importance of these concepts is confirmed again by the experiences gained here.

The modelling of the simulators was not straightforward, and several decisions concerning the level of abstraction had to be made. So the specifications were not developed in one monolithic step as it could appear from this presentation, but they were often modified and gradually improved. During formal proving some inconsistencies and insufficiencies were found that had to be corrected. For example some requirements of the auxiliary functions were found that are essential but were forgotten before. So the proofs also turned out to be - next to the statement of correctness they verify - a good validation method for the specifications, because they enforce an intensive occupation with the specifications during that several errors can be revealed.

The proofs turned out to be quite difficult, what is not too surprising since TimeWarp is a complicated distributed mechanism. But even for facts that seem to be quite obvious a lot of technical overhead has to be done. There is naturally a strong interconnection between the chosen model and the proofs, and trying to find the proofs leads to modifications of the specifications, which again have some influence on already existing proofs that have to be modified again. So it seems that the process of specifiying and verifying cannot be totally separated, even though an ideal way of developing a system would realize this separation: The specifications are done by focusing just on the application itself, while the proofs are carried out independently - in the ideal case even automatically.

These experiences give some motivation for future work to be done. Since a formal description of TimeWarp is available now, further investigations are possible. Additional refinement steps can be done, leading to a verified implementation of a distributed and efficient simulator. The *gvt*-algorithm can be modelled in detail and verified with the methods of Focus. And it is possible to examine formally if TimeWarp is really faster than other ways of (conservative) synchronization, and different versions (e.g. the different cancellation mechanisms) of TimeWarp could be compared.

Concerning Focus, also some ideas for further improvements arised. In this work several components occurred, that compute essentially the same results, but show a different behaviour concerning time. It could be fruitful to investigate the concept of *re-timing* in further detail and make it available for the methodology Focus. To broaden the acceptance of Focus for users not too familiar with formal methods, it could be useful to offer a simpler and more pragmatic introduction to Focus or just to a selected sub-part of Focus, i.e. a kind of "Focus light". Furthermore a *guideline* for developing systems with Focus could help to reach this aim (as done for dynamic systems in [HS97a]). Some additional support for proofs would also improve the methods of Focus, concerning the general proof principles as well as the use of (semi-)automated provers during the process of development. With an integrated tool for specifying, verifying and simulating a system the usability would grow essentially. With AutoFocus the first steps are taken in this direction ([HS97b]).

References

[Bau94] Herbert Bauer. *Verteilte diskrete Simulation komplexer Systeme.* PhD thesis, Technische Universität München, 1994.

[BBSS97] Manfred Broy, Max Breitling, Bernhard Schätz, and Katharina Spies. Summary of Case Studies in FOCUS - Part II. Technical Report SFB 342/24/97 A, Technische Universität München, 1997.

[BDD$^+$93] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report SFB 342/2-2/92 A, Technische Universität München, 1993.

[Bro92] Manfred Broy. Compositional Refinement of Interactive Systems. Technical Report 89, Digital Systems Research Center, Palo Alto, 1992. To appear in JACM.

[Bro93] Manfred Broy. (Inter-)Action Refinement: The Easy Way. In Manfred Broy, editor, *Program Design Calculi*, volume 118 of *Computer and System Sciences, NATO ASI Series.* Springer, 1993.

[BS97] Manfred Broy and Ketil Stølen. FOCUS *on System Development.* Springer, 1997. Manuscript, to appear.

[Fuj90] Richard M. Fujumoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[Gaf85] A. Gafni. *Space Management and Cancellation Mechanism for Time Warp.* PhD thesis, University of Southern California, 1985.

[HS97a] Ursula Hinkel and Katharina Spies. Spezifikationsmethodik für mobile, dynamische FOCUS-Netze. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch 1997.* GMD Verlag (St. Augustin), 1997.

[HS97b] Franz Huber and Bernhard Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch 1997, pp. 343-352.* GMD Verlag (St. Augustin), 1997.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AUTOFOCUS - A Tool for Distributed Systems Specification . In Bengt Jonsson and Joachim Parrow, editors, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 1135, pages 467–470. Springer, 1996.

[Jef85] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[JS85] David R. Jefferson and Henry Sowizral. Fast Concurrent Simulation using the Time Warp mechanism. *Proceedings of the SCS Distributed Simulation Conference*, pages 63–69, 1985.

[KRF$^+$96] B. Kannikeswaran, R. Radhakrishnan, P. Frey, P. Alexander, and P.A. Wilsey. Formal Specification and Verification of the pGVT algorithm. In M.-C. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science 1051, pages 405 – 425. Springer, 1996.

Acknowledgements

I would like to thank Bernhard Schätz and Rolf Schlagenhaft for their discussions about the TimeWarp mechanism and its formalization, and Manfred Broy, Ursula Hinkel, Katharina Spies and Konstanze Tauber for their helpful comments about earlier drafts of this paper.

A Definitions

The following definitions, abbreviations, properties and operators are used in the proofs in section B. The properties of the auxiliary functions are formulated in a form similar to algebraic specifications. When implementing these functions, it just has to be assured that they fulfill the properties given here.

Types of Channels and Constants

C	$=$	$EVENTS^{\underline{\infty}}$
$EVENTS$	$=$	$\mathcal{P}(\{[sn, t_g, t_e, d] \mid sn \in \{+, -\}, t_g, t_e \in TIME^+\})$ with d containing the description of the change in the simulation model, caused by an event (not further specified). Note that $\mathcal{P}(A)$ denotes the powerset of A.
EV	$\subset$	$EVENTS$
$HIST$	$=$	$STATES \times TIME$
n	$:$	constant defining the number of partitions of the model
N	$=$	$\{1, 2, \ldots, n\}$
O	$=$	$EVENTS^{\underline{\infty}}$
$STATES$	$:$	Set of states of the simulated world (not further specified), including static and dynamic aspects
$START$	$\in$	$STATES$
$TIME$	$=$	$\{0, 1, 2, \ldots\}$
$TIME^+$	$=$	$TIME \cup \{+\infty\}$
X	$=$	$\{vt \mid vt \in TIME\}^{\underline{\infty}}$
Y	$=$	$(EVENTS \cup \{info\})^{\underline{\infty}}$
Z	$:$	Set of data needed for implementing the gvt-algorithm
$init$	$\in$	Z

Abbreviations

A_N	$=$	$\bigcup_{j \in N} A_j$
$A_{N,i}$	$=$	$\bigcup_{j \in N} A_{j,i}$
$A_{i,N}$	$=$	$\bigcup_{j \in N} A_{i,j}$
$\vec{x}$	$=$	$(x_1, \ldots, x_n)$

Types of the Auxiliary Functions

$cancel$	:	$EVENTS \times TIME \times TIME \rightarrow EVENTS$
get	:	$HIST \times TIME \rightarrow STATES$
gvt	:	$Z \rightarrow TIME^{+}$
$join$	:	$(STATES \times \ldots \times STATES) \rightarrow STATES$
nxt	:	$EVENTS \times TIME \rightarrow TIME$
nt	:	$EVENTS \times TIME \rightarrow TIME$
$next$	:	$STATES \times EVENTS \times TIME \rightarrow STATES$
P_i	:	$EVENTS \rightarrow EVENTS$
Π_i	:	$EVENTS \rightarrow EVENTS$
$part_i$	:	$STATES \rightarrow STATES$
$+$	:	$EVENTS \times EVENTS \rightarrow EVENTS$
$result$	:	$EVENTS^{\underline{\infty}} \rightarrow EVENTS$
$select$	:	$EVENTS \times TIME \rightarrow EVENTS$
sim	:	$STATES \times EVENTS \times TIME \rightarrow EVENTS$
$triggered$	:	$Z \rightarrow BOOLEAN$
$update$	:	$Z \times Y \rightarrow Z$

Algebraic Properties of the Auxiliary Functions

The element $+\infty$ is an upper bound for all elements in $TIME$:

$$\forall t \in TIME : +\infty > t \tag{1}$$

The union-operator is generalized to be defined on the special constants $\surd$ and $info$. These two are treated like the empty set:

$$A \cup \surd = \surd \cup A = A \tag{2}$$
$$A \cup info = info \cup A = A \tag{3}$$

The function $result$ collects all elements occurring in the given stream and returns them as one set:

$$result(x) = \bigcup_{t \in Time} x.t \tag{4}$$

The function call $nxt(ev, vt)$ ("next time") returns the earliest simulation-time t that is contained as execution-time t_e in the events ev with $t > vt$. If there is no further event in *ev*, then $+\infty$ will be returned (the function *min* is assumed to return the minimum of a set, and $+\infty$ if this set is empty):

$$nxt(ev, vt) = min\{t_e | t_e > vt \wedge \exists t_g, d.\ [+, t_g, t_e, d] \in ev\} \tag{5}$$

The function $nt(ev, vt)$ is used to make some proofs more readable, and is similar to *nxt*, but returns a simulation-time t with $t \geq vt$:

$$nt(ev, vt) = min\{t_e | t_e \geq vt \wedge \exists t_g, d.\ [+, t_g, t_e, d] \in ev\} \tag{6}$$

If there is no event in *ev* with the execution-time $t_e = vt$, then a simulation step for *vt* is neutral, i.e. does not change the state s and does not return any events:

$$nt(ev, vt) > vt \Rightarrow s = next(s, ev, vt) \wedge sim(s, ev, vt) = \emptyset \tag{7}$$

If there are no events left (i.e. *vt* is set to $+\infty$), the simulation steps are neutral:

$$s = next(s, ev, +\infty) \wedge sim(s, ev, +\infty) = \emptyset \tag{8}$$

The *sum* of two sets is defined to be the union of the sets with all matching messages and their antimessages being neutralized:

$$A + B = (A \cup B) - \{e, \bar{e} \mid \exists t_g, t_e, d. \quad \begin{array}{l} e = [+, t_g, t_e, d] \wedge \\ \bar{e} = [-, t_g, t_e, d] \wedge \\ e, \bar{e} \in A \cup B \qquad \} \end{array} \tag{9}$$

$\Pi_i(ev)$ contains all events in *ev* needed for simulating partition i. The basic properties are:

$$\Pi_i(\emptyset) = \emptyset \tag{10}$$
$$\Pi_i(A \cup B) = \Pi_i(A) \cup \Pi_i(B) \tag{11}$$
$$i \neq j \quad \Rightarrow \quad \Pi_i(A) \cap \Pi_j(A) = \emptyset \tag{12}$$
$$\bigcup_{i \in N} \Pi_i(A) = A \tag{13}$$

$P_i(ev)$ contains all events of *ev* created during a simulation step in partition i:

$$P_i(\emptyset) = \emptyset \tag{14}$$
$$P_i(A \cup B) = P_i(A) \cup P_i(B) \tag{15}$$
$$i \neq j \quad \Rightarrow \quad P_i(A) \cap P_j(A) = \emptyset \tag{16}$$
$$\bigcup_{i \in N} P_i(A) = A \tag{17}$$

The single components of the distributed simulator together produce the same events as the centralized simulator (this is an instance of axiom (17))

$$\bigcup_{i \in N} P_i(sim(s, ev, vt)) = sim(s, ev, vt) \tag{18}$$

A single component works correctly, i.e. the events that are generated in partition i (left-hand-side) are indeed returned from the simulation that does only simulate partition i (right-hand-side):

$$P_i(sim(s, ev, vt)) = sim(part_i(s), \Pi_i(ev), vt) \tag{19}$$

The description of partition i after a step in the centralized simulator is exactly the same as the description contained in a single component after doing a local simulation step. This is an analogon to (19):

$$part_i(next(s, ev, vt)) = next(part_i(s), \Pi_i(ev), vt) \tag{20}$$

The function *get* returns the state out of *hist* that was stored with vt as timestamp:

$$(s, vt) \in hist \quad \Leftrightarrow \quad get(hist, vt) = s \tag{21}$$

All elements out of *outQ* that have a timestamp smaller than *gvt* are selected by *select*:

$$[+, t_g, t_e, d] \in outQ \wedge t_g \leq gvt \quad \Leftrightarrow \quad [+, t_g, t_e, d] \in select(outQ, gvt) \tag{22}$$

The function *cancel* yields the antimessages to those events out of *outQ* that are generated between vt' and vt.

$$[+, t_g, t_e, d] \in outQ \wedge vt' \leq t_g \leq vt \quad \Leftrightarrow \quad [-, t_g, t_e, d] \in cancel(outQ, vt', vt) \tag{23}$$

The interaction of the three functions *update*, *triggered* and *gvt* must assure that there will be a monotonic increasing of the *gvt* messages. This is just postulated by (let (z_i) a sequence of states $z_i \in Z$):

$$\begin{aligned} & z_0 = init \wedge z_{i+1} = update(z_i) \\ & \Rightarrow \forall m \in TIME : \exists j, t \in TIME : t \geq m \wedge triggered(z_j) = true \wedge gvt(z_j) = t \end{aligned} \tag{24}$$

Operators from Focus

Only quite informal definitions are summarized here. See [BS97] for further reference and the formal definitions.

$\dagger \in \emptyset \times \emptyset$	:	the empty relation
$A \succ B$	:	a “concatenation” of relations: $(A \succ B)[x; z] \Leftrightarrow \exists y : A[x; y] \wedge B[y : z]$
$A \| B$	:	a parallel composition of relations: $(A \|\| B)[(x, x'); (y, y')] \Leftrightarrow A[x; y] \wedge B[x'; y']$
$x \ \& \ xs$	:	stream resulting from appending x in front of the stream xs
$x \frown y$	;	the concatenation of the streams x and y
$\bar{x}$	:	the stream x with all $\surd$-messages removed
$A \rightsquigarrow B$	:	the refinement relation as mentioned in section 3
$S \otimes T$	:	a component consisting of two subcomponents S and T, interconnected in an appropriate way.
$A^{\underline{\infty}}$	:	an infinite timed stream containing elements out of the set A and $\surd$.

B Proofs

In this section of the appendix all proofs for the verification steps can be found. The proofs are conducted in a more mathematical style, and are not formulated in a rigorous, logical level as it would be necessary for (semi-) automated theorem provers.

B.1 Refinement of CED to CSS

The statement to be shown according to section 6 is

$$\text{CED} \overset{(\dagger;RT)}{\rightsquigarrow} \text{CSS}$$

Since no input streams have to be considered, this interaction refinement is slightly simpler than a general interaction refinement. Expanding the proof obligation leads to

$$\forall o^2 : \text{CSS}() = o^2 \Rightarrow \exists o^1 : \text{CED}() = o^1 \wedge RT[o^1; o^2] \tag{25}$$

Both components are state-oriented, deterministic and total. In the proof the single calculation steps with the respective output are related to each other. The states of CED and CSS are both characterized by a triple of the form (for $i \in \{1, 2\}$)

$$z^i = (s^i, ev^i, vt^i).$$

The initial states result from the initialization defined by the specifications:

$$\begin{aligned} z^1_{init} &= (START, EV, nxt(EV, 0)) \\ z^2_{init} &= (START, EV, 0) \end{aligned}$$

It turns out to be useful to introduce the abbreviation $x \overset{a/b}{\to} y$. It states that any component S (whose identity should be clear from the context) makes a transition from state x to state y by reading the message a and writing the messsage b. This can be expressed semantically by (with i and o denoting the respective input and output streams)

$$\begin{aligned} x \overset{a/b}{\to} y \quad &\Leftrightarrow \quad S[x](a\&i) = b\&S[y](i) \\ &\Leftrightarrow \quad \exists t \in TIME : z.t = x \wedge i.t = a \wedge z.(t+1) = y \wedge o.(t+1) = b \end{aligned}$$

This notion can be generalized to streams (instead of single messages a and b) in a straightforward way. Using this notation and with expanding the relation RT as defined in section 6 the proof obligation (25) can be reformulated to

$$\forall o^2, z_2 : z^2_{init} \overset{o^2}{\to} z^2 \quad \Rightarrow \quad \exists o^1, z^1 : z^1_{init} \overset{o^1}{\to} z^1 \ \wedge\ \overline{o}^1 = \overline{o}^2 \tag{26}$$

This will be shown with the help of the concept of *refinement mappings.* In order to do this, a function r is defined that relates similar states of both simulators. It maps states of CSS to states of CED, and is defined in this case by

$$r(s^2, ev^2, vt^2) =_{df} (s^2, ev^2, nt(ev^2, vt^2)) \tag{27}$$

and has two following properties that are shown later:

$$r(z^2_{init}) = z^1_{init} \tag{28}$$

and

$$\begin{aligned} &\forall m \in EVENTS, z^2, z^{2'} . \\ &\quad z^2 \xrightarrow{m} z^{2'} \;\Rightarrow\; r(z^2) \xrightarrow{m} r(z^{2'}) \vee \left(r(z^2) = r(z^{2'}) \wedge m = \surd \right) \end{aligned} \tag{29}$$

With the help of these properties of the function r the following stronger formalization of the proof obligation (26) is proved by induction over the length of o^2:

$$\forall o^2, z_2 : z^2_{init} \xrightarrow{o^2} z^2 \quad \Rightarrow \quad \exists o^1 : r(z^2_{init}) \xrightarrow{o^1} r(z^2) \;\wedge\; \overline{o}^1 = \overline{o}^2 \tag{30}$$

The base case for the induction described by $o^2 = \epsilon = o^1$ is trivial. With $x \frown y$ denoting the concatenation of streams, and for simplicity, allowing y to be a single message, let

$$o^{2'} =_{df} o^2 \;\frown\; m \tag{31}$$

and

$$z^2_{init} \xrightarrow{o^2} z^2 \xrightarrow{m} z^{2'}$$

If (30) is taken as induction hypothesis it can be concluded

$$\exists o^1 : r(z^2_{init}) \xrightarrow{o^1} r(z^2) \;\wedge \tag{32}$$
$$\overline{o}^1 = \overline{o}^2 \tag{33}$$

It remains to be proved

$$\exists o^{1'} : r(z^2_{init}) \xrightarrow{o^{1'}} r(z^{2'}) \;\wedge\; \overline{o^{1'}} = \overline{o^{2'}}$$

According to (29) there are two cases to investigate:

- For *case 1* assume

$$r(z^2) \xrightarrow{m} r(z^{2'}) \tag{34}$$

Then $o^{1'}$ can be taken as

$$o^{1'} =_{df} o^1 \frown m \tag{35}$$

With that it follows from (32) and (34) resp. (35), (33) and (31)

$$\begin{array}{l} r(z^2_{init}) \xrightarrow{o^1 \frown m} r(z^{2'}) \\ \overline{o^{1'}} = \overline{o^1 \frown m} = \overline{o^1} \frown \overline{m} = \overline{o^2} \frown \overline{m} = \overline{o^2 \frown m} = \overline{o^{2'}} \end{array}$$

- Assume for *case 2*

$$r(z^2) = r(z^{2'}) \wedge \tag{36}$$
$$m = \surd \tag{37}$$

Taken $o^{1'}$ simply as

$$o^{1'} =_{df} o^1 \tag{38}$$

so again it can be concluded from (32), (36) and (38) resp. (38), (33), (37) and (31)

$$\begin{array}{l} r(z^2_o) \xrightarrow{o^{1'}} r(z^{2'}) \\ \overline{o^{1'}} = \overline{o^1} = \overline{o^2} = \overline{o^2} \frown \overline{m} = \overline{o^2 \frown m} = \overline{o^{2'}} \end{array}$$

It remains to be shown that both of the mentioned properties of r are valid. They follow from the specifications of the simulators. Property (28) is obvious, following from the definitions of z^1_0 and z^2_0.

For the proof of (29) again two cases are to be distinguished:

- For *case 1* it is assumed that

$$nt(ev^2, vt^2) = vt^2$$

This means that at the simulation of the virtual time vt a "real" simulation step is done and not an "empty" step. Let

$$z^2 \xrightarrow{m^2} z^{2'}$$

be an arbitrary calculation step of CSS with $z^2 = (s, ev, vt)$. The indices are omitted here since the values for layer CED and CSS coincide. By expansion of the specification of CSS (Table 4) this step can be formulated as

$$(s, ev, vt) \overset{\Pi_0(sim(s,ev,vt))}{\rightarrow} (next(s, ev, vt), ev \cup sim(s, ev, vt), vt + 1)$$

and the values m^2 und $z^{2'}$ result to

$$m^2 = \Pi_o(sim(s, ev, vt)$$
$$z^{2'} = (next(s, ev, vt), ev \cup sim(s, ev, vt), vt + 1)$$

Following the definition of r (27) it can be derived

$$r(z^2) = (s, ev, nt(ev, vt)) = (s, ev, vt)$$
$$r(z^{2'}) = (next(s, ev, vt), ev \cup sim(s, ev, vt), nt(ev \cup sim(s, ev, vt), vt + 1))$$

If a calculation step of CED is inspected, starting from the state $r(z^2)$, this results in the statement

$$r(z^2) \overset{m^1}{\rightarrow} z^{1'}$$

with

$$m^1 = \Pi_o(sim(s, ev, vt)$$
$$z^{1'} = (next(s, ev, vt), ev \cup sim(s, ev, vt), nt(ev \cup sim(s, ev, vt), vt + 1))$$

Therefore $m^1 = m^2$ and $z^{1'} = r(z^{2'})$ are indeed valid.

- In *case 2*

$$nt(ev^2, vt^2) > vt^2$$

is assumed. The case $nt(ev, vt) = +\infty$ is included. From the algebraic specification of the auxiliary functions (7) and (8) it follows

$$next(s, ev, vt) = s$$
$$sim(s, ev, vt) = \emptyset$$

As a result a calculation step of CSS starting from $z^2 = (s, ev, vt)$ has the form

$$(s, ev, vt) \overset{\surd}{\rightarrow} (s, ev, vt + 1) =_{df} z^{2'}$$

So

$$r(z^2) = (s, ev, nt(ev, vt))$$
$$r(z^{2'}) = (s, ev, nt(ev, vt + 1))$$

Since further

$$nt(ev, vt) > vt \Rightarrow nt(ev, vt) = nt(ev, vt + 1)$$

it is shown that

$$r(z^2) \overset{\surd}{\rightarrow} r(z^{2'})$$

- The case

$$nt(ev^2, vt^2) < vt^2$$

cannot occur according to the definition (6) of nt.

$\square$

B.2 Refinement of CSS to DSS

The statement to be proved is

$$\text{CSS} \overset{(\dagger;DELAY)}{\rightsquigarrow} \text{DSS}$$

Since both CSS and DSS are components that are specified by states, the proof is using these internal states as basis. The state of CSS consists of a simple triple of its internal data states $Z^2 = (s, ev, vt)$. For the state of DSS the controller, the simulators and the messages on the internal channels have to be considered. Therefore, the state of DSS has to be described as the product of the states of all constituents. Since the synchronized model is chosen, it is sufficient to describe the state of a channel by the single message that is transferred at a certain time. The state Z^3 therefore is an element of the set

$$Z^3 \in (STATES \times EVENTS \times TIME)^n \times X^n \times Y^n \times C^{n^2}$$

and can be written in the form

$$\begin{aligned} Z^3 \;=\; & ((s_1, ev_1, vt_1), \ldots, (s_n, ev_n, vt_n), \\ & x_1, \ldots, x_n, \\ & y_1, \ldots, y_n, \\ & c_{11} \ldots, c_{1n}, c_{21}, \ldots c_{2n}, \ldots, c_{n1}, \ldots c_{nn}) \end{aligned}$$

In order to prove the refinement the states of the two simulators are related through a relation R. It is valid in the intial state, and stays valid at each calculation-step both simulators perform. The relation R states informally that CSS and DSS are in similar states concerning the progress of simulation. They pass similar states throughout their whole calculations. This relation is defined by

$$R[Z^2, Z^3] \equiv \forall i \in N : \begin{array}{l} part_i(s) = s_i \wedge \\ \Pi_i(ev) = ev_i \cup c_{N,i} \wedge \\ vt_i = vt \end{array} \tag{39}$$

For the initial states

$$Z^2.0 = (START, EV, 0)$$

and

$$Z^3.0 = (\ldots, (part_i(START), \Pi_i(EV), 0), \ldots, \surd, \ldots, \surd, \ldots, \surd, \ldots)$$

it is obvious that

$$R[Z^2.0; Z^3.0]$$

A calculation step of both systems can be represented by a transition from Z^2 to $Z^{2'}$ and from Z^3 to $Z^{3'}$ respectively with the abbreviations Z for $Z.t$ and Z' for $Z.(t+1)$ for an arbitrary $t \in TIME$.

According to the specifications it holds that

$$\text{CSS}[(s, ev, vt)]() = \Pi_0(sim(s, ev, vt)) \;\&\; \text{CSS}[(s', ev', vt')]()$$

with

$$s' = next(s, ev, vt) \tag{40}$$
$$ev' = ev \cup sim(s, ev, vt) \tag{41}$$
$$vt' = vt + 1 \tag{42}$$

and

$$\text{DSS}[(\ldots, (s_i, ev_i, vt_i) \ldots, x_i, \ldots, y_i, \ldots, c_{ij}, \ldots)]() =$$
$$\bigcup_{i=1}^{N} Y_i \;\;\&\;\; \text{DSS}[(\ldots, (s_i', ev_i', vt_i') \ldots, x_i', \ldots, y_i', \ldots, c_{ij}', \ldots)]()$$

with

$$\begin{aligned}
s'_i &= next(s_i, ev_i \cup c_{N,i}, vt_i) && (43)\\
ev'_i &= ev_i \cup (ev_N)_i = ev_i \cup c_{N,i} && (44)\\
vt'_i &= vt_i + 1 && (45)\\
c'_{ij} &= \Pi_j(sim(s_i, ev_i \cup c_{N,i}, vt_i)) && (46)\\
x'_i &= \surd && (47)\\
y'_i &= \Pi_0(sim(s_i, ev_i \cup c_{N,i}, vt_i)) && (48)
\end{aligned}$$

Now

$$R[Z^2; Z^3] \Rightarrow R[Z^{2'}; Z^{3'}]$$

is proved by assuming $R[Z^2; Z^3]$ valid and showing for all $i \in N$:

- $part_i(s') = s'_i$

$$\begin{aligned}
&part_i(s') && = [40]\\
&part_i(next(s, ev, vt)) && = [20]\\
&next(part_i(s), \Pi_i(ev), vt) && = [39]\\
&next(s_i, ev_i \cup c_{N,i}, vt_i) && = [43]\\
&s'_i
\end{aligned}$$

- $\Pi_i(ev') = ev'_i \cup c'_{N,i}$

$$\begin{aligned}
&\Pi_i(ev') && = [41]\\
&\Pi_i(ev \cup sim(s, ev, vt)) && = [11]\\
&\Pi_i(ev) \cup \Pi_i(sim(s, ev, vt)) && = [18]\\
&\Pi_i(ev) \cup \Pi_i(\textstyle\bigcup_{j\in N} P_j(sim(s, ev, vt))) && = [19]\\
&\Pi_i(ev) \cup \Pi_i(\textstyle\bigcup_{j\in N} sim(part_j(s), \Pi_j(ev), vt)) && = [11]\\
&\Pi_i(ev) \cup \textstyle\bigcup_{j\in N} \Pi_i(sim(part_j(s), \Pi_j(ev), vt)) && = [39]\\
&ev_i \cup c_{N,i} \cup \textstyle\bigcup_{j\in N} \Pi_i(sim(s_j, ev_j \cup c_{N,j}, vt_j)) && = [46]\\
&(ev_i \cup c_{N,i}) \cup \textstyle\bigcup_{j\in N} c'_{j,i} && = [44]\\
&ev'_i \cup c'_{N,i}
\end{aligned}$$

- $vt'_i = vt'$

$$\begin{aligned}
&vt'_i && = [45]\\
&vt_i + 1 && = [39]\\
&vt + 1 && = [42]\\
&vt'
\end{aligned}$$

With $R[Z^2.0; Z^3.0]$ and $R[Z^2.t; Z^3.t] \Rightarrow R[Z^2.(t+1); Z^3.(t+1)]$ the statement

$$\forall t \in TIME : R[Z^2.t; Z^3.t] \qquad (49)$$

can be concluded by induction.

Let $Z^2 = (s.t, ev.t, vt.t)$ be the state for a time t. On the one hand, the specification of CSS yields

$$o^2.(t+1) = \Pi_o(sim(s.t, ev.t, vt.t),$$

but on the other hand it can be concluded that

$$\begin{array}{lcl}
o^3.(t+2) & = & \text{[Table 5]} \\
\bigcup_{j \in N} y_j.(t+1) & = & \text{[Table 6]} \\
\bigcup_{j \in N} \Pi_o(sim(s_j.t, ev_j.t \cup c_{N,j}.t, vt_j.t)) & = & [11] \\
\Pi_o(\bigcup_{j \in N} sim(s_j.t, ev_j.t \cup c_{N,j}.t, vt_j.t)) & = & [49, 39] \\
\Pi_o(\bigcup_{j \in N} sim(part_j(s.t), \Pi_j(ev.t), vt.t)) & = & [19] \\
\Pi_o(\bigcup_{j \in N} P_i(sim(s.t, ev.t, vt.t))) & = & [18] \\
\Pi_o(sim(s.t, ev.t, vt.t)) & &
\end{array}$$

Together with the obvious fact

$$o^3.1 = \bigcup_{j \in N} Y_j.0 = \bigcup_{j \in N} \surd = \surd$$

it follows

$$\forall t \in TIME: \quad o^2.t = o^3.(t+1)$$

and together with the defined property of the semantic $o^2.0 = o^3.0 = \surd$ it follows the statement

$$\text{DSS}() = \surd \;\&\; \text{CSS}(),$$

that was to be proved. □

B.3 Refinement of DSS to DED

The proof of

$$\text{DSS} \overset{(\dagger;O)}{\rightsquigarrow} \text{DED}$$

is done by taking advantage of the modularity as mentioned already in section 6. So the two types of components can be refined separately.

B.3.1 Refinement of CLSS to CLED

The statement to be proved

$$\text{CLSS} \overset{(Y;O\|X)}{\rightsquigarrow} \text{CLED}$$

is according to the definition of refinement a consequence of

$$\begin{array}{l}\text{CLED}(\vec{y}^4) = (o^4, \vec{x}^4) \Rightarrow \\ \quad \exists \vec{y}^3, o^3, \vec{x}^3 : \text{CLSS}(\vec{y}^3) = (o^3, \vec{x}^3) \wedge Y[y^3; y^4] \wedge O[o^3; o^4] \wedge X[x^3; x^4]\end{array}$$

that is proved now. Let the left-hand-side of the implication be valid for arbitrary y_i^4. From the specification of CLED in Table 7 the following statement can be concluded:

$$o^4.(t+1) = \bigcup_{i \in N} y_i^4.t$$

From this it follows

$$[+, t_g, t_e, d] \in o^4.(t+1) \Leftrightarrow \exists i \in N : [+, t_g, t_e, d] \in y_i^4.t \tag{50}$$

The stream y_i^3 can be defined in a unique way from y_i^4 through the relation Y and

$$Y[y^3; y^4] \tag{51}$$

is therefore trivially valid. The output streams x_i^3 and x_i^4 show the property

$$\begin{array}{l}\forall i \in N, t \in TIME : x_i^3.t = \surd \\ \forall i \in N, t \in TIME : (x_i^4.t = \surd \vee \exists gvt \in TIME : x_i^4.t = gvt) \\ \forall l \in TIME : \exists k, gvt \in TIME : \quad gvt \geq l \ \wedge \ x_i^4.k = gvt\end{array}$$

due to the specification of the controllers and the postulated "liveness" of the gvt-algorithm (24). Hence it follows immediately

$$X[\vec{x^3}; \vec{x^4}] \tag{52}$$

The stream o^3 is fixed uniquely according to specification CLSS in Table 5 by the input $\vec{y^3}$ through

$$[+, t_g, t_e, d] \in o^3.(t+1) \Leftrightarrow \exists i \in N : [+, t_g, t_e, d] \in y_i^3.t \tag{53}$$

From that it follows

$$\text{CLSS}(\vec{y^3}) = (o^3, \vec{x^3}) \tag{54}$$

Now it can be shown

$$\begin{array}{lll} [+, t_g, t_e, d] \in o^4.(t+1) & \Rightarrow & [50] \\ \exists i \in N : [+, t_g, t_e, d] \in y_i^4.t & \Rightarrow & [\text{Definition of } Y\text{, Section 6.3}] \\ \exists i \in N : [+, t_g, t_e, d] \in y_i^3.(t_g+1) & \Rightarrow & [53] \\ \exists i \in N : [+, t_g, t_e, d] \in o^3.(t_g+2) & & \end{array}$$

and

$$\begin{array}{lll} [+, t_g, t_e, d] \in o^3.(t+1) & \Rightarrow & [53] \\ \exists i : [+, t_g, t_e, d] \in y_i^3.t & \Rightarrow & [\text{Definition of } Y\text{, Section 6.3}] \\ \exists i : t = t_g + 1 \wedge \exists t' : [+, t_g, t_e, d] \in y_i^4.t' & \Rightarrow & [50] \\ t = t_g + 1 \wedge \exists t' : [+, t_g, t_e, d] \in o^4.(t'+1) & & \end{array}$$

and with that it follows

$$O[o^3; o^4] \tag{55}$$

Combining (54), (51), (55) and (52) shows the statement to be proved. □

B.3.2 Refinement of SP$_i$SS to SP$_i$ED

The proof obligation

$$\text{SP}_i\text{SS} \overset{(X\|C;Y\|C)}{\rightsquigarrow} \text{SP}_i\text{ED}$$

is expanded to

$$\begin{array}{l} \text{SP}_i\text{ED}(x^4, c_{Ni}^4) = (y^4, c_{iN}^4) \\ \Rightarrow \quad \exists x^3, y^3, c_{ij}^3 : \text{SP}_i\text{SS}(x^3, c_{Ni}^3) = (y^3, c_{iN}^3) \wedge X[x^3; x^4] \wedge Y[y^3; y^4] \wedge C[c^3; c^4] \end{array}$$

The proofs are not carried out in detail, only the basic ideas are sketched. The full proofs would need a lot of technical considerations with many different cases, and would not be a help for a better understanding of the basic ideas. The proofs follow the idea described in section 6, i.e. the formal treatment is splitted in two parts:

- considering the computation without rollbacks and without interfering *gvt*-messages, and
- showing how rollbacks can be "eliminated" and *gvt*-messages can be "delayed" with respect to the abstract view of the level of DSS.

Computation without rollbacks

To compare the calculation of SP$_i$SS with a calculation from SP$_i$ED free from rollbacks, a relation R is used to relate states of the simulators, given through

$$\begin{array}{rcl} R[z^3; z^4] & \equiv & s^3 = s^4 \\ & \wedge & vt^3 = vt^4 \\ & \wedge & \pi(ev^3 \cup C^3_{Ni}, vt^3) = \pi(ev^4 + C^4_{Ni}, vt^4) \end{array}$$

saying that two states are similar if s and vt contain the same values, and the event sets are the same when only the events with an execution time less or equal to vt are considered. π is formally defined through

$$\pi(ev, vt) =_{df} \{[sg, t_g, t_e, d] \mid [sg, t_g, t_e, d] \in ev \wedge t_e \leq vt\}$$

Now for every step of SP$_i$ED it can be shown that SP$_i$SS is doing a sequence of similar steps (namely the same step followed by a sequence of empty steps until the right vt is reached) with the same output :

$$\begin{array}{rcll} R[z^3.t^3; z^4.t^4] & \Rightarrow & \exists t^{3'}. & R[z^3.t^{3'}; z^4.(t^4+1)] \\ & & \wedge & ev \in \bigcup_{t^3 \leq k < t^{3'}} C^3_{iN}.k \Leftrightarrow ev \in outQ.(t^4+1) \end{array}$$

Since it is obvious that after some empty steps of SP$_i$SS the relation R is valid for the first state of SP$_i$ED, it can be followed together with the implication as induction step that the output of both simulators is the same, using the assumption that all events in $outQ$ are eventually sent.

It is easy to see that all events contained in $outQ$ are sent via y_i, i.e.

$$ev \in outQ.t \;\Rightarrow\; \exists k' \in TIME : k' \geq t \;\wedge\; \Pi_0(\{ev\}) \in y_i.k'$$

So assume that

$$ev \in outQ.t \quad \text{with} \quad ev = [+, t_g, t_e, d] \tag{56}$$

This must be a positive message as explained in B.5. From the liveness condition of the gvt-algorithm (24) it can be concluded that there will be a gvt-message on the channel x causing this event to be sent:

$$\begin{array}{rcll} \exists k, gvt \in TIME : & & gvt \geq t_g & (57) \\ & \wedge & x.k = gvt & (58) \\ & \wedge & k \geq t & (59) \end{array}$$

With (56) and (59) with the fact (here unproved) that events are not removed from *outQ* too early, it can be concluded that

$$ev \in out.k \tag{60}$$

From the specification of SP$_i$ED follows

$$y_i.(k+1) = \Pi_0(select(outQ.k, gvt)) \tag{61}$$

(60) and (57) with (22) lead to

$$ev \in select(outQ.k, gvt) \tag{62}$$

With (61) and (62) follows

$$\Pi_0(\{ev\}) \in y_i.(k+1)$$

that shows the statement to be proved. $\square$

Elimination of Rollbacks

In order to demonstrate the idea of the elimination of a rollback, Figure 9 is used. A calculation of SP$_i$ED with a rollback (upper half) is compared with a similar calculation without this rollback (lower half), i.e. with a calculation where the message causing the rollback was received early enough. In the first case, the simulation starts in the state (s, ev, vt). At times k to l the event sets c_0 to c_n are received (It can be assumed that no other rollbacks occur during this calculation, since then this one could be removed first). At time l a straggler is received, causing the cancellation of all outputs $o_1, \ldots, o_n$ at time $l+1$. The elements s and vt are set back to the original values, while ev' still contains the sum of ev with the events in $c_0, \ldots, c_n$. From there the next calculation step is done.

Now this calculation can be compared with another calculation *without* rollback: In that case, all messages $c_o, \ldots, c_n$ are received altogether at time k. So the calculation step done at this time is the same as the last one in the other case: State s and the event set $ev + c_o + \ldots + c_{n+1}$ are used for simulating time vt. Note that for both cases the corresponding streams of abstraction layer of DSS are the same, since all events are placed in the streams of DSS directly at the locations that correspond to their creation time.

Delay of *gvt*-messages

With Figure 10 it can be demonstrated how *gvt*-messages (with their consequences) are delayed, i.e. moved backwards in the streams. Assume that the event sets c_0, c_1 and c_2 are received at times k, $k+1$ and $k+2$, and some internal events d_0 (sent to itself by the

SP$_i$ED		k	k+1		l	l+1	l+2	
C_{iN}	$\vdots$	c_0	c_1	$\ldots$	c_n	c_{n+1}		$\vdots$
C_{Ni}	$\vdots$		o_1	$\ldots$	o_n	$-o_1, \ldots, -o_n$	o_{n+2}	$\vdots$
Z^4	$\vdots$	s ev vt		$\ldots$ $\ldots$ $\ldots$		s ev' vt	s'' ev'' vt''	

C_{iN}	$\vdots$	$c_0, \ldots, c_{n+1}$		$\vdots$
C_{Ni}	$\vdots$		o_{n+2}	$\vdots$
Z^4	$\vdots$	s ev vt	s'' ev'' vt''	$\vdots$
		k	$k+1$	

Figure 9: Elimination of a rollback

component at the preceding step) are received at tick k. In the upper case, gvt is received at time k. This causes the component just to send messages to the controller (not shown in the table), while the state Z^4 does not change. All received events $c_0 \cup d_0$ are sent to itself again via C_{ii}. In the next tick, a calculation step for vt is performed, using the set $ev \cup c_0 \cup c_1 \cup d_0$ as event set. As output o and d_1 are produced, and the state results as noted in the table.

If it is assumed[6] that c_1 contains messages not needed for simulating the time vt, the same behaviour is gained by delaying the gvt-message one tick backwards, illustrated by the lower half of the table. The calculation step for vt is here performed at time k, using the set $ev \cup c_0 \cup d_0$. The output o and the internal events d_1 are produced, and again s' and vt' and the proper event set are now forming the new state. At time $k+1$ now gvt is received, so that the state remains constant and all received events $c_1 \cup d_1$ are sent again, so that at the next tick $k+2$ in both cases the event set to be considered for the next step to come is $ev \cup c_0 \cup c_1 \cup d_0 \cup d_1$. So there are no further differences in the following calculations of both cases. Again, this move of the messages in the streams does not change the abstract representation of the streams.

[6]If this is not the case, the input streams should be rearranged in a way as it was done for eliminating the rollbacks.

SP$_i$ED		k	k+1	k+2	
C_{iN}	$\vdots$	c_0	c_1	c_2	$\vdots$
C_{ii}	$\vdots$	d_0	$c_0 \cup d_0$	d_1	$\vdots$
C_{Ni}	$\vdots$		-	o	$\vdots$
X	$\vdots$	gvt	$\surd$	$\surd$	$\vdots$
		s	s	s'	
Z^4	$\vdots$	ev	ev	$ev \cup c_0 \cup c_1 \cup d_0$	$\vdots$
		vt	vt	vt'	

C_{iN}	$\vdots$	c_0	c_1	c_2	$\vdots$
C_{ii}	$\vdots$	d_0	d_1	$c_1 \cup d_1$	$\vdots$
C_{Ni}	$\vdots$		o	-	$\vdots$
X	$\vdots$	$\surd$	gvt	$\surd$	$\vdots$
		s	s'	s'	
Z^4	$\vdots$	ev	$ev \cup c_0 \cup d_0$	$ev \cup c_0 \cup d_0$	$\vdots$
		vt	vt'	vt'	
		k	k+1	k+2	

Figure 10: Delay of a *gvt*-message

B.4 Proof of the Interaction-Refinement Property

Exemplarily only the IAR property for the relation O is shown here, i.e.

$$O[x^3; o^4] \wedge O[z^3; o^4] \Rightarrow x^3 = z^3$$

To show the equality of the two streams x^3 and z^3 the equality of the single elements of the streams is considered. These elements are sets of events. Assume the left-hand side of the implication, let t an arbitrary $t \in TIME$ and $e \in x.t$ with $e = [+, t_g, t_e, d]$ (Note that negative messages are not occurring in these streams). From $O[x^3; o^4]$ it follows $t = t_g + 2 \ \wedge \ \exists t' : e \in o^4.t'$. Since also $O[z^3; o^4]$ is assumed, $e \in z^3.(t_g + 2)$ can be concluded and therefore $e \in z^3.t$. As $e \in z^3.t$ implies $e \in x^3.t$ in a similar way as well, the equality is shown. $\square$

B.5 Proof of the Overall Refinement

The statement to be proved is

$$result(\text{CED}()) = result(\text{DED}())$$

that is reformulated by defining $o^1 = \text{CED}()$ and $o^4 = \text{DED}()$ and expanding the definition of $result$ as

$$\bigcup_{t \in TIME} o^1.t = \bigcup_{t \in TIME} o^4.t$$

Since

$$\text{CED} \overset{(\dagger; RT \succ DELAY \succ O)}{\rightsquigarrow} \text{DED}$$

is valid, the relation

$$(RT \succ DELAY \succ O)[o^1, o^4] \equiv \exists o^2, o^3.\ \ RT[o^1; o^2] \wedge DELAY[o^2; o^3] \wedge O[o^3, o^4]$$

holds. Assume e to be in the left hand side of the above equality. Note that e is a positive message of the form $[+, t_g, t_e, d]$, since negative messages are not generated in CED:

$e \in result(o^1)$	$\Rightarrow$	[Definition $result$, (4)]
$\exists t \in TIME.\ e \in o^1.t$	$\Rightarrow$	[Definition RT]
$\exists t' \in TIME.\ e \in o^2.t'$	$\Rightarrow$	[Definition $DELAY$]
$\exists t' \in TIME.\ e \in o^3.(t'+1)$	$\Rightarrow$	[Definition O (second half), e positiv]
$\exists t'' \in TIME.\ e \in o^4.t''$	$\Rightarrow$	[Definition $result$, (4)]
$e \in result(o^4)$		

Now assume $e \in result(o^4)$. By inspecting the specifications it gets obvious that e again must be a positive message: All events have been elements of the set $outQ$, and events are inserted there only by sim (producing only positive events) or deleted during a rollback by "adding" negative messages already contained in $outQ$.

$e \in result(o^4)$	$\Rightarrow$	[Definition $result$, (4)]
$\exists t \in TIME.\ e \in o^4.t$	$\Rightarrow$	[Definition O, e positiv]
$\exists t_g \in TIME.\ e \in o^3.(t_g+2)$	$\Rightarrow$	[Definition $DELAY$]
$\exists t_g \in TIME.\ e \in o^2.(t_g+1)$	$\Rightarrow$	[Definition RT]
$\exists t' \in TIME.\ e \in o^1.t'$	$\Rightarrow$	[Definition $result$, (4)]
$e \in result(o^1)$		

So both sides of the equality are proved. $\square$