Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Value Passing in MoDeST

submitted by

Arnd Hartmanns

on 2007-05-21

Advisor

Prof. Dr.-Ing. Holger Hermanns

Reviewers

Prof. Dr.-Ing. Holger Hermanns
Prof. Bernd Finkbeiner, Ph.D.

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, 2007-05-21

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, 2007-05-21

Abstract

MoDeST is a language to model stochastic timed systems. It provides a wide range of orthogonal features, but does not have direct support for passing values between concurrent processes. In the first part of this thesis, a semantics for value passing in MoDeST will be developed, with particular focus on the specific differences between MoDeST and existing value-passing modelling languages. In the second part, key steps of the implementation of these semantics for use with the Möbius simulation and performance evaluation framework will be highlighted, most notably concerning timed aspects of value passing.

Acknowledgements

This thesis has benefited from the support of several people, whom I would like to thank sincerely.

Holger Hermanns not only offered the topic initially, but has continually supported my work in every way. Our discussions on value passing, semantics, MoDeST, and implementation issues were not only of immediate value for this thesis, but also offered insights beyond.

Though asked on short notice, Bernd Finkbeiner immediately accepted to review this thesis, and the interest he showed was of great mental support during the final days of completing this text.

I further benefited from Reza Pulungan's unlimited patience when it came to resolving technical issues, and Christa Schäfer's immediate organisational support whenever it was needed.

Last but not least, Thomas Mike Peters deserves a special mention for proofreading and commenting on the thesis in an extremely constrained timeframe.

Table of Contents

1 Value Passing Semantics

1.1 MoDeST

MoDeST is a "modelling and description language for stochastic timed systems" [1]. It takes a modular, compositional approach to modelling reactive systems, in which the language's operators such as nondeterministic choice or parallel composition are used to combine small models into larger and more complex ones. MoDeST is suitable for a variety of applications, providing concepts known from object-oriented programming languages (e.g. exceptions and exception handling) and traditional modelling languages like Promela and LOTOS, and extending these with probabilistic branching and time.

Throughout most of the first part of this thesis, I will use a subset of the MoDeST language and its semantics, leaving out both probabilistic aspects and time; MoDeST as specified in [1] will be referred to as *full MoDeST*. In section 1.6, I will briefly show that my modifications to the semantics do not interfere with the omitted features and that the necessary modifications to the semantic rules presented before are simple.

1.1.1 MoDeST Semantics Overview

The semantics of MoDeST is defined in two steps: First, a MoDeST process is associated with a Stochastic Timed Automaton (STA), which can then be formally interpreted as a Timed Probabilistic Transition System (TPTS).

1.1.1.1 From MoDeST Processes to STA

An STA is similar to a labelled transition system, particularly when leaving out time and probabilities. In the non-timed non-stochastic subset of STA that we will use, states (or locations) correspond to MoDeST terms, while transitions (or edges) are labelled with an action, a guard, and an assignment function. The guard is a Boolean expression, possibly referencing some of the global or local variables, that determines whether an edge is enabled. An edge that is enabled can be taken: its action is "executed" and the assignments are performed atomically, possibly updating the variables and thus affecting the next transitions' guards.

It is important to note that an STA is a finite representation of a MoDeST process; the expressions in the guard and the assignment function are never evaluated, but kept symbolically when constructing the STA.

$$\frac{}{a \xrightarrow{a,tt,\emptyset} \surd}$$

$$\text{par}\{ :: P_1 \ldots :: P_n \} \stackrel{\text{def}}{=} \left(\ldots\left(P_1 ||_{B_1} P_2\right) \ldots\right) ||_{B_{n-1}} P_n$$
$$\text{with } B_j = \left(\cup_{i=1}^{j} \alpha(P_i)\right) \cap \alpha\left(P_{j+1}\right),\ \alpha(P) = \text{alphabet}(P) \setminus \{\tau, \lightning\}$$

$$\frac{P \xrightarrow{a,g,A} P' \wedge a \notin B}{P ||_B Q \xrightarrow{a,g,A} P' ||_B Q} \qquad \frac{P \xrightarrow{a,g_1,A_1} P' \wedge Q \xrightarrow{a,g_2,A_2} Q' \wedge a \in B}{P ||_B Q \xrightarrow{a,g_1 \wedge g_2,A_1 \cup A_2} P' ||_B Q'}$$

Figure 1: MoDeST operational semantics: Actions and parallel composition

Figure 1 lists the inference rules we are mainly concerned with: The basic axiom for actions that allows any action to complete, and the rules for parallel composition of processes. Because any number of processes can be combined in a `par` statement, they are first translated into a sequence of (binary) applications of the || operator, for which inference rules for synchronised and interleaved execution are given. See section 1.6 for a listing of these rules complete with time, probabilities, and value passing.

1.1.1.2 From STA to TPTS

The STA in its symbolic nature is too abstract for any reasoning that has to rely on the evaluations of variables. A concrete interpretation of a MoDeST process can be obtained by translating the STA into a TPTS, which is an infinite-state labelled transition system where time, probability distributions and the evaluations of the variables are made explicit.

Even after taking away time and probabilities, the TPTS we get may still have infinitely many states because of possibly infinite variable domains.

1.2 Motivation

In general, value passing in a modelling language is a way for concurrently executing processes to exchange data. Examples of communication that we may want to model include network protocols, interprocess communication on a single machine, or user interaction involving data input or output. Value passing is needed for these scenarios whenever we cannot completely abstract from the actual data that is being transferred, particularly if the correctness of the system depends on it.

Full MoDeST currently has no special constructs for value passing. However, there are global variables, and processes can synchronise by simultaneously executing a common action; they could thus exchange data through a global variable and manage the communication through clever use of synchronisation.

```
int val;
bool flag;

process SendCount() {
  int n;
  do { when(!flag) tau {= val=n, flag=true, n++ =} }
};

process ReceiveCount() {
  int m;
  do { when(flag) tau {= m=val, flag=false =} }
};

par { :: send :: receive }
```

Figure 2: Passing values with global variables

Figure 2 shows a trivial example for value passing with global variables – two processes where all nonnegative integers are transferred once, in ascending order, from a sender to a receiver. It is important to note that in this case, there is not even synchronization between the two processes, but instead, a Boolean variable is used to indicate when a new value is available or should be sent. The communication is therefore purely asynchronous, and even with synchronisation as described above, it is not possible to make it synchronous in terms of STA transitions: The value to be transmitted would have to be assigned to and read from `val` in one shot, which is not possible because of the atomicity of assignment execution.

There are additional complications with trying to simulate value passing this way as soon as one wants to implement broadcasting or longer asynchronous buffers; all in all, it is limited in expressivity and inelegant.

Our goal is thus to find a way of introducing explicit constructs for value passing that yield elegant and concise models while providing a maximum of expressivity. All the while, we want to preserve MoDeST's orthogonality of features, making value passing an aspect that can easily be omitted from or added to the language in the same way as time and probabilistic branching.

1.3 Approaches to Value Passing

There already exist a number of modelling languages that implement some form of value passing. It is instructive to inspect their approaches in order to avoid common pitfalls and be able to choose the best and most suited aspects for use in Value Passing MoDeST.

1.3.1 SPIN/Promela: Channels

The modelling language Promela, implemented in the SPIN model checker [9], uses channels for interprocess communication. Channels in Promela are typed buffers of some finite length. Values can be sent into a channel as long as it is not full, and can be received from a

channel as long as it is not empty. This means that communication via channels of positive length is always asynchronous, but synchronous communication can be achieved with channels of length zero. In this case, an attempt to send data will block as long as there is no other process attempting to receive from the same channel.

In addition to this basic communication scheme, Promela offers some more advanced operations on channels such as sorted send and receive, polling, and conditional receive. Conditional receive, probably the most interesting feature, allows the modeller to restrict the executability of a receive operation depending on the next value in the channel; in practice, this can be used to allow receiving only when a certain constant would be received.

All in all, value passing via channels is simple and intuitive, particularly in the context of modelling queues and network protocols. Still, the model of a one-to-one communication channel as in Promela does not allow broadcasting or any other form of multi-way communication.

1.3.2 LOTOS: Gates and Synchronisation

In its basic form without value passing, the ISO Specification Language LOTOS [4] provides synchronization between processes on common actions (the action name is called *gate*) in a similar way to MoDeST's parallel composition. Full LOTOS keeps this basic synchronization mechanism, but adds communication: An action is now a gate together with a list of values. There is a sense of direction for each of these values: We use the exclamation mark followed by an expression to indicate sending, and the question mark followed by a variable to indicate that we want to receive a value and store it in that variable. Intuitively, and for a single value, this direction can be interpreted just like sending to or receiving from a channel of length zero in Promela.

1.3.2.1 Synchronization and Communication

Using the same approach to synchronization as in MoDeST means that some design decisions have to be made when introducing a direction to the actions: When one process wants to send and others are ready to receive, obviously a value will be passed. What happens, however, when two processes want to send at the same time, or some processes are ready to receive, but there is no sender ready?

In the former case, there are two evident approaches: The two sending processes do not synchronise, but send sequentially, or they synchronise if they are about to send the same value and block otherwise. For LOTOS, the second approach was chosen, with this kind of interaction being named *value matching*.

In the latter case, the receiving processes could either be blocked, waiting for a non-existent sender, or some arbitrary value (of the right type) could be generated, simulating an unknown sender that is not (yet) specified in the model. LOTOS again uses the second approach, called *value generation*. This makes it possible to model *open systems* where a part of the environment is not modelled, but may generate values from a specified domain in some nondeterministic manner.

Process A	Process B	Condition	Interaction	Effect
g $!E_1$	g $!E_2$	value(E_1) = value(E_2)	value matching	synchronization
g $!E$	g $?x$	value(E) $\in$ dom(x)	value passing	synchronization, x = value(E)
g $?x$	g $?y$	dom(x) = dom(y)	value generation	synchronization, $x = y = v$, $v \in$ dom(x)

Figure 3: Communication scenarios in LOTOS

```
process Channel()
{
  int size, first, last;
  objtype c[capacity];
  do
  {
    :: when(size < capacity)
       in ?c[pos] {= last = (last + 1) % capacity, ++size =}
    :: when(size > 0)
       out !c[first] {= first = (first + 1) % capacity, --size =}  }
}
```

Figure 4: Simulating channels with synchronization on gates

An overview of the possible communication scenarios showing these decisions can be found in Figure 3.

1.3.2.2 Expressivity

Comparing LOTOS' synchronization on gates to channel-based communication as in Promela, the most obvious differences are that communication in LOTOS is always synchronised, but broadcasting is possible. If we were to use the LOTOS approach in MoDeST, we would prefer not to lose expressivity, so it is essential to show that this approach is at least as expressive as a channel-based one. Fortunately, assuming the existence of such an extension to MoDeST, it is easy to simulate communication channels as processes. See Figure 4 for an example, where a dedicated process acts as a buffer, storing its values in an array, and offering to receive or send values whenever the channel is not full or not empty, respectively.

1.3.2.3 Advanced Features

There is one advanced feature in LOTOS that is both important and useful: *Selection predicates*, similar to Promela's *conditional receive*. As the name implies, it allows guarding communication with an additional predicate that can reference the receiving variables. In this

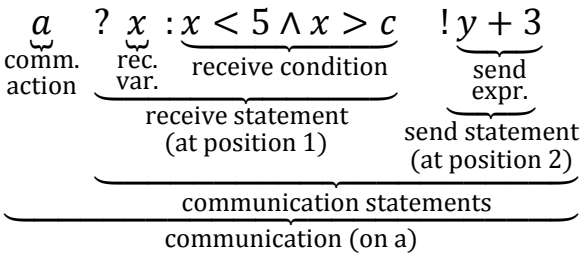


Figure 5: Term reference

way, receiving can be restricted to certain values depending on the state of the system. Because synchronisation is forced when it is possible – this depends only on the gate, not on the values – using selection predicates can lead to deadlock when there are senders trying to send values that a process does not accept. On the other hand, when there is no sender, a value that is acceptable to all participating processes will be generated. This kind of *value negotiation* is particularly powerful in models where some common initial parameters have to be agreed upon between different processes.

1.3.2.4 Comparison: FSP

FSP, short for *Finite State Processes* [12], is interesting for a comparison because it has a few simple mechanisms that allow the use of value passing in a very similar way to LOTOS, and there is a very useful implementation, LTSA[1], for simulation and verification.

FSP has no explicit support for value passing, but actions can be parameterised. The set of parameters for an action has to be finite, and the semantics treats every pair of action and concrete parameter as a single, distinct action. The FSP equivalent to sending in LOTOS is simply executing an action with a constant parameter, while a variable name can be used as a parameter to simulate receiving. Semantically, receiving will result in edges for all possible values to be generated in the transition system model, though most of them will hopefully be discarded during parallel composition. In fact, this is similar to LOTOS' semantics, where gates with a receiving term imply infinitely many possible actions; the restriction to finite domains in FSP makes a direct implementation possible.

In FSP, value passing and value generation occur under the same conditions as in LOTOS; only for value matching there is a difference: For two equal values, FSP processes will synchronise, but when sending different values, they will just send sequentially: The actions are different, so no synchronization is forced.

1.4 Value Passing MoDeST

We saw that LOTOS-style value passing is at least as expressive as channel-based value passing. It also fits well into MoDeST because, on the surface, it is simply an extension of the existing parallel composition. Therefore, LOTOS-style value passing will be the approach I am going to add to MoDeST.

[1] http://www.doc.ic.ac.uk/ltsa/

$$\frac{}{a\ !e \xrightarrow{a<\text{eval}(e)>,tt,\emptyset} \surd} \qquad \frac{v \in \text{dom}(x)}{a\ ?x \xrightarrow{a<v>,tt,\{x=v\}} \surd}$$

$$\frac{P \xrightarrow{a<v_1>,g_1,A_1} P' \wedge Q \xrightarrow{a<v_2>,g_2,A_2} Q' \wedge v_1 = v_2 \wedge a \in B}{P \;||_B\; Q \xrightarrow{a<v_1>,g_1 \wedge g_2,A_1 \cup A_2} P' \;||_B\; Q'}$$

Figure 6: Naïve value passing semantics

```
process Chan()
{
  𝔹 x;
  do { :: in ?x; out !x }
}
```

Figure 7: Fatal process for the naïve semantics

I will first present the semantics for the simplified case where there is only one value per action (or gate, if we were to strictly adhere to the LOTOS naming convention). I will then add conditional receive, and briefly discuss how to extend the semantics to a list of values at the end.

In order to avoid confusion when going into the details of value passing (or: communication), it is helpful to agree on a set of terms for the different aspects of communication. The terms used for the remainder of this thesis are shown in Figure 5.

1.4.1 The Naïve Approach

To translate value-passing MoDeST processes into STA, the axiom for actions and the inference rule for parallel composition will have to be modified. The naïve approach is to simply extend the actions with the values being transmitted, similar to the FSP semantics, but still distinguish between the action and the values.

The resulting semantics in Figure 6 evaluates the send expression for send statements, and creates all possible edges for receive statements. During parallel composition, we make sure that the two values of the synchronising actions match; this leads to value matching for two senders, and discards all unnecessary edges created by the receiving party during value passing. For value generation, no edges are discarded, so we get the expected result – every possible value can be generated.

One problem with this approach is already evident: For infinite value domains, which are not explicitly forbidden in MoDeST, value generation leads to infinitely many edges. Even if we decide to put up with this consequence, it leads to a fatal problem. Consider the example in Figure 7: In order to create an edge for `out !x`, we have to compute eval(`x`) – but we can-

not know the value for `x`; in fact, infinitely many different values are possible. We see that for value generation, which happens for `in ?x`, we would not only have to generate infinitely many edges, but infinitely many different destination states as well.

Thus, there are two problems with this naïve approach, one of them fatal: We generate infinitely many edges, which are undesirable considering that, up to this point, the STA has been finite, and we cannot perform the necessary evaluations at all.

1.4.2 A Finite Symbolic Semantics

What we need is a semantics that yields finite STA even for value passing processes. The key to finiteness for the STA for MoDeST was its symbolic nature: Assignments and conditions were noted as terms, and no evaluations were performed. If it is possible to make the semantics for Value Passing MoDeST finite, we expect to be able to do this by exploiting the symbolic capabilities of the STA model.

An example similar to the one from the previous section has been used by H. Lin as a motivation for *Symbolic Transition Graphs with Assignment* (STGA) in [11], refining the symbolic approach from [8]. While the title and the motivation of [11] promises an easy remedy for the naïve semantics' flaws – particularly since we already have a symbolic formalism with assignments, STA – there is one major problem: Lin works with a value-passing variant of CCS, which only has local identifiers in a functional sense that are introduced and bound by receiving statements during value passing. MoDeST's variables, on the other hand, are of an "imperative" style, i.e., they are declared at some point, are truly variable, and can be global. Because of this, the STGA semantics for value-passing CCS, which work with sets of free variables and term substitutions, cannot be directly transferred to MoDeST.

1.4.2.1 The Free Identifier

To solve these issues, the new symbol ? is introduced, which is treated like a variable most of the time, but cannot be used as such by the user, and is a special case both in the operational semantics and when translating STA to TPTS. I will refer to ? as the *free identifier*.

The free identifier is basically a variable without a value; its value is determined nondeterministically when executing actions. This allows us to think of assigning the free identifier to a variable as "freeing" that variable: it will have some arbitrary value after the assignment. If we choose to assign some value to the free identifier, this will not be persistent: The free identifier is local to a transition, so if it gets a concrete value, this will be lost after the execution of the transition.

This informal behaviour of the free identifier, which will be formalised in the rules for the transition from STA to TPTS, allows it to be used to identify the value being transmitted during value passing as long as that value is not known. The free identifier will therefore only survive parallel composition in the case of value generation.

$$\frac{}{a\ !e \xrightarrow{a\ !e,tt,\{?=e\}} \surd} \qquad \frac{}{a\ ?x \xrightarrow{a\ ?,tt,\{x=?\}} \surd}$$

$$\frac{P \xrightarrow{a\ c_1,g_1,A_1} P' \wedge Q \xrightarrow{a\ c_2,g_1,A_1} Q' \wedge a \in B}{P\ ||_B\ Q \xrightarrow{a\ c,g,A} P'\ ||_B\ Q'} \qquad c = \begin{cases} ? \text{ if } c_1 = ? \wedge c_2 = ? \\ !e \text{ if } c_1 = !e \vee c_2 = !e \end{cases}$$

$$g = g_1 \wedge g_2 \wedge (e_1 = e_2 \text{ if } c_1 = !e_1 \text{ and } c_2 = !e_2)$$

$$\begin{aligned} A = A_1 \cup A_2 \\ \cup\ (\{x = e | x = ? \in A_1 \wedge ? = e \in A_2\} \\ \setminus \{x = ? | x = ? \in A_1 \wedge ? = e \in A_2\}) \\ \cup\ (\{x = e | x = ? \in A_2 \wedge ? = e \in A_1\} \\ \setminus \{x = ? | x = ? \in A_2 \wedge ? = e \in A_1\}) \end{aligned}$$

Figure 8: Finite symbolic value passing semantics

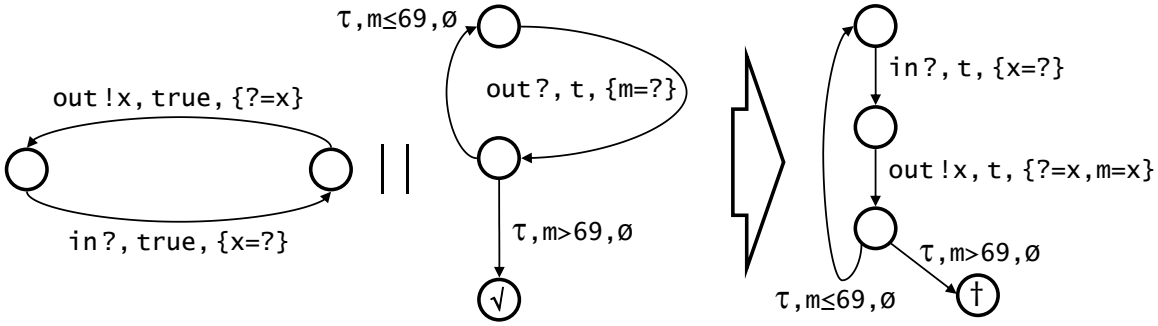


Figure 9: Example STAs for parallel composition with the finite symbolic semantics

1.4.2.2 The Semantic Rules in Detail

The semantic rules for a finite symbolic value passing semantics for MoDeST can be found in Figure 8. With the help of the free identifier, the axioms for the actions are simple: When sending, no variable is assigned a value in the current process, but the value being transmitted is known in terms of an expression, which we assign to the free identifier. Conversely, when receiving, we do not know the actual value that we may receive, so we assign the free identifier to the receiving variable.

For synchronisation during parallel composition, we first have to determine the complete action label of the resulting transition, i.e. we need to determine the value of c: If there is at least one sender, the value being sent should be available to outer processes, so sending dominates receiving. If both processes send, value matching has to occur, so we make sure

$$\frac{s \xrightarrow{a,g,A} s' \wedge w \in \mathrm{Val} \wedge (v \circ Q_w)(g) \text{ holds}}{\langle s, v\rangle \xrightarrow{a} \langle s', v \circ A \circ Q_w\rangle}$$

$$\text{where } Q_w \in \mathrm{Var} \to \mathrm{Val} \text{ s.t. } Q_w(x) = \begin{cases} x \text{ if } x \neq ? \\ w \text{ if } x = ? \end{cases}$$

Figure 10: From STA to TPTS

that the expressions being sent have the same value by adding this condition to the guard g. At this point, it is important to note that the free identifier cannot occur in the expressions being sent, because we did not allow it to be used as a variable by the user. Finally, we have to make sure that value passing works as expected, i.e. that receiving variables are assigned the concrete value being transmitted instead of the free identifier. This happens when computing the new assignment function A from A_1 and A_2.

Figure 9 shows an example where these rules are applied to the process *Chan* from Figure 7 in parallel composition with a simple server that receives from the channel and crashes when the received value is too large.

1.4.2.3 From STA to TPTS, with the Free Identifier

We formalise the behaviour of the free identifier by modifying the rules that create TPTS transitions from an STA; see Figure 10. In this modified rule, Q_w is an additional assignment function that maps the free identifier to some nondeterministically chosen value w from the correct domain.

1.4.3 Conditional Receive

At this point, adding conditional receive as conditions for receive statements that can reference the value that might be received is straightforward: For an action a that receives a value for variable x with condition b, all occurrences of x in b are replaced with the free identifier, and the resulting condition is put into the guard. In this case, we get the transition $a\ ?x{:}b \xrightarrow{a,b[?/x],\emptyset} \surd$, or in a concrete example, $a\ ?x{:}\ x \leq 5 \wedge x > y \xrightarrow{a,?\leq 5 \wedge ? > y,\emptyset} \surd$.

The TPTS semantics ensures that a value for the free identifier is chosen such that the guard condition holds. Without conditional receive, the relevant condition had no effect because the free identifier could not occur in the guard, but now, it forbids the generation of those values that do not satisfy the receive condition.

1.4.4 Passing Multiple Values

LOTOS supports passing a list of values on every gate, allowing any combination of receive and send statements. To support this in our finite symbolic value passing semantics, we have to distinguish between the positions in the list of values in order to be able to generate different values for different positions. The straightforward solution is to allow not only one, but instead many free identifiers $?_1 \ldots ?_k$. The rules and conditions that ensure the correct

```
process IChannel () {
  relabel {ina, inb, outa, outb} by {in, in, out, out}
    hide {syncb, synca}
    par {
      :: ChannelA()
      :: syncb
    }
}

process ChannelA() {
  type x; ina ?x;
  par {
    :: relabel {ina, outa} by {inb, outb} hide {syncb} ChannelB()
    :: syncb; outa !x; synca
  }
}

process ChannelB() {
  type x; inb ?x;
  par {
    :: relabel {inb, outb} by {ina, outa} hide {synca} ChannelA()
    :: synca; outb !x; syncb
  }
}
```

Figure 11: The infinite channel

operation of value passing and value matching now have to work with a list of values, but no fundamental changes have to be made. For the explicit semantics, please refer to section 1.6.

The only design decision here is what to do in case of an arity mismatch, i.e. when two processes have to synchronise on one action, but one transmits a different number of values than the other. Both for the semantics and for the implementation, I have chosen to not add any additional condition to the parallel composition semantics, but simply treat missing values as if the process would receive anything (intuitively, behave as `?x` for the right type).

1.5 Examples

Although there is evidence that the semantics we chose and adapted is sufficiently powerful and practically useable (after all, we are very close to the semantics of LOTOS, an ISO standard), I would like to support this with two examples:

1.5.1 The Infinite Channel

In section 1.3.2.2, we have already seen that (finite) communication channels as known from other modelling languages can be simulated with a LOTOS-style semantics. In Value Passing MoDeST, we can even model a channel that can take any number of messages and thus have infinite capacity. In fact, we do not even have to rely on variables with infinite do-

main, but just use recursion in parallel composition, complemented by action set manipulations (`relabel` and `hide`) to make it work. Being allowed to use full (and even nested) recursion, though, is essential for our construction, which is shown in Figure 11.

The basic idea of the construction is to use one process instance for every message in the channel. These process instances will be able to receive a message, and will then create a new inner instance that is able to receive another message, but is blocked from sending its value until after the outer instance has sent its own value by means of synchronisation requirements.

The `IChannel` process serves as a wrapper construction, hiding internal actions and streamlining the interface. Processes `ChannelA` and `ChannelB` are used in alternation to form the channel construction described above. They are almost identical in behaviour – except for a permutation of action names. We need two processes in order to allow an instance of one of the two to communicate only with its direct outer and inner processes, but not further up or down the instantiation chain. This is realised by hiding the right synchronisation actions and relabelling the communication actions, effectively "joining" the inner actions with the current processes' ones.

1.5.2 SCSI-2 Bus Arbitration

In [7], H. Garavel and H. Hermanns have shown how to extend a LOTOS model – used for the functional model-checking of the SCSI-2 (*Small Computer Systems Interface*) standard's bus arbitration protocol – to incorporate the necessary information for the evaluation of performance issues. The original model relies heavily on LOTOS' communication and value negotiation features, which makes translating this model into MoDeST a suitable and relevant demonstration of the usefulness of value passing in MoDeST.

Garavel's and Hermanns' extension mainly consisted of adding delays simulating system load, bus delay and disk service time to the model. These were added only in an abstract manner to the LOTOS specification, which was translated into a labelled transition system and then an Interactive Markov Chain (IMC), thereby instantiating the abstract delays. Converting IMCs to MoDeST specifications – and correctly modelling Markov delays – is a generally nontrivial problem that is not the focus of this thesis, which is why I will rely on an intuitive understanding of the delays in the system at hand to obtain a descriptive model. The emphasis is on the elegant use of the new communication features in MoDeST.

1.5.2.1 Translating the Model

The LOTOS model has two fundamental processes: disk and controller. Several disks and a controller (and possibly another process representing properties of the bus) will be composed in parallel, communicating on gates `ARB`, `CMD` and `REC`, representing bus arbitration, commands from the controller and results from the disks, respectively. The essential communication, namely the negotiation of exclusive bus access, occurs on gate `ARB`. Every disk and the controller have a unique SCSI number $id \in \{0, \ldots, 7\}$, and signal a request for bus access on wire id of the eight electrical wires responsible for arbitration, which can be read by all devices in the system. The device with the highest SCSI number requesting access

```
process DISK [ARB, CMD, REC, MU] (N:NUM, L:NAT, READY:BOOL):noexit :=
  CMD !N;
    DISK [ARB, CMD, REC, MU] (N, L+1, READY)
  []
  ARB ?W:WIRE [not (READY) and C_PASS (W, N)];
    DISK [ARB, CMD, REC, MU] (N, L, READY)
  []
  [not (READY) and (L > 0)] ->
    MU !N; (* Markov delay *)
      DISK [ARB, CMD, REC, MU] (N, L-1, true)
  []
  ARB ?W:WIRE [READY and C_LOSS (W, N)];
    DISK [ARB, CMD, REC, MU] (N, L, READY)
  []
  ARB ?W:WIRE [READY and C_WIN (W, N)];
    REC !N;
      DISK [ARB, CMD, REC, MU] (N, L, false)
```

Figure 12: The SCSI disk process in LOTOS

"wins" arbitration and has exclusive bus access for the duration of one command or result. To model arbitration in LOTOS, an eight-tuple of booleans represents the wires' voltage levels, and the constraints $\texttt{PASS}(W,n) \coloneqq \neg w_n$, $\texttt{WIN}(W,n) \coloneqq w_n \wedge \neg \bigvee_{i=n+1}^{7} w_i$ and $\texttt{LOSS}(W,n) \coloneqq w_n \wedge \bigvee_{i=n+1}^{7} w_i$ are applied to the communication by the devices, thus negotiating an acceptable solution (i.e., a correct winner).

Figure 12 shows the disk process in the LOTOS model. To translate this process into MoDeST, we will first replace the top-level choice and the tail recursion with a MoDeST construct combining both: `do`. We then have to decide how to concretely model the abstract Markov delays; this we can do at all because MoDeST has clock variables, so we can model the passing of time, and because MoDeST allows sampling probability distributions, which we can use to model delays of probabilistically determined length.

The direct solution – an in-place substitution of `MU !N` by `{= c=0, d=Exponential(rate_mu) =}; when(c>=d) tau` – does not work because this causes a disk to block the entire bus while intuitively processing some request, which is neither conceivable in a real system nor the behaviour corresponding to the LOTOS model. The problem with this approach is that the choice containing the translated delay would not be atomic, effectively disabling all other possible behaviours while waiting for the end of the delay. We resolve this by factoring out the delay choice into a process parallel to the remaining choices, as shown in Figure 13. While this is probably not an absolutely correct translation of a Markov delay, it matches our intuition of a hard disk being divided into one part communicating with the bus, and another part actually performing disk access, but not blocking the bus' communication.

The final obvious difference between the LOTOS and the MoDeST model is the inclusion of vacuous receive statements for the actions `cmd` and `rec` for values other than the current disk's SCSI number. These are needed for the parallel composition of the disks and the controller to work correctly: In the LOTOS model, the disks were composed with synchronisation limited to gate ARB. In MoDeST, there is no direct way to limit synchronisation without completely hiding the unwanted actions, so the statements implement a form of input-enabledness on `cmd` and `rec` for the disks; otherwise, a disk would be prevented from executing e.g. `rec !n` because there are no matching partners.

```
process Disk(int n) {
  int l, _; float d; bool ready; wire w; clock c;

  par {
    :: do {
         :: when(!ready && l>0) {= c=0, d=Exponential(rate_mu) =};
            when(c>=d) {= l=l-1, ready=true =} }
    :: do {
         :: cmd !n; {= l=l+1 =}
         :: when(!ready) arb ?w:pass(w, n)
         :: when(ready)  arb ?w:loss(w, n)
         :: when(ready)  arb ?w:win(w, n);
            rec !n;
            {= ready=false =}
         :: cmd ?_:_<>n
         :: rec ?_:_<>n }
  }
}
```

Figure 13: The SCSI disk process

```
process Controller(int nc) {
  int pending = nc, n; int t[nd]; float d; wire w; clock c;

  par {
    :: do {
         :: when(pending==nc)
            tau ?n:(n>=0 && n<nd && t[n]<8 && n!=nc);
            {= c=0, d=Exponential(rate_mu) =};
            when(c>=d) {= pending=n =} }
    :: do {
         :: when(pending==nc) arb ?w:pass(w, nc)
         :: when(pending!=nc) arb ?w:loss(w, nc)
         :: when(pending!=nc) arb ?W:win(w, nc);
            cmd !pending;
            {= pending=nc, t[pending]=t[pending]+1 =}
         :: rec ?n:(n<>nc && n>=0 && n<nd); {= t[n]=t[n]-1 =} }
  }
}
```

Figure 14: The SCSI controller process

For completeness, a translation of the controller process obtained with the same method is shown in Figure 14. An actual simulation and performance evaluation using the MoDeST model is described in section 2.6.1.

1.6 Full Operational Semantics

In section 1.1, I stated that the value passing semantics I have introduced would not interfere with the probabilistic and timed aspects of full MoDeST, and that the necessary changes would be simple. Therefore, I will now present the modified semantic rules for full MoDeST with value passing; most of the changes compared to the rules from the previous sections result from the fact that the destination of a transition is now a probability distribution over states and assignment functions.

1.6.1 Grammar

$$P := \ldots \mid \text{act } c_1 \ldots c_k \text{ where } c_i := !e \mid ?x\colon b,\ e \in \text{Exp},\ x \in \text{Var},\ b \in \text{Bxp}$$

1.6.2 Alphabet and Assignments

$$\alpha(\text{act } c_1 \ldots c_k) = \alpha(\text{act})$$

$$\text{A}(\text{act } c_1 \ldots c_k) = \{x = ?_i | c_i = ?x\colon b\} \cup \{?_i = \text{e} | c_i = !e\}$$

1.6.3 Axiom

$$\underbrace{act\ c_1 \ldots c_k}_{P} \xrightarrow{\text{act } c_1' \ldots c_k',\ r,\ \text{false}} \text{D}\big(\text{A}(P), \surd\big)$$

$$\text{with } c_i = !e \Rightarrow c_i' = !e,\ c_i = ?x\colon b_i \Rightarrow c_i' = ? \text{ and } r = \bigwedge_{\substack{1 \le i \le k \\ c_i = ?x:b_i}} b_i[?_i/x]$$

1.6.4 Parallel Composition

$$\frac{P_1 \xrightarrow{\text{act } a_1 \ldots a_k, g_1, d_1} \mathcal{W}_1 \wedge P_2 \xrightarrow{\text{act } b_1 \ldots b_l, g_2, d_2} \mathcal{W}_2 \wedge \text{act} \in B \cap \text{PAct}}{P_1 \mid\mid_B P_2 \xrightarrow{\text{act } c_1 \ldots c_k, g, d_1 \wedge d_2} (\mathcal{W}_1 \times \mathcal{W}_2) \circ \text{M}_{\text{par}}^{-1}}$$

$$\frac{P_1 \xrightarrow{\text{act } a_1 \ldots a_k, g_1, d_1} \mathcal{W}_1 \wedge P_2 \xrightarrow{\text{act } b_1 \ldots b_l, g_2, d_2} \mathcal{W}_2 \wedge \text{act} \in B \cap \text{IAct}}{P_1 \mid\mid_B P_2 \xrightarrow{\text{act } c_1 \ldots c_k, g, d_1 \vee d_2} (\mathcal{W}_1 \times \mathcal{W}_2) \circ \text{M}_{\text{par}}^{-1}}$$

$$\text{where } a_i = !v \vee b_i = !w \Rightarrow c_i = !v,\ a_i = b_i = ? \Rightarrow c_i = ?$$

$$\text{and } g = g_1[e/?_i \text{ if } a_i = !e] \wedge g_2[e/?_i \text{ if } b_i = !e] \wedge \bigwedge_{\substack{1 \le i \le k \\ a_i = !e_1 \\ b_i = !e_2}} e_1 = e_2$$

$$\text{M}_{\text{par}}(\langle A_1, P_1' \rangle, \langle A_2, P_2' \rangle) \stackrel{\text{def}}{=} \begin{cases} \langle \emptyset, \text{throw inconsistent} \rangle \text{ if } A_\cup \text{ is not a function} \\ \langle A_\cup, P_1' \mid\mid_B P_2' \rangle \text{ otherwise} \end{cases}$$

$$\text{where } A_\cup = A_1 \cup A_2 \cup \{x = e | x = ?_i \in A_1 \wedge ?_i = e \in A_2\} \cup \{x = e | x = ?_i \in A_2 \wedge ?_i = e \in A_1\}$$
$$\setminus \{x = ?_i | x = ?_i \in A_1 \wedge ?_i = e \in A_2\} \setminus \{x = ?_i | x = ?_i \in A_2 \wedge ?_i = e \in A_1\}$$

2 Implementation

2.1 MoTor

The actual modelling and analysis of MoDeST specifications is supported by MoTor[1], the *MoDeST Tool Environment*, developed mainly at the University of Twente, Netherlands. The vast expressiveness of the MoDeST language, which covers many different more limited models, makes MoDeST specifications in general undecidable. The fact that a generic algorithm for their analysis is thus impossible played a crucial role in the design of MoTor. It was not intended to offer a single general tool, but instead support the analysis of (analysable) submodels with specialised algorithms or existing tools.

The basic design of and the philosophy behind MoTor are well-explained in [2] and [10]. The former presents some details of the implementation; these sections are now somewhat outdated and should only be read as a very general overview of the tool's architecture and some of the underlying ideas. I will summarise the design and the current state of MoTor:

MoTor consists of three main parts. The first is the parsing engine that translates textual MoDeST code into an *abstract syntax tree* (AST). This tree is then used by the *first-state-next-state interface* (fsns), which in essence provides a stepwise exploration of the STA corresponding to a MoDeST specification, and the Möbius interface, where the AST is translated into C++ code that can be used to simulate a specification, including statistical evaluations of specified aspects, by the Möbius modelling environment.

MoDeST is nowadays used predominantly from within Möbius. There seems to be a consensus among those who specified MoDeST and developed MoTor that the fsns is deprecated, and it is not actively developed. A comparable tool that could replace it was recently developed by Christophe Boutter as part of his work on an Eclipse plugin for MoDeST [5]. Due to these ongoing developments, I have not done any work on the fsns, but implementing value passing in the Eclipse plugin as well is expected to be a useful future project.

2.1.1 Möbius

Möbius is a modelling and performance evaluation tool for complex systems, developed at the University of Illinois at Urbana-Champaign, USA. The systems can be specified in one of several available formalisms. This ability to choose between different formalisms and even combine models specified in different ones is one of the unique characteristics of this toolset.

To achieve this flexibility, Möbius has an abstract notion of a *model* that allows the core system to be extended with concrete models of different formalisms, as long as they can be expressed in such a way as to conform to Möbius' abstract process definition. This character-

[1] http://fmt.cs.utwente.nl/tools/motor/

ises a process as a collection of *state variables* containing the information of the model in a particular state, and *actions* that can change this state. State changes can happen over time, and a convenient way of sampling different probability distributions is also included, such that timed and stochastic models can be represented naturally. To evaluate – or measure – properties of interest, so-called *reward variables* can also be present.

2.1.1.1 Nondeterminism

As such, MoDeST specifications seem to have a natural translation to the Möbius' interface. There is, however, a single concept that is not present in Möbius – nondeterminism [3, section 2.3]. Möbius evaluates models from an entirely stochastic perspective, where nondeterminism does not fit in at all. In MoDeST, on the other hand, nondeterminism is a key language concept, which is at the heart of all composition mechanisms (think of parallel composition in particular).

The choice of the MoTor designers was to circumvent this limitation by relying on "Bernoulli's principle of insufficient reason, which states that all events over a sample space should have the same probability unless there is evidence to the contrary" [3, section 3.2], meaning that whenever there is a nondeterministic choice in MoDeST, this is presented to Möbius as a probabilistic choice with uniform distribution over the possible alternatives. This can, however, have unforeseen consequences; for example, repeating a branch in an `alt` statement makes it twice as probable. While trivial repetitions can be avoided by careful modelling, small overlaps or partial repetitions are much harder to spot or correct. (The unification of `LOSS` and the second part of `PASS` in the model in section 2.6.1 would be an obvious example that is hard to resolve immediately.)

2.1.1.2 Extending Möbius

Implementation-wise, Möbius is separated into a specification layer implemented in Java and an execution layer written in C++, which can be accessed by the user through the Java GUI. Consequently, the *abstract functional interface* (AFI), which developers extending Möbius use to access the abstract Möbius model from their code, is a collection of C++ base classes to be derived from by the concrete models.

A concise introduction to the Möbius framework is presented in [6], while lots of additional papers can be found on the project's website[1].

2.1.2 Limitations

When used to incorporate MoDeST specifications for simulation into Möbius, MoTor imposes some limitations, both on the aspects of the MoDeST language that can be used and on the actual semantics that will be in effect during simulation. I will give an overview of the most important ones – those who affect the implementation of value passing – without intending this to be a complete list:

[1] http://www.mobius.uiuc.edu/papers.html

The most severe restriction on the usable language features is on recursion: MoTor allows only a very simple form of tail recursion; nested recursion, for example, is not possible at all. This is a fundamental limitation in the way process instantiation is handled internally.

Another restriction of the usable features concerns the use of clock variables in guards and assignments. Clock variables themselves can only be reset to zero, and guard statements may only contain one single reference to a clock variable. In a sense, this brings the potential of clock use in MoDeST close to that in timed automata. Not surprisingly, this limitation greatly simplifies working with delays; we will notice this when implementing conditional receive.

The main semantic limitation of MoTor regards delays and the passing of time. Basically, there is no unnecessary progress of time (*as-soon-as-possible* semantics); if an action becomes enabled only after a certain amount of time, it will then be taken without any additional delay (if it is taken at all). Further delays not specified in a single guard will thus only occur if enforced by another guard, e.g. during parallel synchronisation.

Another semantic limitation – or: deviation – is that assignments are not executed atomically. Whereas the assignment block `{= x=y, y=x =}` should result in an in-place swapping of `x` and `y` according to the MoDeST semantics in [1], it will result in `x` and `y` having the same value in MoTor. To make matters worse, the exact order of execution for assignments on parallel actions in synchronisation is not specified, but fixed. Because value passing is essentially an assignment as well, I will correct the behaviour of the assignments before starting with the implementation of communication in section 2.4.2.

Some additional limitations, like the lack of full support for `structs` or arrays of clocks, arise from the current state of implementation and we can hope for these to be lifted in the future with ongoing development of MoTor.

2.2 Design Decisions

Several steps during the implementation required decisions that would influence the expressiveness of the actual value passing implementation in comparison to the semantics definition in part 1; some other decisions yet did not affect the expressive power of the implementation, but possibly its runtime performance.

2.2.1 Communication and Probabilistic Branching

The first necessary decision was how to represent communicating actions in the AST. The grammar given in section 1.6.1 already does not allow the combination of communication and probabilistic branching (`palt`) in a single statement: When combined with the full MoDeST grammar as found in [1], we are allowed to either have a single action `a`, to have probabilistic branching `a palt` ..., or to have communication. The straightforward approach of treating communicating actions the same way as actions leading to a `palt` statement, without worrying about combinations of the two (see section 2.4.1), is therefore valid.

```
int x = 3;
a ?x palt {
  :(x/4}: {= y=3*x =} b
  :(2*x/4}: {= y=23*z+5*x =} d
}
```

Figure 15: Combining communication, branching, and assignments

In particular, this is straightforward because communication is essentially a combination of guard and assignment, and assignments are usually only handled by `palt` (thus `{= x=5 =}` implicitly becomes `tau palt { :1: {= x=5 =} }`). Since we cannot simply *encode* communication into `palt`, we implement it *similarly*, which is how the distinction between the two cases – `palt` or communication – is established implementation-wise.

Not being able to combine communication and explicit assignments may sometimes be unhandy, but does not remove any expressivity; we can, if we really need it, combine the assignment and the communication in `par`.

Then again, having the possibility to combine both could result in either misleading syntax or semantic difficulties. Consider the MoDeST code in Figure 15, which combines communication and `palt`. This should be considered misleading if communication, probabilistic branching, and assignments are *performed* atomically (as we expect), because it is actually *written* as assignment-branching-assignment. On the other hand, making the branching depend on the first assignment would break the complete existing semantics, as we would have to allow some kind of redividing edge in the STA (cf. section 1.1.1).

2.2.2 Expressive Power of Receive Conditions

Conditional receive, on the other hand, is very loosely specified; the grammar in section 1.6.1 allows any Boolean expression as a receive condition, where any occurrence of the receiving variable will refer to the new value being received. Clearly, this is much too powerful to have any chance of being implemented efficiently. For integer variables and only small restrictions on the expressions, this could be seen as a constraint solving problem; while these are still NP-hard in general, convenient frameworks[1] do exists that can speed up the solving somewhat in many cases. Still, solutions will not only be needed when an action is actually taken, but already when deciding whether an action is enabled, so we would have to compute them very often. For floating-point variables, not even constraint programming will help.

To be able to find solutions with small computational effort while keeping a reasonable level of expressiveness for the conditions, I chose to restrict receive conditions such that every condition will constrain the corresponding value to a single interval in its domain. As clocks will cause additional complications (see section 2.3 for details), they can only occur once – that is as the first additive operand of the condition. Only having to deal with single intervals

[1] One efficient framework with a usable C++ interface is Gecode, the generic constraint development environment, a joint development of Kungliga Tekniska högskolan, Sweden, and Saarland University. See http://www.gecode.org/.

B	≔ x *op* clock_expr
clock_expr	≔ clockfree_expr
	| *c addop* clockfree_expr

where $op \in \{<, >, \leq, \geq\}$, $addop \in \{+, -\}$, c is a clock variable, and clockfree_expr any expression without reference to clock variables

Figure 16: Receive conditions for ?x:B

removes much of the computational complexity; we do not even have to slow down simulations with memory allocation and deallocation since we do not allow multiple intervals. The resulting grammar for receive conditions is shown in Figure 16.

The most obvious capability we lose in receive conditions is the specification of disjunctive or inequality constraints. Still, the benefit of extremely fast computation with the memory footprint known at compile time should be worth having to go through an extra step for disjunctions: We do not lose the ability to express them, but just have to do so by offering parallel communication alternatives.

2.2.3 Technical Issues

A small technical issue is that communicating actions are limited to a fixed number of communication statements `?x:b` or `!x`. The current value is 16, but this can be raised by changing a constant and recompiling. The reason for this is that precomputing the number of communication statements for every possible synchronisation of communicating actions at compile-time is infeasible, and I would, again, rather avoid dynamic memory allocation and deallocation at runtime for performance reasons.

Another issue that has technical and semantic reasons is that communication is limited to primitive data types – `int`, `float` and `bool`. While passing around complex data types could sometimes be helpful, giving a proper meaning to value generation for data types that might include structural information and dependencies would be difficult, and implementing this in a generic fashion is impossible since one cannot predict the semantics of, say, some user-defined `struct`.

2.2.4 Computing Synchronisation

Finally, a decision had to be made that would not affect the expressiveness of the implemented language fragment, but could have possible benefits or disadvantages during compile- or runtime: When should synchronising actions be combined in terms of their communication semantics, i.e. passing, matching, and generation of values?

Performing all the necessary computations regarding conditions for enabledness and effects on variables in the MoDeST compiler and emitting highly specialised code for every combination of actions would most likely give higher performance at runtime, but is not only difficult to implement: There have already been issues with the MoDeST compiler generating too much code that would then take far too long to actually compile in the past – this is

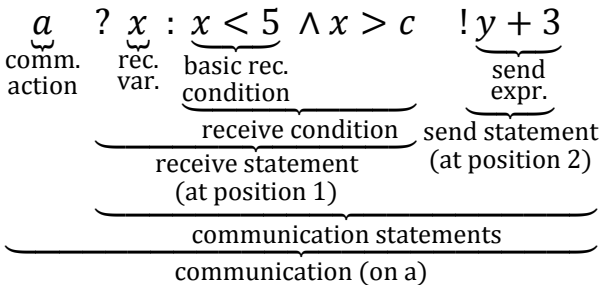


Figure 17: Updated term reference

why the tree structure of the model is not built into the generated code, but only written as XML and then re-read by the simulation binary.

If we had n parallel processes with m occurrences of the common action a each, we would have to generate specialised communication code for all m^n possible combinations, whereas in the current implementation, merely one `ModestSyncAction` devoid of any specialisations is instantiated for every combination. It is to be expected that this compile-time approach would also increase compile times to unacceptable levels.

On the other hand, restricting the generation of specialised code to single occurrences of actions and letting the synchronisation take place at runtime results in only $m \cdot n$ instances of communication-related code to be compiled. Because we have made sure that the necessary runtime computations – mostly w.r.t. conditional receive – are cheap, we will trade off acceptable compile times at the expense of a slight loss in runtime performance.

2.3 Timed Conditional Receive

This section will rely heavily on the use of the terms introduced in section 1.4. For quick reference, a slightly updated version of the term overview is included in Figure 17.

Without clocks, conditional receive is easy: For synchronising actions $a_1 \ldots a_k$ with communication statements $a_{i_1} \ldots a_{i_s}$ each, we have to determine the lower and the upper bound of the interval I_{i_j} of values that the receive condition of a_{i_j} admits; for sending, these bounds are equal, and for conditional receive, they can immediately be read from the conditions. The actual values that can then possibly be used for $?_j$ are those in $\bigcap_{i=1}^{k} I_{i_j}$.

As soon as basic receive conditions contain clocks, however, the situation is much more complicated: When deciding whether a set of synchronizing communicating actions is enabled, it is no longer sufficient to just consider the current evaluation of all variables and answer “yes” or “no” due to the fact that some receive conditions may become true only after a certain amount of time; yet during this time, other conditions containing clock variables may become false. We thus do not only have to calculate the possible values for $?_j$, but also the interval of time during which the synchronised communication can occur. As time progress is “global”, a timed condition in one communication statement always affects all other communication statements of one action as well.

	$?x$	$!e$		$?(x: x \geq e_1 \wedge x \leq e_2)$	
	x not clock	e clock-free	e with clock	e_1, e_2 clock-free	e_1, e_2 with clock
value constraint	$]-\infty, +\infty[$	$[\mathrm{v}(e), \mathrm{v}(e)]$	$]-\infty, +\infty[$	$[\mathrm{v}(e_1), \mathrm{v}(e_2)]$	$]-\infty, +\infty[$
clock constraint	$[-1, -1]$	$[-1, -1]$	$[\mathrm{v}(e), \mathrm{v}(e)]$	$[-1, -1]$	$[\mathrm{v}(e_1), \mathrm{v}(e_2)]$

Figure 18: Intervals induced by some communication statements

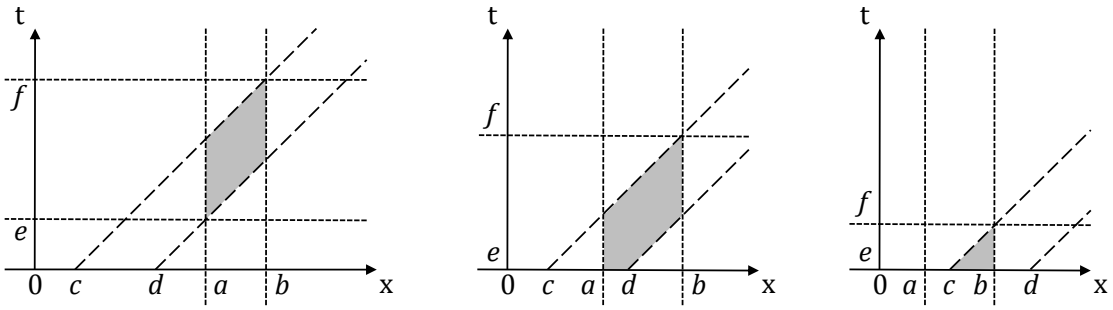


Figure 19: From constraints on ? to constraints on t

Worse yet, time does not only affect enabledness, but possibly also the resulting assignments: For example, `par{ :: a ?(x:x<=c) :: a !c }` will cause a delay of at least three time units (if clock variable `c` is zero initially), and assign to `x` the length of the delay. We therefore have a situation where conditions local to a communication statement have to be composed globally for all communication statements and then be reapplied to the local constraints concerning the values for ?. In detail, this works as follows:

In the **first step**, we obtain two intervals for every communication statement a_{i_j}. The first interval $I^1_{i_j}$ keeps track of the clock-free constraints on the values of $?_j$, while the second one, $I^2_{i_j}$, is determined by the current evaluation of those conditions containing clocks. As a concrete example, `a ?(x:x<=c+1 && x>=5)` results in $I^1_{1_1} = [5, \infty[$ and $I^2_{1_1} = [-1,3]$ if initially $\mathtt{c} = 2$. ("No bound" for clock constraints is represented as -1). Figure 18 lists the different intervals resulting from basic types of communication statements.

The **second step** is to intersect the intervals of all participating actions, i.e. $I^b_j = \bigcap_{i=1}^{k} I^b_{i_j}$. In the implementation, these first two steps are actually performed in one loop.

Up to this point, we have treated timed and clock-free conditions independently. Consequently, the current representation of clock constraints in I^2_j is just an interval specifying the values for $?_j$ licensed by the timed conditions *unless time elapses*. To be able to compute the *global clock constraint*, that is the interval of *time* during which the whole communication is

allowed to happen, we have to convert I_j^2, which constrains $?_j$, to an interval T_j constraining time (using I_j^1 as well).

This conversion is the **third step** in our transformation; it is illustrated in Figure 19: If $I_j^1 = [a, b]$ and $I_j^2 = [c, d]$, the allowed combinations of values for $?_j$ and time progress are shown in grey, so we want $T_j = [e, f]$. The actual calculations are now simple and can be inferred from the graphs; nevertheless, care has to be taken to accurately compute open- or closedness of the resulting interval bounds and to correctly cover all special (and degenerate) cases.

Once we have computed T_j for every communication position, we can finally determine the global clock constraint $T = \bigcap_{j=1}^{s} T_j$ as the **fourth** transformation **step**. Again, steps three and four are performed in a single loop in the actual implementation.

The final, **fifth step** is to reapply the global clock constraint to the individual constraints for $?_j$ on each position j, usually tightening the bounds given by the I_j^1. This is just the inverse of step three, so we can go back to Figure 19: This time, $T = [e, f]$ and $I_j^2 = [c, d]$ are given, and we want to obtain stricter values for $I_j^1 = [a, b]$. Again, the necessary calculations are obvious.

After completion of these five steps, every I_j^1 specifies the possible values for $?_j$ after time progress in the interval T. Because time advances linearly with gradient 1, this would still allow us to recompute the actual interval of possible $?_j$ after some fixed time progress in T, but for consistency with the rest of MoTor and its as-soon-as-possible semantics, no unnecessary time progress will occur in the implementation – the left bounds of T and I_j^1 are the values that will be chosen.

2.4 Implementation Details

Many basic cornerstones of the implementation have already been laid out in sections 2.2 and 2.3, though in a mostly abstract manner and most notably without detailing where the discussed aspects will later fit into MoTor.

This section is therefore intended as an overview of the changes that I implemented, with the emphasis being not so much on the actual code, but rather on how and where the changes fit into MoTor and on their interaction with each other and the existing components.

An informal schematic overview of the way a MoDeST specification is processed in Möbius is shown in Figure 20.

2.4.1 Parsing

MoTor's parsing stage is built using the ANTLR[1] parser generator, which substantially simplifies the parser specification. The ANTLR AST representation generated in the parsing

[1] "*ANother Tool for Language Recognition*", http://www.antlr.org/

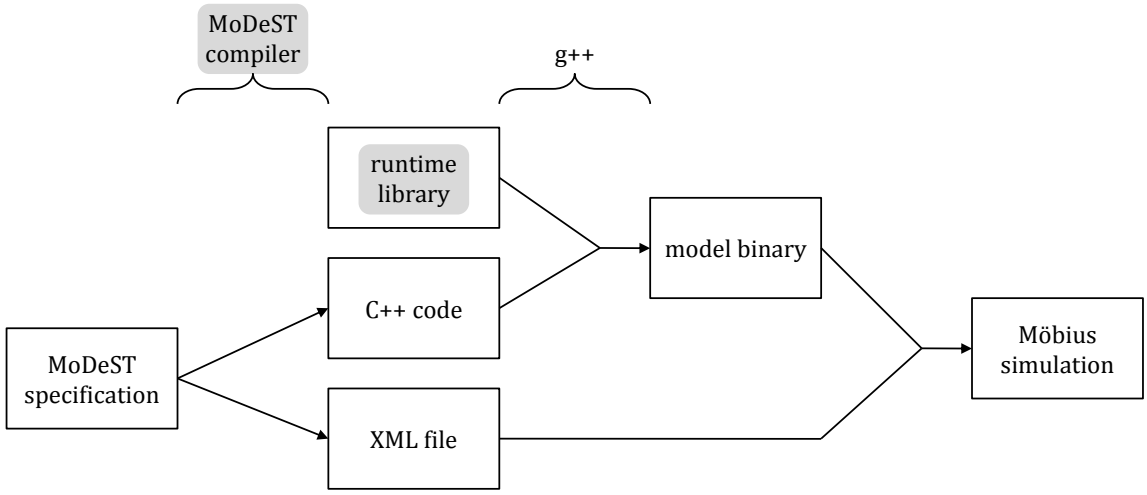
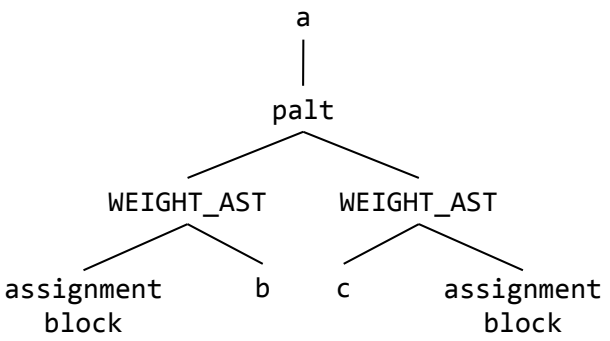


Figure 20: MoDeST processing in Möbius, main MoTor contributions highlighted

```
a palt { :5: {= x=3 =} b
       :4: {= y=5 =} c }
```




```
a ?(x:x<5 && x>3) !(y+13)
```

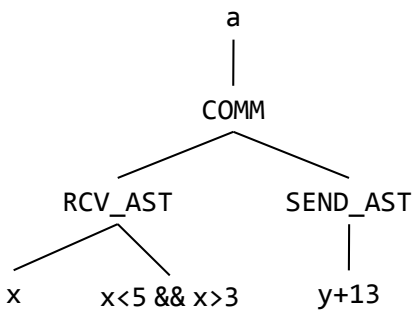


Figure 21: Simplified parser representations of `palt` *and communication*

stage is used throughout Möbius, however, so the use of ANTLR is not limited to the parser. Unfortunately, ANTLR also seems to introduce its own set of bugs, makes it hard to avoid duplicating code whenever one has to walk the trees it generates, and suffers from a lack of well-structured documentation. Still, except for issues with ANTLR, extending the parsing stage was straightforward:

The lexer[1], the first step in parsing, already recognises all the symbols necessary for communication; I have just added an imaginary token `COMM` for use in the parser.

In section 2.2.1, I argued that the internal representation of communication can be made very similar to that of `palt`. The parser[2] is where the foundation for this representation is laid when creating a communication AST in `perform_action`. Figure 21 compares the representation of `palt` and communication after running the parser.

[1] In file `parser/lexer.g`
[2] In file `parser/parser.g`

```
SyncAction: Fire
   Action 1: Fire
     x := 2*y
     z := 2*x
   Action 2: Fire
     y := x + z
```

```
SyncAction: Fire
   Action 1: Prepare
     lhs1 := x, rhs1 := 2*y
     lhs2 := z, rhs2 := 2*x
   Action 2: Prepare
     lhs3 := y, rhs3 := x + z
   Action 1: Assign
     lhs1->setValue(rhs1)
     lhs2->setValue(rhs2)
   Action 2: Assign
     lhs3-setValue(rhs3)
```

Figure 22: Old assignment behaviour (left) compared to the new assignment behaviour (right)

The third parsing step in MoTor is a tree walker[1] that walks the AST generated by the parser, transforms it, and generates non-ANTLR objects for MoDeST constructs. For communication, a new type of construct, the `CommConstruct`[2], is introduced. This is quite similar to, but much simpler than, the `PaltConstruct` as well.

2.4.2 Fixing the Assignments

As mentioned in section 2.1.2, MoTor does not execute assignments atomically. This not only rules out in-place swapping of variables and messes up recursive process calls (after the first recursion step in `process A(x, y) { A(y, x) }`, we have $x = y$), but is also bound to cause "interesting" effects when normal assignments are combined with communication, which is essentially just an inter-process assignment. Consequently, the assignments have to be fixed.

The assignments of one alternative in a `PaltConstruct` are translated into one class derived from `ModestAssignmentFunctor`[3] for use in the Möbius model. These had a single `operator()` that performed the assignment. The violation of atomicity originated in two places: First, the assignments within a single `operator()` were executed sequentially, so the first assignment could already modify values that later ones used. Second, the assignment functors of synchronizing actions with `palt`s were also called sequentially, leading to the same problem on yet another level.

In order to be able to execute a set of assignments atomically, we have to split an assignment into two steps, the second of these being the actual assignment of values to the specified variables. In the first step, we precompute the left- and right-hand sides of the assignments and store them for the second step. Because we need to resolve the problem at the level of synchronizing actions, the modified `ModestAssignmentFunctor`s have to expose both steps so that a `ModestSyncAction`[4] may call the first one for all participating actions before moving on to the second.

---

[1] In file `reptng/model_gen.g`
[2] In files `reptng/constructs.h`, `reptng/constructs.cpp`
[3] In file `backend/mobius/mobius-runtime-lib/ModestConstruct.h`
[4] In file `backend/mobius/mobius-runtime-lib/ModestSyncAction.h`

Figure 22 shows a pseudocode-based comparison of the old and the new assignment behaviour for two synchronising actions with assignments that were harmed by the old behaviour.

2.4.3 Communication

The parser now generates a suitable representation of actions with communication, and the assignments work as specified. At this point, we have to think about how to actually do the communication at runtime (i.e. when simulated by Möbius), and about how to generate the relevant model code.

2.4.3.1 Runtime

At simulation runtime, Möbius has access to all actions in the model, represented as `ModestActions`[1] when not synchronised, or as `ModestSyncActions` otherwise. It can query whether they are enabled (`Enabled()`), ask for the time needed to complete an action (`SampleDistribution()`) – which is, in MoDeST, the time until the action becomes enabled – and finally `Fire()` them.

For each of these functions, we have to perform the calculations outlined in section 2.3. An action (which may be a synchronised one) is then enabled if and only if none of the value constraint intervals is empty; the time needed for completion is the left bound of the global clock constraint; and on firing an action, values from the respective intervals have to be chosen and assigned.

Apart from smaller technicalities concerning the operation of these functions, the interesting point is how to actually collect all the information necessary for the computations, and where and when to perform them.

For this purpose, the `ModestCommunicationCollector`[2] was designed. It contains pairs of values of the data types relevant for communication (which represent the value constraint interval bounds), stores the local and global clock constraints, and keeps track of whether communication occurs at a certain position at all, and if so, of what data type.

Whenever the described computations have to be performed, a `ModestCommunicationCollector` is passed to an instance of a customised class derived `ModestCommunicationInfo`[3], which is associated with each participating action. These classes are generated by the MoDeST compiler and contain the specifics of the communication occurring at a certain action; when given a `ModestCommunicationCollector cc`, they will add their own constraints to the ones already in cc, implementing steps one and two of section 2.3.

Steps three, four and five are implemented in `ModestCommunicationCollector`'s method `transform()`, which causes the collector to "normalise" itself.

When an action is being fired, we also have to select and assign correct values. The `ModestCommunicationAssigner` class is central to this, essentially representing a frozen `Modest-`

[1] In file `backend/mobius/mobius-runtime-lib/ModestAction.h`
[2] In file `backend/mobius/mobius-runtime-lib/ModestConstruct.h`
[3] In file `backend/mobius/mobius-runtime-lib/ModestConstruct.h`

```
SyncAction: Fire
   CommunicationCollector cc()
   Action 1: Fire(cc)
      CommunicationInfo 1: match(cc)
   Action 2: Fire(cc)
      CommunicationInfo 2: match(cc)
   cc.transform()
   CommunicationAssigner ca(cc)
   Action 1: Assign(ca)
   Action 2: Assign(ca)
```

Figure 23: Executing communication

`CommunicationCollector` where all intervals are singletons. It can be created from a collector, thereby choosing the values – either according to a uniform distribution over the respective interval, or deterministically selecting the left bound when clocks are involved. It is then passed on to the assigners of the actions, which do nothing but assign the chosen values to the right variables.

Again, a pseudocode illustration is provided, showing the firing of a synchronised action containing two communicating ones, in Figure 23.

2.4.3.2 Generating the Code

The process described above makes use of one major abstract base class for which an inheriting, specialised subclass has to be generated for every action: `ModestCommunicationInfo`. Generating these subclasses is handled by `ProcessCommunication`[1] in the MoDeST compiler, which in turn is called by the `WriteModelVisitor`[2] when it encounters a `CommConstruct`.

As generating C++ code is rather involved technically, especially when one has to deal with an environment (ANTLR) that makes reusing portions of code without just copying them difficult, `ProcessCommunication` is relatively long without being particularly interesting, so I will refrain from a detailed explanation.

2.4.4 Supporting Bits and Pieces

To make all of the above actually work, lots of other, mostly mechanical extensions of the existing code base were necessary. For instance, the (model) code generation and the wiring of the generated classes to the constructs is actually implemented in a set of tightly intertwined “visitors” (implementing the Visitor design pattern and visiting AST nodes); most of these had nothing to do with communication, but they had to be extended to process `CommConstructs` nonetheless. The most interesting one of these might be the `DependencyVisitor` that generates the code telling Möbius which actions affect which variables and vice-versa; still, most of what was needed for communication could be adapted from the existing code for `palt` and guards.

[1] In file `backend/mobius/process_communication.g`
[2] In files `backend/mobius/write_model_visitor.h`, `backend/mobius/write_model_visitor.cpp`

2.5 Testing

Testing newly written code with a well-founded set of (unit) tests is imperative to ensure the success of any programming project. However, the integrated nature of the Möbius environment, where the definite results of a MoDeST specification – simulation results – are obtained by the invocation of a long series of components, involving parts such as the reward model which are essentially black boxes, makes automatically testing small portions of code infeasible. Even though the MoDeST compiler itself is a largely independent component, it only outputs C++ code that is then compiled and used by Möbius, so testing the MoDeST compiler itself would already mean having to anticipate the generated code in such detail that an automated procedure could verify the it.

During development, I therefore opted for manual sanity checks of generated code after every major advance of the compiler. This way, we can be pretty sure that the compiler *generally* works as *intended* once it is complete.

It then still remains to check whether the intended *behaviour* of the compiler actually yields the intended *results*, both in terms of correct translation and correct simulation results, and whether any non-obvious or boundary cases still lead to unintended behaviour or results.

This can then actually be done in a semiautomatic fashion: All or most relevant correctness properties of the compiler can be encoded in MoDeST specifications that lead to unexpected simulation results for errors in code generation or in the static runtime code. If we keep the specifications small, we can expect to find isolated erroneous aspects of the program and not have the result skewed by several different bugs occurring in one run.

As a simple example for this approach, let us look at testing the correction of the assignments (cf. test case 14 to test case 20): We first try to see whether assignments still work at all. This will not reveal any bugs related to atomicity issues, but highlight basic problems that would affect any assignment. Only then do we go on to include other constructs such as `alt`, guards, and clocks. Once we know that assignments still seem to work as they did before the changes, we can proceed to check the new behaviour, i.e. atomicity of the basic assignments in a list and of parallel assignments; see Figure 24 for one of the two relevant test cases. Everything seems to work correctly, so we can finally try some advanced uses – in this case, arrays and process instantiations. Arrays are often problematic because accessing an

In-place swapping of values is now possible:

```
a {= x=9.0, y=-1.0 =};
a {= x=y, y=x =};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	-1.0	-1,0000000000E00	✓
y	9.0	9,0000000000E00	✓

Figure 24: Test case 17

array is obtaining a single value through a complex expression, in contrast to simply using a single identifier; errors occurring only with process instantiation are usually issues with scope, local variables, and parameter passing (which is, in fact, just another assignment).

A list of all the MoDeST specifications used for testing, including the expected results and those obtained in a specified simulation setting, can be found in Appendix A. Several off-by-one errors, e.g. occurring only on value generation for float variables without bounds specified in conditional receive, were found with these test cases (and of course, subsequently fixed). As such, the test cases will of course not highlight any bugs still hidden in the current implementation (although I am confident that these do not even exist…), but they will be useful for future programmers working on MoTor, allowing them to ensure that their changes or extensions should not break the current communication functionality and behaviour.

2.6 Examples

In section 1.5, I presented two examples showing the usefulness of Value Passing MoDeST. For the first one, the Infinite Channel, there is no hope of actually using it in MoTor in that form due to the nested recursion. We should, however, be able to simulate the SCSI arbitration example in Möbius, and doing so will give us an insight into how restricting – or maybe just how inconvenient – the limitations of the implementation are. If we are lucky, the intuitive translation of the Markov delays will have been so good that simulation gives us results close to those obtained in [7].

2.6.1 Simulating the SCSI-2 Bus Arbitration

The MoDeST specification given as a translation of the LOTOS model for SCSI-2 bus arbitration in section 1.5.2 is still abstract considering the requirements of MoTor in primarily two ways; that is the actual representation of the `wire`, and the receive conditions, which are still way too powerful.

2.6.1.1 Adapting the Model

In order to represent the `wire`, we abstract away from the actual electrical wires one step further compared to the original model: The only relevant result is the number of the highest-numbered wire that is active during an arbitration period. If we thus use a single integer w as a representation of the `wire`, we can still express the three predicates used: $\texttt{PASS}(w,n) = (w \neq n), \texttt{WIN}(w,n) = (w = n), \texttt{LOSS}(w,n) = (w > n)$, and let value negotiation determine appropriate values. We do need to take care, however, that no value is generated that does not correspond to a device in the system.

Rewriting the receive conditions is now mostly straightforward: Disjunctions and inequalities have to be rewritten to separate alternatives, and the complex condition for the value generation on `tau` in the controller, which was merely a way to choose an appropriate disk nondeterministically, is replaced by an alternative for every disk. The MoTor-compliant

```
process Disk(int n) {
  int l, w, _; float d; bool ready = !true; clock c;

  par {
    :: do {
         :: when(!ready && l>0) {= c=0, d=Exponential(rate_mu) =};
            when(c>=d) {= l=l-1, ready=true =} }
    :: do {
         :: cmd !n; {= l=l+1 =}
         :: when(!ready) arb ?(w:w<n && w>=0) // pass 1/2
         ::              arb ?(w:w>n && w<nd) // pass 2/2, loss
         :: when(ready)  arb !n;              // win
            rec !n;
            {= ready=!true =}
         :: cmd ?(_:_>n && _<nd)
         :: cmd ?(_:_<n && _>=0)
         :: rec ?(_:_>n && _<nd)
         :: rec ?(_:_<n && _>=0) }
  }
}
```

Figure 25: The SCSI disk process for Möbius

```
process Controller(int nc) {
  int w, pending = nc, n; int t[nd]; float d; clock c;
  par {
    :: do {
         :: when(pending==nc && t[d1]<8)
            {= c=0, d=Exponential(rate_lambda) =};
            when(c>=d) {= pending=d1 =}
         :: ... (for every disk) ...
    :: do {
         :: when(pending==nc) arb ?(w:w<nc && w>=0) // pass 1/2
         ::                   arb ?(w:w>nc && w<nd) // pass 2/2, loss
         :: when(pending!=nc) arb !nc;              // win
            cmd !pending;
            {= pending=nc, t[pending]=t[pending]+1 =}
         :: rec ?(n:n<nc && n>=0); {= t[n]=t[n]-1 =};
            alt {
              :: when(n==d1) {= ctr1=ctr1+1 =};
                 {= tp1=((float)ctr1)/gc =}
              :: ... (for every disk) ... }
         :: rec ?(n:n>nc && n<nd); ...
  }
}
```

Figure 26: The SCSI controller process for Möbius

processes are shown in Figure 25 and Figure 26; `...` marks code in the controller that is little more than repetition.

```
extern const float rate_mu = 400, rate_lambda = 400, rate_nu = 400;
const int nd = 4; // actual number of devices
const int d1 = 0, d2 = 1, d3 = 2, ctrl = 3;
action arb, cmd, rec;
int ctr1, ctr2, ctr3;
float tp1, tp2, tp3;
clock gc;

process Disk(int n) { ... }
process Controller(int nc) { ... }

process Bus() {
  clock c; float d;

  do {
    :: when(c>=d) arb {= c=0, d=Exponential(rate_nu) =}
    :: rec
    :: cmd }
}

par {
  :: Controller(ctrl)
  :: Disk(d1)
  :: Disk(d2)
  :: Disk(d3)
  :: Bus() }
```

Figure 27: The complete SCSI model for Möbius

When "putting it all together", i.e. instantiating the processes and adding global declarations, we add two more new things: A process `Bus` that models the bus delay, and global variables for the throughput of the disks, which are updated whenever the controller receives a result. The complete model can be found in Figure 27.

2.6.1.2 Simulation Results

Our model of the SCSI-2 bus arbitration protocol for three disks can now be put into Möbius and serve as the foundation for a reward model, indicating that we would like to obtain the steady-state value of the throughput variables `tp1` through `tp3`, a study that will increment the load put on the controller (i.e. `rate_lambda`), and finally a solver that will perform the simulation.

For disks having SCSI numbers 0 to 2 and the controller having number 3, the result for the lowest and the highest priority disk are shown in Figure 28. While the result does not exactly coincide with the one in [7], the fundamental trend of relative starvation of the low priority disk with increasing load is clearly visible. For a model translation relying on intuition – and therefore without any intention of a formal proof of model equivalence – this should be quite satisfying; even more so when we consider that the as-soon-as-possible semantics implemented in Möbius might skew the result as well.

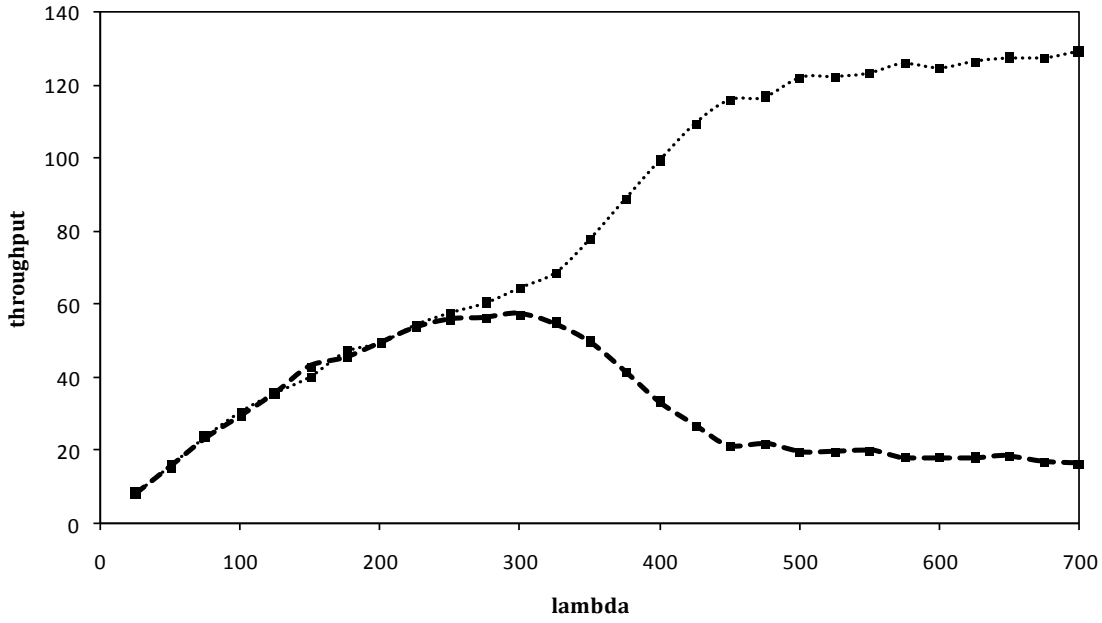


Figure 28: Throughput of high priority disk (dotted) and low priority disk (dashed) under increasing load

Conclusion

In the first part of this thesis, we have investigated some existing languages that already include value-passing. We have then chosen one of the most expressive approaches for use in MoDeST – LOTOS-style value passing, including value matching, value generation and value negotiation.

We then realised that giving a semantics for this approach that works well with MoDeST's notion of variables is actually far from trivial, and the resulting semantics obviously differs from other value-passing semantics, like the one introduced for CCS by H. Lin. Still, the introduction of a so-called *free identifier* ? allowed us to translate value-passing MoDeST to STA in a purely symbolic way, keeping the finiteness of the representation.

The second part of the thesis introduced the tools supporting simulation and evaluation of MoDeST specifications, MoTor and Möbius. We saw that some limitations on the previously developed semantics are necessary to obtain an efficient implementation.

Of particular interest was how to deal with clocks in conditional receive. In this setting, time deeply affects the behaviour of communication, and we saw a straightforward way of calculating the corresponding constraints.

The result of this thesis is thus an extension of the MoDeST semantics that introduces new semantic concepts to preserve finiteness, and a working implementation that allows simulation of value-passing MoDeST processes, including full support for timed aspects.

References

[1] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MoDeST: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.

[2] Henrik C. Bohnenkamp, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. The MoDeST modeling tool and its implementation. In Peter Kemper and William H. Sanders, editors, *Computer Performance Evaluation / TOOLS*, volume 2794 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2003.

[3] Henrik C. Bohnenkamp, Holger Hermanns, Ric Klaren, Angelika Mader, and Yaroslav S. Usenko. Synthesis and stochastic assessment of schedules for lacquer production. In *QEST*, pages 28–37. IEEE Computer Society, 2004.

[4] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[5] Christophe Boutter. An Eclipse plugin for MoDeST. Diploma thesis, May 2007. Advisor-Holger Hermanns.

[6] T. Courtney, D. Daly, S. Derisavi, V. Lam, and W. H. Sanders. The Möbius modeling environment. In *Tools of the 2003 Illinois International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems, Universität Dortmund Fachbereich Informatik research report no. 781/2003*, pages 34–37, 2003.

[7] Hubert Garavel and Holger Hermanns. On combining functional verification and performance evaluation using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer, 2002.

[8] Matthew Hennessy and Huimin Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[9] Gerard J. Holzmann. *The SPIN MODEL CHECKER*. Addison-Wesley Pearson Education, 2003.

[10] Joost-Pieter Katoen, Henrik C. Bohnenkamp, Ric Klaren, and Holger Hermanns. Embedded software analysis with MoTor. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 268–294. Springer, 2004.

[11] Huimin Lin. Symbolic transition graph with assignment. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 50–65, London, UK, 1996. Springer-Verlag.

[12] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

Appendix A: Test Cases and Results

All test cases were designed to use a (subset of a) common set of variables, so that we can use one reward model for all cases. All we have to do when going from one test case to another is thus to change the MoDeST code, but keep a common header with all variable declarations (Figure 29), and compile and run the simulation.

The reward model then measures the variables at time 10.0, so we have enough room to test timed effects. The solver we use is set to use the Lagged Fibonacci random number generator with seed 31465 on exactly 10000 batches. We are only really interested in the mean results, so the confidence intervals will not be reported. For all tests, Möbius version 2.0 was used.

Some of the results depend on probabilistic choices, i.e. the choice of actions in `alt` statements, explicit probability sampling, and value generation, and are marked with ≈. Since the Möbius simulator is instructed to run between 1000 and 10000 simulations, the results we expect are the appropriate mean values. Since the actual result depends on the random number generator's seed and the simulation batch size, they may not precisely match the mean we expect, but they should be close.

Syntax Checks

Value passing introduces new syntax: actions with communication. We first check whether the MoDeST compiler will generate errors on malformed communication statements, which can mainly be either obvious syntax errors or receive conditions that the implementation does not allow:

Test case 1
Sending the value of an expression requires brackets:

```
a !n==0?1:2
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

```
action a, b;
int f, n, m;
float x, y, z;
clock c, d;
bool s, t;
int ar[2];

// Do nothing, but at least refer to all variables so they
// can always be picked up by the reward model:
a {= f=0, n=0, m=0, x=0.0, y=0.0, z=c, s=t =};
```

Figure 29: Common variable declaration header

Test case 2
Actions cannot be “sent”:

```
a !b
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 3
We cannot “receive” into actions:

```
a ?b
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 4
We cannot receive into clock variables:

```
a ?c
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 5
We cannot have multiple references to clock variables in an expression:

```
a !(c+c)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 6
Clock variables `c` can only occur in expressions of the form `c+<expression>`:

```
a !(c*2)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 7
Communication cannot lead to `palt`:

```
a !n palt b {= :2: b =}
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 8
The left-hand of a conditional receive subexpression must be the receiving expression string:

```
a ?(ar[1]:ar[0+1]<5.0)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 9
Receive conditions must be conjunction of simple receive subexpressions:

```
a ?(x:x<5.0 || x>3.0)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 10
Comparison operators in receive subexpressions have to be from $\{<, >, \leq, \geq\}$. Equality would be equivalent to value matching, so use `!` instead.

```
a ?(x:x==10.0)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 11
Timed conditional receive: Comparison operators have to be from $\{\leq, \geq\}$:

```
a ?(x:x<c)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 12
Timed conditional receive: Expressions with clocks have the usual restrictions:

```
a ?(x:x<=c+c)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Test case 13
Timed conditional receive: Expressions with clocks have the usual restrictions:

```
a ?(x:x<=c*x)
```

expected result: *MoDeST compiler error*
actual result: *MoDeST compiler error* ✓

Assignments and Basic MoDeST

We have changed the way assignments behave. We now test whether assignments still behave as they did before the changes, except for the situations for which we corrected their behaviour:

Test case 14
Assignments basically still work:

```
a {= n=1, m=n, x=3.0*4.0 =};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	1	1,0000000000E00	✓
m	0	0,0000000000E00	✓
x	12.0	1,2000000000E01	✓

Test case 15
Assignments still work in `alt`, and `alt` is unbiased:

```
alt {
  :: a {= n=-1, x=-1.0 =}
  :: a {= n=1, x=1.0 =}
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	≈ 0	-4,4000000000E-03	✓
x	≈ 0.0	-4,4000000000E-03	✓

Test case 16
Clocks can be assigned to variables, and guards can still force time to advance:

```
a {= n=3 =};
when(c>=n) a {= z=c =};
when(z<=x) a;
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
n	3	3,0000000000E00	✓
x	0.0	0,0000000000E00	✓
z	3.0	3,0000000000E00	✓

Test case 17
In-place swapping of values is now possible:

```
a {= x=9.0, y=-1.0 =};
a {= x=y, y=x =};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	-1.0	-1,0000000000E00	✓
y	9.0	9,0000000000E00	✓

Test case 18
Parallel assignments are now executed atomically:

```
par {
  :: a {= x=5.0 =}
  :: a {= y=x-5.0 =}; a
};
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
x	5.0	5,0000000000E00	✓
y	-5.0	-5,0000000000E00	✓

Test case 19
Processes still work, especially w.r.t. parameters, local variables, and access to global variables:

```
process P(int m) {
  int n;
  a {= n=m-2, x=-12.0+m =}
}

a {= n=13 =};
P(13);
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	13	1,3000000000E01	✓
m	0	0,0000000000E00	✓
x	1.0	1,0000000000E00	✓

Test case 20
Assignments work with arrays as intended:

```
a {= ar[ar[1]+1]=1 =};
par {
  :: a {= ar[0]=ar[1]+5 =}
  :: a {= ar[1]=ar[0]-3 =}
};
{= x=ar[0], y=ar[1], f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	6.0	6,0000000000E00	✓
y	-3.0	-3,0000000000E00	✓

Solitary Value Generation

Communication statements outside of `par` only make sense when used to generate values. It is thus convenient to check the basics of value generation in this setting:

Test case 21
Probabilistic (uniform) resolution of value generation works, though with only 10000 simulation batches and value generation domains $[-2^{31}, 2^{31}]$, the values for n and x are expected to be close to the expected value only relative to 2^{31}:

```
a ?n;
a ?(m:m>0);
a ?x;
a ?(y:y>=-3 && y<=9);
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	≈ 0	4,4303087439E06	✓ (values vary wildly with RNG seed)
m	≈ 536870912	5,3505798932E08	✓
x	≈ 0.0	4,0977696426E06	✓ (values vary wildly with RNG seed)
y	≈ 3.0	3,0372183266E00	✓

Test case 22
Empty value generation domains yield deadlock:

```
a ?(x:x<0.0 && x>5.0);
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
x	0.0	0,0000000000E00	✓

Test case 23
Openness of bounds is respected for both integers and floats:

```
a ?(n:n>3 && n<5);
a ?(x:x<0.0 && x>=0.0);
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
n	4	4,0000000000E00	✓
x	0.0	0,0000000000E00	✓

Value Matching

Value matching is the simplest form of communication between parallel processes, so we test it first:

Test case 24
Simple forms of value matching work as expected:

```
par {
  :: a !(1.0+3.0) !(0.0)
  :: a !(4.0) !x
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	0.0	0,0000000000E00	✓

Test case 25
Value matching leads to deadlock if the values do not match:

```
par {
  :: a !(1.0+3.0) !(1.0)
  :: a !(4.0) !x
};
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
x	0.0	0,0000000000E00	✓

Test case 26
Value matching leads to deadlock if the types do not match:

```
par {
  :: a !(0.0)
  :: a !n
};
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
x	0.0	0,0000000000E00	✓

Test case 27
We are not limited to binary matching, and alternatives are chosen appropriately:

```
par {
  :: a !(3*3)
  :: a !(9)
  :: alt {
       :: a !(4+5)
       :: a !(4*5)
     }
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓

Value Passing

The next step up from matching values is to actually pass values around:

Test case 28
Value passing works, and it is performed atomically:

```
par {
  :: a !(3.0) ?y
  :: a ?x !x
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	3.0	3,0000000000E00	✓
y	0.0	0,0000000000E00	✓

Test case 29
Arrays are handled correctly:

```
a {= ar[1]=2 =};
par {
  :: a !(3) ?ar[ar[1]-2]
  :: a ?n !(ar[n+1]+2)
};
{= m=ar[0], f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	3	3,0000000000E00	✓
m	4	4,0000000000E00	✓

Test case 30
Value passing leads to deadlock if the types do not match:

```
par {
  :: a !(3.0)
  :: a ?n
};
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
n	0	0,0000000000E00	✓

Test case 31
We are not limited to binary value passing, and can even mix it with value matching:

```
par {
  :: a !(4) !(-3.5) !x
  :: a ?n ?x
  :: a ?m !(0.5*(-7))
  :: a ?f ?y
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	4	4,0000000000E00	✓
m	4	4,0000000000E00	✓
x	-3.5	-3,5000000000E00	✓
y	-3.5	-3,5000000000E00	✓

Conditional Receive

Adding conditional receive to communication between parallel processes is putting it all together: Value generation turns into value negotiation, and it all has to work.

Test case 32
Conditional receive basically works:

```
a {= m=3 =};
par {
  :: a ?(n:n<=m)
  :: a ?(m:m>n)
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	≈ 2	2,0012000000E00	✓
m	= n	2,0012000000E00	✓

Test case 33
Unsatisfiable conditions disable their alternative:

```
a {= m=3 =};
par {
  :: a ?(n:n<=m); a ?(n:n<m)
  :: a ?(m:m>n); alt {
                   :: a ?(m:m>4); b {= n=15 =}
                   :: a ?(m:m>=0)
                 }
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	≈ 0.5	5,0040000000E-01	✓ $\left(=\frac{1}{6}+\frac{1}{9}+\frac{2}{9}\right)$
m	= n	5,0040000000E-01	✓

Test case 34
Receive conditions work with value passing, but if the value is not licensed by the conditions, lead to deadlock:

```
a {= m=3 =};
par {
  :: a ?(n:n<=m)
  :: a !m
};
par {
  :: a ?(n:n<=m)
  :: a !(m+2)
};
{= f=1 =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
n	3	3,0000000000E00	✓
m	3	3,0000000000E00	✓

Timed Conditional Receive

Receive conditions without clocks can either allow some values or deadlock whenever no value exists or is sent that satisfies the conditions. With clocks, however, a receive condition can initially allow no values and therefore block, but become enabled when sufficient time has passed. We check whether this really works:

Test case 35
Timed conditional receive basically works and can force time to elapse:

```
a ?(x:x>=5.0 && x<=c);
{= f=1, z=c =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	5.0	5,0000000000E00	✓
z	5.0	5,0000000000E00	✓

Test case 36
It also works in `par` when the value being sent determines the time that has to elapse:

```
par {
  :: a ?(x:x<=c)
  :: a !(3.0)
};
a {= f=1, z=c =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	3.0	3,0000000000E00	✓
z	3.0	3,0000000000E00	✓

Test case 37
Sending the value of a clock variable can also cause a wait until it has the right value:

```
par {
  :: a ?(x:x>=7.0)
  :: a !c
};
a {= f=1, z=c =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	7.0	7,0000000000E00	✓
z	7.0	7,0000000000E00	✓

Test case 38
Having a clock variable both in the sending and the conditional receive expression is possible, but usually yields conditions that are either trivial or, as in this case, unsatisfiable:

```
par {
  :: a ?(x:x<=c)
  :: a !(c+2.0)
};
a {= f=1, z=c =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
x	0.0	0,0000000000E00	✓
z	0.0	0,0000000000E00	✓

Test case 39
There are other ways (and lots of them, too!) to create unsatisfiable conditions using clocks:

```
par {
  :: a ?(x:x<=c)
  :: a ?(y:y>=c+1.0)
};
a {= f=1, z=c =}
```

	expected result	*actual result*	
f	0	0,0000000000E00	✓
x	0.0	0,0000000000E00	✓
y	0.0	0,0000000000E00	✓
z	0.0	0,0000000000E00	✓

Test case 40
We can work with multiple clocks, but have to keep in mind that they advance synchronously:

```
par {
  :: a ?(x:x>=3.0) ?(z:z<=c)
  :: a !c ?(y:y>=5.0)
};
a {= f=1, y=d, z=c =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	5.0	5,0000000000E00	✓
y	5.0	5,0000000000E00	✓
z	5.0	5,0000000000E00	✓

Inter-Process Communication

In all previous tests, no explicit process declarations and instantiations were used. They introduce lots of complications with parameters and local variables. We now investigate whether this adversely affects communication:

Test case 41
Parameters hide global variables:

```
process P(int n) {
  a ?(n:n>3); a !n
}

par {
  :: P(n+3)
  :: a !(5); a ?m
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	0	0,0000000000E00	✓
m	5	5,0000000000E00	✓

Test case 42
Local variables hide global variables:

```
process P() {
  float x;
  a ?x; a !x
}

par {
  :: P()
  :: a !(5.0); a ?y
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
x	0.0	0,0000000000E00	✓
y	5.0	5,0000000000E00	✓

Test case 43
Arrays work, too:

```
process P(int n) {
  float as[3];
  a ?as[0] ?as[1] ?ar[n];
  a !as[ar[1]-1] !as[ar[1]] !as[ar[1]+1]
}

par {
  :: P(1)
  :: a !(5.0) !(n+2.0) !(1);
     a ?x ?y ?z
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	0	0,0000000000E00	✓
x	5.0	5,0000000000E00	✓
y	2.0	2,0000000000E00	✓
z	0.0	0,0000000000E00	✓

Test case 44
Communication even works in simple tail-recursive settings (more advanced settings are not possible since MoTor does not perform synchronisation between different processes when using tail-recursion):

```
process P(int n) {
  par {
    :: when(n!=0) a ?m
    :: when(n!=0) a!(m+n); P(n-1)
    :: when(n==0) {= f=1 =}
  }
}

P(3)
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	0	0,0000000000E00	✓
m	6	6,0000000000E00	✓

Test case 45
Multiple invocations of the same process work as expected:

```
process P(int n) {
  int m;
  a ?(m:m<n && m>=0); a !m
}

par {
  :: P(13)
  :: P(5*2)
  :: P(3+m+2*4)
  :: a; a ?m
};
{= f=1 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	0	0,0000000000E00	✓
m	≈ 4.5	4,5571000000E00	✓

Booleans

Last but not least, Boolean variables can also be sent, received, and matched. They behave somewhat differently (for example, conditional receive is equivalent to value matching and therefore redundant), so we should test whether they behave as intended:

Test case 46
Boolean value generation generates true or false each with probability 0.5:

```
a ?s;
a {= f=1, n=s?1:0 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	≈ 0.5	5,0760000000E-01	✓

Test case 47
Boolean value matching works:

```
par {
  :: a !(1.0==2.0*0.5)
  :: a !(!s)
};
a {= f=1, n=s?1:0 =};
par {
  :: a !(1.0==2.0*0.5)
  :: a !s
};
a {= f=2 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	0	0,0000000000E00	✓

Test case 48
Finally, Boolean value passing works, too:

```
par {
  :: a !true
  :: a ?s
  :: a ?t
};
a {= f=1, n=s?1:0, m=t?1:0 =}
```

	expected result	*actual result*	
f	1	1,0000000000E00	✓
n	1	1,0000000000E00	✓
m	1	1,0000000000E00	✓

Appendix B: MoTor on Windows

Before I started this thesis, no one had been working with MoTor in a Windows environment; and in fact, although Möbius ran fine, MoTor as it was would not work on Windows. Seeing a worthy challenge, I took advantage of Reza Pulungan's[1] infinite patience, which I'd like to thank him for, to find out the necessary steps to *make it work*, summarised in this short guide.

For reference, the environment used consisted of Möbius 2.0 and MoTor 20061129 on Windows XP with Sun's Java SDK 1.5.0_09 and a recent version of Cygwin[2] that had gcc 3.4.4, GNU Make 3.81 and the tcsh/csh.

The Problem

The root of most of the problems is that Cygwin of course uses Unix-style paths, for example representing `C:\WINDOWS` as `/cygdrive/c/windows`. Working with these Cygwin-style paths is fine as long as we stay entirely within the Cygwin environment. However, we have to leave the this environment at two points: When calling ANTLR to build MoTor, because this is a Java program and Java is not aware of Cygwin and its path syntax, and when Möbius calls MoTor to compile models, because again, Möbius is a Java application, and, being outside of Cygwin, does not know how to start the csh script that is used to call MoTor.

Wrapping Java for Cygwin

In order to be able to call `java` and the Java compiler `javac` with Cygwin-style paths, we need wrappers that first convert these to standard Windows paths. Fortunately, complete wrapper scripts already exist[3]; these just have to be modified so the paths to `java.exe`, `javac.exe` et al. are correct for the system at hand. Making a copy of each wrapper, adding the `.exe` suffix, will allow us to use both `java` and `java.exe` to call the wrapper from within Cygwin, which is important for MoTor's make script..

./configure

In the following, it is assumed that Möbius has been installed. The next step is to make sure the environment variable `CLASSPATH` is set; if not, `export CLASSPATH=.` helps. This should be all that is needed to run `./configure --with-mobius=<mobius toplevel folder>` from the MoTor root folder without any errors concerning Java or Möbius. If not, there is most probably a problem with the wrappers or the `PATH` environment variable.

[1] http://depend.cs.uni-sb.de/index.php?254
[2] http://www.cygwin.com/
[3] http://www.cygwin.com/cgi-bin/cvsweb.cgi/wrappers/java/?cvsroot=cygwin-apps

```
if(ttdfname.compare(0, 10, "/cygdrive/", 10) == 0) {
  int pos;
  ttdfname.erase(0, 10);
  ttdfname.insert(1, ":");
  while((pos = ttdfname.find("/", 0, 1)) != std::string::npos) {
    ttdfname.replace(pos, 1, "\\");
    ttdfname.insert(pos, "\\");
  }
}
```

Figure 30: Naïvely rewriting the path

Modifying MoTor

The next problem is that Möbius will try to call a csh script directly from a Windows (Java) application. This will fail, so the MoDeST plug-in has to be modified such that it calls a Windows batch file instead.

This can be accomplished by opening `MoDeSTInfo.java`[1], appending `.bat` to the string in `private String momodestPath="momodest";` around line 86, and appending `.replace('\\', '/')` in front of the semicolon in `cmd_args[½] = mif.getSourceFile();` around lines 191 and 198.

Another issue will occur when a compiled model is simulated because an absolute Cygwin-style path is written into a file. The guilty code can be found in main.cpp[2] around line 1055 (`ttdfname.append(".ttd");`). A very simple (though not elegant) way to resolve this is to add the lines shown in Figure 30 after the one mentioned above.

Compiling and Installing MoTor

At this point, running `make` from the MoTor root folder should yield lots of warnings, but no errors. If successful, running `make install_mobius` will install MoTor into Möbius. Möbius then has to be started once for configuration; after that, the current user's My Documents folder should have a new subfolder named `.mobius`.

In the `PMSettings` file in that subfolder, the line `<string id="InstalledModule"> Mobius.AtomicModels.MoDeST.MoDeSTInterface</string>` must be inserted somewhere next to similar lines.

Now, the batch file that Möbius will call has to be created. It should be called `momodest.bat` and put into Möbius' bin folder with the content shown in Figure 31, adapted to match the folders on the machine at hand and omitting the line breaks after `-c`.

[1] To be found in `backend/mobius/mobius-editor/Mobius/AtomicModels/MoDeST/`
[2] To be found in `backend/mobius/`

```
@echo off
IF "%1"=="--quiet" C:\Progra~1\Cygwin\bin\bash --login -i –c
  "/cygdrive/c/mobius/Mobius-2.0/bin/momodest %1 $(cygpath -u
"%2") %3"
IF NOT "%1"=="--quiet" C:\Progra~1\Cygwin\bin\bash --login -i –c
  "/cygdrive/c/mobius/Mobius-2.0/bin/momodest $(cygpath -u "%1") %2"
```

Figure 31: Batch file for Möbius

Installation Complete

Möbius should now work with MoTor when started from within Cygwin (and if Möbius' `bin` folder is in `PATH` within Cygwin).