The MODEST Modeling Tool and Its Implementation

Henrik Bohnenkamp[1], Holger Hermanns[1,2], Joost-Pieter Katoen[1], and Ric Klaren[1]

[1] Formal Methods and Tools Group, Department of Computer Science
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

[2] Department of Computer Science
Saarland University, D-66123 Saarbrücken, Germany

Abstract. This paper is about the tool-suite MOTOR that supports the modeling and analysis of MODEST specifications. In particular, we discuss its tool architecture, and the implementation details of the tool components that do already exist, in particular, the parser, the SOS implementation, an interactive simulator, and a state-space generator. As the expressiveness of MODEST goes beyond existing notations for real-time as well as probabilistic systems, the implementation of these tool components has a non-trivial intrinsic complexity.

1 Introduction

Contrary to traditional software engineering where correctness issues prevail, non-functional aspects such as reliability and performance are of paramount importance for *embedded* software design [14,23]. Nowadays, specification languages are used for the description of embedded system's behaviour at the various design stages. Some of these languages provide support for the representation of quantitative aspects, and a few, such as extensions of the Corba IDL [29,10, 15] have proved useful in a very pragmatic system engineering context, where they provide guidelines for run-time adaptation to ensure certain non-functional properties. These languages, however, are lacking a precise semantical meaning, and therefore do not support quantitative analysis at design time. Rigorous specification formalisms on the other hand, such as stochastic process algebras [21, 19,6,12], do have such precise semantics, but their learning curve is typically too steep from a practitioner's perspective.

Recently, we have developed the specification language MODEST that covers a wide spectrum of modeling concepts, possesses a rigid, process-algebra style semantics, and yet provides modern and flexible specification constructs [11]. MODEST specifications constitute a coherent starting-point to analyse distinct system characteristics with various techniques, e.g., model checking to assess functional correctness and discrete-event simulation to establish the system's reliability. Analysis results thus refer to the *same* system specification, rather than to different (and potentially incompatible) specifications of system perspectives like in the UML. MODEST is a modeling language that has a rigid formal basis and incorporates several ingredients from light-weight notations (e.g., SDL and the UML) such as exception handling. MODEST has a stochastic process algebra 'core' and contains features such as simple data types, structuring mechanisms like composition and abstraction, atomic statements, non-deterministic and probabilistic branching and timing.

With MODEST, we take a *singleformalism*, *multisolution* approach. Our view is to have a single specification that addresses various aspects of the system under consideration. Analysis takes place by extracting simpler models from MODEST specifications that are tailored to the specific properties of interest. For instance, for checking reachability properties, a possible strategy is to "distill" an automaton from the MODEST specification and feed it into an existing model checker such as provided by CADP [16]. On the other hand, for carrying out an evaluation of the stochastic process underlying a MODEST specification, a discrete-event simulator can be used. Viewed differently, one may consider MODEST as an overarching notation for a spectrum of models, ranging from ordinary finite-state automata, to timed automata [7], and discrete event stochastic processes such as stochastic automata [12] and Markov Decision Processes [28].

Our Contribution. In this paper, we focus on the tool MOTOR (MODEST Tool enviRonment), which is aimed to provide the means to analyse and evaluate MODEST specifications. The tool is written in the C++ programming language. The reason for this choice was the good trade-off between speed of the implementation, modularity of the code, and easy extendability. We first discuss the architectural issues that arose in the planning phase of MOTOR. Since MODEST is capable of describing many different aspects of a system design, a tool supporting such a language has a non-trivial intrinsic complexity. The second topic and main contribution of this paper is the implementation of the structural operational semantics of MODEST. Although tools for process algebraic languages are around for some time (several of them mentioned below), only few are written in object-oriented languages like C++ or Java, and, to our knowledge, their implementation details have not been described in the literature. Our paper is therefore a practical contribution to tool development for process-algebraic specification languages like MODEST in an object-oriented programming environment.

Related Work. *Multiformalism Approaches.* In our approach, a single formalism is used to describe the various aspects of the system under study. Alternatively, the "multiformalism multisolution" approach in MÖBIUS [13] supports several modeling formalisms and various solution methods. Earlier work in this direction has been reported for the tool SHARPE [30]. The APNN-TOOLBOX [3, 2] also supports the integration of multiple modeling formalisms into a single software tool-environment, as does GREATSPN [9].

Open Tool Architectures. CADP [8,17] is a widespread toolkit for design and verification of complex systems. The toolbox is designed as an open platform for the integration of other specification, verification and analysis techniques. Similar ideas are implemented in the Acf prototype [24], which provides Apis for standard state space exploration functions, and they can also be found in Möbius [13].

Stochastic Process Algebra Tools. Various software tools exist that support the modeling and analysis of stochastic process algebras (SPAs). The PEPA workbench [18], TIPPtool [20], TwoTowers [5] (that supports EMPA) are some prominent examples of such tools. The Modest language incorporates most functionality of the SPAs supported by these tools apart from priorities.

2 The Modest Modeling Language

In this section, we highlight the most important features of Modest, as far as they are needed to understand the rest of the paper. For a complete overview, we refer the reader to [11].

2.1 The Modest Concepts in a Nutshell

Modest is a process algebra, enhanced with some additional features. The most basic activity is an *action*. Similar to process algebras like CSP, ACP and CCS, Modest actions are combined by means of operators, building processes: alt and do describe a choice between processes (terminating and looping, respectively), par describes parallel composition, hide and relabel the operators for hiding and relabelling actions. Modest allows to describe probabilistic choice by means of the palt construct: after the execution of an action, a successor state can be chosen probabilistically. It has a notion of *guard* (a boolean condition describing when an action is allowed to execute) and *deadline* (a boolean condition describing when an action *must* fire at the latest), described by the when and urgent operators, respectively. To make guards and deadlines work, simple data-structures are incorporated, and a notion of *clock* is used, a variable-like entity that changes its value linearly and continuously with time. Data and clocks can be evaluated in guards and deadlines, and can be manipulated in palt constructs.

2.2 Stochastic Timed Automata

The formal semantics of Modest has been defined in [11]. We briefly present the semantic objects, called *stochastic timed automata* (Sta), that are derived from Modest specifications by means of a structural operational semantics.

An Sta consists of a set L of (control) *locations*, and a set of labelled transitions of the form $s \xrightarrow{a,g,d} \mathcal{P}$, where s is a location, a an *action*, g a *guard*, and d a *deadline*. $\mathcal{P}$ is a *discrete* probability distribution on pairs of the form $\langle s', A\rangle$, where s' is a location and A is a (possibly empty) sequence of assignments. The probability distribution is used to express discrete probabilistic branching. We denote by $\mathbf{P}_{\mathcal{P}}\langle s', A\rangle$ the probability that the pair $\langle s', A\rangle$ is chosen.

An STA is a very abstract representation of a MODEST model. It describes the *potential* moves between locations. A more concrete description is obtained if, besides locations, we consider the *valuation* of data variables and clocks. A valuation assigns to each variable its value. A location together with a valuation is called a *state*. For a given valuation, an STA transition is interpreted as follows. The system is allowed to fire transition $s \xrightarrow{a,g,d} \mathcal{P}$ whenever it is at location s and the guard g is true under the current valuation. As soon as deadline d becomes true (say, at time t), the system is *obliged* to fire the transition at time t without any delay, if it has not been fired before. If the guard is false, when the deadline becomes true, the whole system locks: the semantics of STAs does not allow time to progress anymore. This situation is called a timelock. The system is thus allowed to wait in location s as long as no deadline of either one of its outgoing transitions becomes valid. Once the transition $s \xrightarrow{a,g,d} \mathcal{P}$ is executed, the system moves with probability $\mathbf{P}_{\mathcal{P}}\langle s', A\rangle$ to location s', assigning values to variables (*i.e.,* changing the valuation) according to A in an atomic manner.

3 The Motor Tool Architecture

The MODEST language is very expressive and has been designed without any focus on a particular analysis technique covering the entire language spectrum. Actually, many analysis problems will be undecidable or at least much too complex to be tackled effectively. Therefore, to analyse and evaluate MODEST specifications, it is necessary to have sound and well-designed means to *project* specifications so that the projected model can be analysed with a particular technique, still providing meaningful information about the original model.

From the perspective of a tool developer, it appears worthwhile to design a MODEST tool that (i) provides interfacing capabilities for connection to existing tools for specific projected models, but which (ii) also provides means for enhancement by *native* algorithms for analysis of (classes) of MODEST specifications. These requirements induce a natural architectural structure of MOTOR, as described in the next section.

3.1 Overview

The philosophy behind MODEST as described earlier suggests a specific architectural setup. The principal idea is to hide the core functionality, *i.e.,* the language-specific parts that are common to several subsequent analysis and post-processing functions, behind a set of well-designed software interfaces. These interfaces enable the connection of the MOTOR core (called the core module) to several other, external tools. To facilitate easy extendibility, the MOTOR core is equipped with a *modular* structure, *cf.* Figure 1. In the following, we describe the *core module* and the interfaces that serve as docking points for other, so-called *satellite* modules.

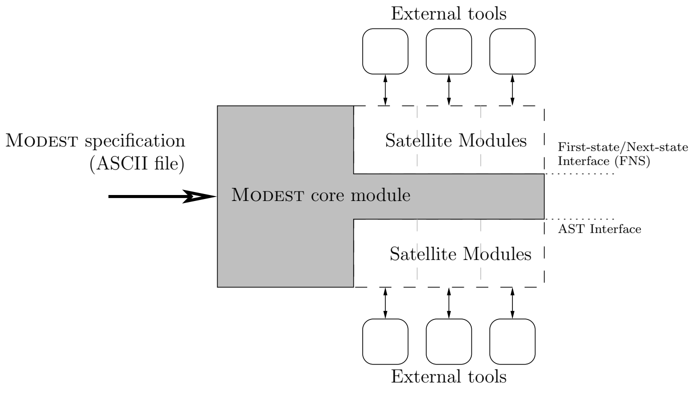

Fig. 1. The Motor architecture

3.2 The Core Module

The core module takes a Modest specification as input, which has been composed with an ordinary text editor. The core module consists of two parts: the language parser, that produces a parser abstract syntax tree (PAST) from a Modest specification, and an implementation of the operational semantics (SOS). Two programming interfaces are offered to the user: the *first/next-state interface* (Fns), and the AST (abstract syntax tree) interface.

The First/Next-State API. This interface provides access to the global state space of the Modest specification at hand and is similar to the OPEN/Cæsar interface of the Cadp tool environment [16], or the concepts behind the Analyser Component Framework (Acf) [24]. The user of the interface (which can be another software component) can query the initial and the current state of the Modest specification, can derive transitions that are enabled from a given state, and can derive successor states by firing transitions. The mechanisms behind the Fns only keep information about the state that was reached by firing the last transition. The Fns provides however *external representations* of the current state, which are created and destroyed dynamically and are used by the user to reset the internal state of the core module to that represented by the external state. One of the components that make use of the Fns are state-space generators, test-case generators such as TorX [4], or discrete-event simulators.

Hierarchical Semantics. The Fns is a first step towards realising our aims behind Modest. The described interface provides access to Sta states and transitions. The Sta is the most abstract representation of the behaviour of a Modest specification. To enable the use of different analysis algorithms, it must be possible to express more concrete representations of behaviour. A very simple example of such a concretisation is obtained by taking data into account (*cf.* Section 2.2): an Sta does not have a notion of memory or store. The guards and assignment on Sta transitions remain completely uninterpreted. A concretisation of the Sta would then be to enhance the definition of state by a notion of *valuation*, a function that keeps track of the values assigned to variables. The concretisation of a transition then comprises the interpretation of guards and deadlines according to the valuation of the source state, and the modification of valuations according to assignments. To describe concretisations of STA, we will use the *layered semantics* approach [1], which allows to define (more concrete) transition systems in the deductive style of a structural operational semantics, based on abstract transition systems. This approach allows to define *hierarchies* of semantical concretisations. From an implmentors point of view, the layered semantics approach is best realised by means of class inheritance: the states of a concrete semantical layer are realised as subclasses of the class representing abstract states. Accordingly, more concrete transitions are described by subclasses of the classes describing abstract transitions.

AST Interface. The second API is the AST interface that provides access to the abstract syntactic description of a MODEST specification. Since some tools that we are aiming to connect to have a high-level input formalism, it is more convenient to translate the MODEST specification directly into this language. This can usually be performed by recursive traversals of the abstract syntax tree of the specification.

Satellite modules. The satellite modules provide the *real* functionality to MOTOR. They implement either adaptor modules that bridge the gaps between the core and other, external tools, or '*native*' modules, which implement analysis algorithms within the MOTOR framework. Satellite modules make use of the FNS of the AST interface. In the former case, a satellite module might also implement a concretisation of the FNS, providing itself a more concrete FNS.

3.3 The Current State of Implementation

Core Module. The core module has been implemented and provides currently the AST interface. The FNS API is also available. Its implementation realises data access and manipulation. Therefore, the notion of state and transition provided by the FNS is more concrete than that of an STA. It is utilised by a state-space generator and an interactive user interface (see below). The FNS, as it is implemented now, does, however, not yet have the flexibility needed to write (layered) satellite modules on top of it.

Available Satellite Modules

Interactive Simulator. To test the implementation of the MODEST SOS, we have implemented a simple interactive simulator which allows to examine the state of MODEST specifications, derive possible transitions, check for enabledness of these transitions, and to execute them. This simulator provides basically a simple textual user interface to the implemented FNS API.

State-space Generator. We have implemented a simple state-space generator which makes use of the implemented FNS. The generator can produce .dot files [22], .aut files, a textual description of labelled transition systems, and .bcg files, a compact file format used by the Cadp tool environment. The latter format enables to bridge to the verification and visualisation components of Cadp which are mostly focused on functional behaviour, as well as the on-the-fly test-generation tool TorX.

Möbius. Recently we finished the implementation of a satellite module that translates Modest specifications into the AFI of Möbius [13]. Modest is incorporated as a new atomic model specification formalism into Möbius. Naturally, the module makes use of the Ast interface.

Concluding Remarks. Note that we have not addressed the issue of how to specify properties to be analysed by target tools, and how to get the results back. This is currently an open issue. The diversity of tools we plan Motor to connect to makes it difficult to develop a clean concept *a priori*. However, we believe the approach we have chosen is flexible enough to support all kinds of solutions for this problem, either by providing dedicated user interfaces of preprocessors to Motor and the target tool, or by letting Motor vanish completely under the hood of the target tool (as is the case with Möbius).

4 Implementation of the Core Module

In this section, we describe the implementation of the Motor core module. The core module currently comprises about 23,000 lines of code. For code generation we used the latest version of the GNU g++ compiler. The core module comprises a language parser and the implementation of the Modest SOS. Motor has a simple command-line user interface and takes a plain ASCII file with the modest specification as input.

4.1 The Parser

The parser module converts a concrete syntax representation of a Modest specification into a PAST (parser abstract syntax tree). During the parse run a first set of semantic and syntactic checks is performed. Emphasis has been put on providing informative feedback to the user about errors in the input, ranging from simple syntactic errors to typing errors. The parser module heavily relies on the parser construction tool ANTLR [25]. ANTLR is also used in other modules to generate tree walkers used for the conversion of a PAST into more specific representations.

4.2 The SOS

The SOS module uses the AST API, and provides the Fns API. As described in Section 3.3, we implemented a notion of state and transition more concrete than Sta. A state consists of a location, which is a syntactic entity, and a valuation of all variables and clocks that are defined and visible (i.e., relevant) in the current location. The SOS module encapsulates this information. Our main design objectives of the implementation of the SOS module were: (i) it should follow the structure of the SOS-rules [11] as close as possible, such that correctness of the implementation can be ensured to a considerable extent by simple code inspection; (ii) it should support full MODEST, in particular allowing arbitrary nesting of operators. This means that processes can be created and terminated dynamically. The design of the necessary data structures is thus highly challenging.

Representing Locations. Structural operational semantics (SOS) typically define a transition system describing the behaviour of a syntactic entity [27]. The deductive nature of SOS suggests a recursive algorithm to derive the transitions (and therefore, the locations) of the underlying transition system. An SOS rule has the following form:

$$\frac{P_1 \xrightarrow{l_1} P_1' \quad \dots \quad P_n \xrightarrow{l_n} P_n'}{op(P_1, \dots, P_n) \xrightarrow{l} P'} \qquad \textit{(side condition)},$$

which means that the outgoing transition(s) of the location $op(P_1, \dots, P_n)$ (where op is an n-ary language operator) are defined in terms of the outgoing transition(s) of the locations $P_1, \dots, P_n$. A transition for $op(P_1, \dots, P_n)$ can be derived provided that the side condition is true, and each of the preconditions $P_i \xrightarrow{l_i} P_i'$ is satisfied. As the behaviour of each language operator is described by its SOS rules, the behaviour (in terms of outgoing transitions) of a complete specification can be determined by recursively computing the outgoing transitions of its sub-processes. This recursive scheme has been implemented.

We do not use syntactic term rewriting to transform locations into successor locations, since this would be too inefficient in time as well as space. We instead decided to use a shared data structure to represent locations. Since all locations that can be derived by means of the SOS are actually (combinations of) sub-expressions of the initial location, we use a single AST that describes the whole MODEST specification at hand, and add attributes to the AST to denote the current location to be represented. Since sub-trees of an AST correspond to sub-expressions on the concrete syntactic level, such attributed AST can indeed express all possible sub-expressions (*i.e.,* locations) of a MODEST specification. The inner nodes of the AST correspond to the language operators of MODESTand have an arity $n \geqslant 1$, like `do`, `par`, `alt`, etc.. The leafs correspond to atomic processes, *i.e.,* plain actions, exceptions, `break`, etc. These can be seen as language operators with arity 0. The attributes we added to the inner nodes do express which of the respective sub-trees does currently contribute to the current location, *i.e.,* is currently *active.*

Example. Figure 2 depicts two ASTs of the MODEST process `alt` { `:: a; e :: b; c; d` }. An **alt** node represents a nondeterministic choice between its two sub-trees while a **seq** node represents the sequential composition of its sub-trees (in this case, just leafs representing actions). Nodes are attributed: **alt**

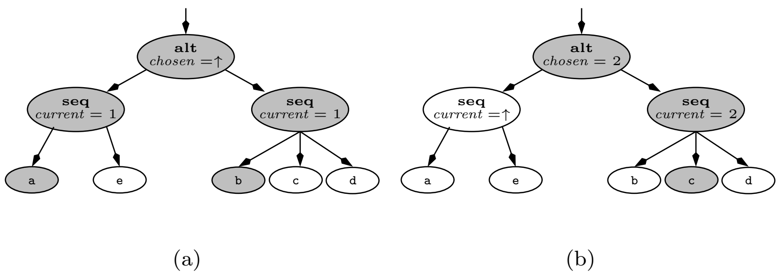

Fig. 2. Example for attributed AST

nodes have an attribute *chosen* indicating which of its sub-trees has executed an action first while **seq** nodes have an attribute *current* denoting which of its sub-trees is currently allowed to execute actions. Nodes are either active (gray) or inactive (white). Figure 2 (a) denotes the initial location of the process: the **alt** node has not chosen a branch yet ($chosen = \uparrow$), and both **seq** nodes activate their respective first subtree ($current = 1$), the atomic processes `a` and `b`, respectively. Figure 2 (b) denotes the location reached after the `b` action has been executed. Attribute *chosen* equals 2, since the right subtree has executed the b action. The left subtree of the **alt** is deactivated, as it cannot exhibit any behaviour anymore. For the right **seq** node, $current = 2$, since the first subtree was the `b` action, which has been executed already. Therefore, the second action is active now. ●

Representing locations by means of ASTs has an interesting property that we exploit in our implementation: a location is uniquely determined by the set of active leafs: the ancestor nodes of all active leafs are also active,and no other node. Therefore, determining the active leafs suffices for establishing the active inner nodes and their attributes.

Deriving Transitions. The primary purpose of the SOS implementation is to derive the outgoing transitions from the current location. This is accomplished by an algorithm that for a given node of the AST returns all the transitions that the process represented by this node can perform. An outline of this algorithm for a node that represents the operator `op` is:

Step 1: Get transitions of all sub-trees that are active according to the attributes of the node (this step is obsolete for atomic processes).

Step 2: Combine the thus derived transitions to new transitions according to the SOS rules defined for `op`. For atomic processes, only one, constant transition is returned.

The implementation of the different SOS rules is straightforward and, for lack of space, not further described here.

Firing Transitions. The second purpose of our SOS implementation is to administer location changes, *i.e.,* to fire transitions. To explain how this is implemented, we must first explain how we represent transitions. Firing a transition means actually to execute actions. In our implementation, a transition contains a collection of references to the leafs in the AST that represent the actions to be executed. If a transition contains more than one of these references, all the referenced leafs/actions take part in a synchronisation. All of these leafs/actions have to be 'executed', and the 'execution' of an leaf/action is a two-step process:

Step 1: A recursive bottom-up algorithm propagates to all nodes on the path from the current leaf to the root that an action has been executed. This information is important for **do** and for **alt** nodes: based on it, they will deactivate all their respective sub-trees that represent the branches of the choice that have "lost" the competition (unless, of course, the choice has been decided earlier already, in which case nothing happens).

Step 2: A second recursive algorithm runs bottom-up from the leafs to indicate *termination*: it informs the respective parent node that the sub-process of the calling node has terminated. The parent node has then to interpret this information relative to its own attributes. For example, a **seq** node that gets informed that one of its sub-trees has terminated will then activate the next subtree and update its own attributes. In case there is no next subtree, the **seq** node will inform its own parent node that it has terminated itself.

The both steps are actually combined into one, but are described here separately for reasons of clarity.

Process Instantiation and Recursion. An important concept in MODEST is that of *process definitions* and *process instantiations.* In particular, this allows us to specify *recursive* processes. In our implementation, a process definition is a mapping from a process name (a string) to an AST, as described above. A process instantiation in the concrete syntax is interpreted as an unary operator which has one parameter: the name of the process to be instantiated. Instantiations are therefore represented in the AST by nodes of type **inst**, which have an attribute denoting the process name. An **inst** node has the (among others) following tasks to perform: (1) it has to serve as a place holder for a process; (2) once it gets enabled by a parent node, it has to make a copy of the AST of the process that it represents, and make itself the root of this copy. This means that during run time the AST of the overall process is dynamically extended. Once the subtree of an **inst** node has been generated, the node does behave neutral with respect to transition-generation and -firing. Other tasks of **inst** will be discussed later.

Probabilistic Branching. The `palt` operator has several tasks in the MODEST language. It describes a transition from one location to another and expresses probabilistic branching. Each branch can have assignments. Its behaviour can be viewed as a weighted hyperedge connecting a single source location and possibly several target locations. In our implementation, the `palt` construct is represented by a **palt** node which has always an atomic process as a subtree (the *palt action*), and zero or more sub-trees that denote the possible successor processes. The probabilistic choice *itself* which is described by a `palt` process *is* *not explicitly implemented.* Instead, the user has to decide what to do in case of a probabilistic branching. A state-space generator might just want to traverse all possible branches, without taking probabilities into account. A stochastic simulator, on the other hand, might want to choose the next branch randomly, according to the given probability distribution.

A **palt** node has an attribute denoting the branch that is going to be activated after the palt action has been executed. This attribute has to be set externally by the user.

From Locations to States. We will now describe our approach to access and modify variables. Recall that the state of a MODEST specification is determined by its control location together with the valuation of all data variables and clocks that are defined and visible in the current location. The most involved part in our implementation is to obtain states from locations. Note that the implementation of the data part in the core module is already a concretisation of the MODEST SOS. The current implementation of the data part, being in an early prototype state, does not yet comply with the architectural ideas we have discussed in Section 3, but is rather a monolithic approach.

Data is *manipulated* in the `palt` construct by means of assignments. Data is *accessed* in `when` constructs, which describe enabling conditions for transitions, and the `urgent` constructs, which describe the conditions when a transition *must* fire at the latest.

Fun with Boolean Functions. Before we present the details of our approach to handle data, we first describe the evaluation of guards and deadlines. For the sake of simplicity, we denote them both as *conditions.* Conditions are Boolean expressions, and one way to deal with them is just to evaluate them, *i.e.,* to find out whether the expression evaluates to true or to false. In MODEST, this approach is not sufficient, since we have to distinguish between two types of conditions: *static* conditions and *dynamic* conditions. Static conditions are those which do *not* refer to clocks. Dynamic conditions are those that do. The important difference is that static conditions will never change their evaluation over time, unless the valuation of the variables they refer to is changed (which can only happen with the firing of a transition). For dynamic conditions, this is different: since they depend on clocks, their evaluation *can* change over time. Therefore, dynamic conditions define implicitly Boolean functions $b : \mathbb{R}^+ \longrightarrow \{\mathit{true}, \mathit{false}\}$ which depend on one continuous parameter: time. In our implementation, we have incorporated an explicit representation of these Boolean functions, since it must be possible to derive the earliest time when a guard or deadline becomes true. In the following, we will denote these functions as BoolFun.

Declarations and Data Blocks. Assignments, guards and deadlines refer to data, and thus it is necessary to make the data values accessible. Declarations are the syntactic entities in a MODEST specification that describe the structure of the data to be used. There are two levels of scope in MODEST: the global scope, in which access to global variables is possible, and local scopes, which are delimited by the body of a process definition.

A variable needs a certain memory space to which its values can be written to and read from. Therefore, in our implementation, we have to reserve memory for declared variables. Variables that are defined in the same scope can be grouped together, *i.e.,* we allocate memory blocks which are large enough to hold the values of all variables that are declared in the same scope.

How does the data fit into the concepts we have introduced so far? Observe that every instantiation of a process with variable declarations opens a new scope and forces to allocate memory for these variables on activation of the process. To that purpose we introduced an attribute for **inst** nodes that points to the memory block storing the values of the variables declared in the process definition represented by the **inst** node. Allocation and deallocation of this data block is done during the generation of the subtree of the **inst** node.

Accessing and Modifying Data. There are two concepts in Modest which access data. Guards and deadlines access variables without modifying them. Assignments, on the other hand, may also modify them. There are two principally different styles to implement the handling of guards, deadlines and assignments. The first is *interpreter-oriented*, the second *compiler-oriented.* In the first approach, a software component must be implemented that interprets the (abstract) syntactic description of conditions and assignments and takes the necessary steps to provide, for example, Boolean values that describe the enabledness of conditions, or to modify the data according to an assignment. In the second, compiler-oriented approach, a software component is generated *at execution-time* of the tool, which is then compiled and dynamically linked back into the running tool. We have implemented a combination of both approaches. We *interpret* all composed conditions, *i.e.,* all conditions that are composed by means of the logical *and*, *or*, or *not* operators. We call conditions which are not composed *primitive.* The tool restricts primitive conditions to be of the form $c \sim expr$ or $expr \sim expr$, where c is a clock, $expr$ is an arbitrary arithmetic expressions *not* referring to clocks, and $\sim \in \{<, \leqslant, >, \geqslant\}$.

For primitive conditions and assignments we rely on a compiler-oriented approach. For both concepts C functions are generated. This relieves us from the duty to also interpret the arithmetic expressions that occur in assignments or conditions. Although an interpreter-oriented approach is straightforward to implement, a compiler-oriented approach is much more efficient, since expensive traversal of the syntax tree of the expressions, and table look-ups for data access are not necessary.

For each primitive condition, two C functions are generated. The first function evaluates the guard and returns true or false, depending on the data that is passed as a parameter to the function. The second C function returns the BoolFun for the atomic condition, *i.e.,* the partially evaluated condition which has the *time* as its only remaining parameter. These two functions are provided regardless whether the condition is dynamic or static. In the latter case conditions can be represented as BoolFuns which are constant true or false.

Additionally, each assignment is translated into a C function. The generated C file contains all the generated functions and is then compiled automatically. The resulting object file is dynamically linked to the running program, and the names of the generated functions are resolved to function pointers. These function pointers are then assigned to attributes in the respective nodes of the AST: the pointers to assignment functions are assigned to attributes in the corresponding **palt** node. The pointers to the condition-related functions are assigned to attributes in the **when** and **urgent** nodes.

Parameter Passing. Process definitions can have parameters. Parameters are treated as local variables in the process definition. The difference to "normal" local variables is that they have to be initialised, when a process is instantiated. For this reason, also a parameter-passing C function is generated which takes care of the initialisation of the respective variables.

External State Representation. In our implementation, the state of a MODEST process is defined implicitly by the attributes of the AST and the allocated memory blocks holding the data. In order to do an exhaustive state space generation, it is however necessary to have an *explicit* state representation, which allows state-comparisons. This process is often called *state-matching*, and makes it possible to determine whether a state has been visited before or not. Clearly, it is much too inefficient to copy the complete AST and all memory blocks whenever a state change has occurred. Here our observation comes to help that the current location of a MODEST process is already defined by the set of active leafs in the AST. The active leafs define a set of paths through the AST to the root of the AST. So our external, explicit state representation exhibits information about the active leafs.

Additionally, an external state representation has to know about the valuations of the variables of the currently active processes. As indicated earlier, each process instantiation has its own, private memory block which contains the data. In the external state representation, we store the *pointers* to these memory blocks. This is combined with a copy-on-write mechanism to implement state changes caused by assignments. While executing a transition, assignments may modify data, and this requires to copy the data space of the state. By using pointers, however, copying the data means just copying these pointers, except for memory blocks which are subject to modification due to assignments. Only these memory blocks are actually copied, and necessary changes are then applied to the copy, as specified by the assignment. Relative to a naïve copying of entire data spaces, this approach has the following advantages.

- the copying of states is much faster, since only pointers are copied.
- if memory has to be copied, only the memory block that really changes is copied.
- since memory is shared, the memory utilisation is higher.
- state-matching is faster, since it can utilise pointer comparison: if we have to compare the memory blocks of two process instantiations, a comparison of

the respective pointers in the external state representation might show that they point to the same memory block. If they are identical, then it is assured that both process instantiations in both states have identical data. Only if they differ, it is necessary to compare the memory block itself. Below, we will describe state-matching in greater detail.

In the following, we assume that the nodes of an AST are uniquely numbered. These *node numbers* are used for the external state representation. More particular, our external state representation comprises the following ingredients:

1. the set of node numbers of the active leafs in the AST. This defines implicitly and uniquely all relevant paths to the root of the AST, all active inner nodes, and therefore, the current location of the state.
2. a map from node numbers to memory pointers. The domain of this map is the set of all node numbers of those **inst** nodes that lie on one of the paths from the actives leafs to the root of the AST. The node number of an **inst** node is mapped to the pointer of the memory block that is maintained in this **inst** node.

State Matching. State-space generators have to keep track of states that have been reached already in order to detect loops. The standard generic container data structures of the C++ Standard Template Library (`set`, `list`, `stack`, `queue`, etc.), expect the definition of equality (==) and a strict order (<) on element classes. In the following, we *define* equality and the order on states, and do therefore refrain from pointer comparison, as described above. Using pointer comparisons is an implementation detail for efficiency improvement, that we can make use of due to the copy-on-write mechanism described above. Including the comparison in the definition would, however, only complicate things unnecessarily.

For illustration purposes, we assume two states, s and s', and denote the set of leaf numbers as $s.l$ and $s'.l$, respectively. By $s.m$ we denote the mapping from node numbers to pointers. Then, $\mathrm{dom}(s.m)$ is the domain of the map, and, for $n \in \mathrm{dom}(s.m)$, $s.m(n)$ denotes the pointer of the memory block mapped to. Abusing C/C++ notation, we denote by $*s.m(n)$ the memory block that is pointed to by $s.m(n)$. Then equality is defined as follows: $s == s'$, if $s.l = s'.l \wedge \bigwedge_{n \in \mathrm{dom}(s.m)} *s.m(n) == *s'.m(n)$. In order to define $<$, we assume that there is a strict order $\prec$ on sets of integers. We also assume w.l.o.g. that, if $\mathrm{dom}(s.m) = \{n_1, \ldots, n_k\}$, then $n_1 < n_2 < \cdots < n_k$. Then we define $s < s'$, iff

$$\begin{aligned}
s.l \prec s'.l \vee s.l = s.l' \wedge [&*s.m(n_1) < *s'.m(n_1) \\
&\vee *s.m(n_1) = *s'.m(n_1) \wedge [*s.m(n_2) < *s'.m(n_2) \\
&\vee \cdots \\
&\vdots \\
&\vee *s.m(n_{k-1}) = *s'.m(n_{k-1}) \wedge *s.m(n_k) < *s'.m(n_k)] \cdots]].
\end{aligned}$$

We use brackets to emphasis the nesting of the formula.

4.3 Classes

In this section, we briefly describe the classes that are used to implement the concepts described in the previous section.

<u>`BoolFun`.</u> The class `BoolFun` implements the BoolFuns. The BoolFuns make use of a class `Flank` of so called *flanks*. Flanks denote the changes of the BoolFun from false to true or vice versa. A `Flank` object is basically a triple (d, u, c), where d denotes the position of the flank on the time scale, u denotes whether it is a change from false to true (UP) or from true to false (DOWN), and c denotes whether the following interval is left-closed (CLOSED) or not (OPEN).

The class `BoolFun` has the following important methods:

`Flank getFirst():` returns the first *flank* of the function from false to true.
`bool operator()(double t):` truth value of the BoolFun at time `t`.

<u>`Guard`.</u> A `Guard` object represents a condition. `Guard` is an abstract class. There are several concrete sub-classes: `AndGuard, OrGuard, NotGuard` are guards that represent Boolean combinations of guards. The class `FuncGuard` represents a primitive static condition, the class `TimedGuard` an primitive dynamic one. The important method of all these classes is:

`BoolFun getBoolFun():` returns the BoolFun defined by the `Guard` object.

<u>`State`.</u> The `State` class implements the external state representation as described before. There are two external operators defined:

`bool operator==(const State& rhs, const State& lhs):` checks equality of states.
`bool operator<(const State& rhs, const State& lhs):` checks the strict order as defined earlier.

<u>`Transition`.</u> The `Transition` class represents STA transitions. A `Transition` object has an action name, a guard, a deadline, and an abstract notion of the probabilistic branching. The important methods of `Transition` are:

`bool isNeverEnabled():` returns true, if the guard of the transition is constantly false, independent of time that may pass.
`Guard* getGuard(), Guard* getDeadline():` returns guard and deadline of the transition.
`void Fire(coordinate_t& c, double offset):` executes the transition, *i.e.,* it initiates the state changes in the AST that lead to the internal representation of the destination state of the transition that is represented by the `Transition` object. The `Transition` class provides methods that return a description of the possible branches and their probability. A branch is identified by an object of type `coordinate_t`, and the parameter `c` denotes the branch that the caller has chosen.
The parameter `double offset` is a time offset. It determines the delay before the transition actually fires. Part of the state change is then to advance the valuation of each clock (that is not being reset) by `offset`.

`ModestModel`. The `ModestModel` is the topmost entity in the SOS implementation. It is constructed during a traversal of the PAST. It contains all information about the specification at hand, *i.e.,* all process definitions, and it harbours the AST used to represent states. The `ModestModel` provides part of the FNS. The provided methods are:

`State* getInitialState()`: returns a pointer to the initial state.
`State* getCurrentState()`: returns a pointer to the current state.
`transition_list& getTransitions(State* state)`: returns a list of STA transitions outgoing from `state`.

`Process`. The nodes of the AST are all sub-classes of the (abstract class) `Process`. The `Process` class has the following methods:

`transition_list& getTransitions()`: returns all transitions that can be derived from the process node in the current state.
`void reset()`: resets the process and all its sub-processes to their initial state.
`void disable()`: This method disables the process and all its sub-processes.

An important, direct subclass of `Process` is `ComposedProcess`, the class that represents all inner nodes of the AST. It has the following additional methods:

`propagateActivity()`: implements the propagation mechanism as described in Section 4.2.
`childDone()`: implements the termination mechanism as described in Section 4.2.

5 Future Directions and Concluding Remarks

In this paper we have presented the design and architecture of the MODEST tool environment MOTOR, and discussed the implementation of the MOTOR core module, the central component of the tool prototype. We have also briefly commented on the current state of the implementation and the existing tools that we already have connected MOTOR to.

Our next effort will be to modify the FNS API implementation such that it is possible to implement hierarchies of semantics on top of the STA semantics (*cf.* Section 3). Our second short-term aim is to connect MOTOR to UPPAAL [26], a model checker for timed systems.

MODEST is capable of expressing many different known formalisms (for an overview, see [11]), like labelled transition systems, timed automata, CTMCs, GSMPs etc. Currently, MOTOR does only support simulation of GSMPs with MÖBIUS, and allows the derivation of LTS with a simple state-space generator. The expressiveness of MODEST comes at a prize. Some of the underlying model classes support concepts that cannot be treated properly in other classes. The most prominent example is nondeterminism, which usually cannot be dealt with in a stochastic analysis (be it numerical, or a simulation). Exceptions are Markov Decision Processes [28], in which at least bounds of the measures of interest can be obtained. One of the problems that arises is to check if a given Sta fulfils the restrictions dictated by a certain target formalism, or not. For example, although Modest can express CTMCs, it is not trivial to check that the Markov property is not violated by a given Sta. It is also very difficult to detect nondeterminism in a stochastic model, without doing an exhaustive state-space analysis. All these problems have to be addressed to make Modest/Motor a success, and make the whole project very challenging.

Acknowledgements. This work has taken place in the context of the HaaST project (Verification of Hard and Soft Real-Time Systems) that is supported by the Dutch Technical Foundation (STW). We thank Pedro D'Argenio (Universidad Cordoba) for his contributions to the Modest modeling language, Theo C. Ruys for sharing his views on the tool implementation and Michel Rosien for his work on an initial state-space generator. William H. Sanders (U. Illinois at Urbana-Champaign) is thanked for facilitating and supporting the interfacing to the Möbius framework.

References

1. Pierre America and Jan J. M. M. Rutten. A layered semantics for a parallel object-oriented language. *Formal Aspects of Computing*, 4(4):376–408, 1992.
2. F. Bause, P. Buchholz, and P. Kemper. A toolbox for functional and quantitative analysis of DEDS. In R. Puigjaner *et al.*, editor, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 1496 of *LNCS*, pages 356–359. Springer-Verlag, 1998.
3. F. Bause, P. Kemper, and P. Kritzinger. Abstract Petri nets notation. *Petri Net Newsletter*, 49:9–27, 1995.
4. A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12th *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer, 1999.
5. M. Bernardo, R. Cleaveland, S. Sims, and W. Stewart. TwoTowers: a tool integrating functional and performance analysis of concurrent systems. In *Formal Description Techniques*, pages 457–467, 1998.
6. M. Bernardo and R. Gorrieri. A tutorial on EMPA: a theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theor. Comp. Sci.*, 202:1–54, 1998.
7. S. Bornot and J. Sifakis. An algebraic framework for urgency. *Inf. and Comp.*, 163(1):172–202, 2000.
8. M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the aldébaran toolset. *Int. J. Softw. Tools for Techn. Transf.*, 1(1/2):166–184, 1997.
9. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Perf. Ev.*, 24(1&2):47–68, 1995.
10. M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *SRDS*, pages 245–253, 1998.

11. Pedro R. D'Argenio, Holger Hermanns, Joost Pieter Katoen, and Ric Klaren. MoDeST – a modelling and description language for stochastic timed systems. In L. de Alfaro and S. Gilmore, editors, *Process Algebra and Probabilistic Methods*, volume 2165 of *LNCS*, pages 87–104, 2001.
12. P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In D. Gries and W.-P. de Roever, editors, *Programming Concepts and Methods*, pages 126–147. Chapman & Hall, 1998.
13. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derasavi, J. Doyle, W.H. Sanders, and P. Webster. The MÖBIUS framework and its implementation. *IEEE Trans. on Softw. Eng.*, 28(10):956–970, 2002.
14. S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation and synthesis. *Proc. of the IEEE*, 85(3):366–390, 1997.
15. S. Frolund and J. Koistinen. Quality-of-service specifications in distributed object systems. *Distr. Sys. Eng.*, 5:179–202, 1998.
16. H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 68–84, 1998.
17. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002.
18. S. Gilmore and J. Hillston. The pepa workbench: a tool to support a process algebra-based approach to performance modelling. In *Computer Performance Evaluation*, volume 794 of *LNCS*, pages 353–368, 1994.
19. H. Hermanns. *Interactive Markov Chains*, volume 2428 of *LNCS*. Springer-Verlag, 2002.
20. H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPTOOL. *Perf. Ev.*, 39(1-4):5–35, 2000.
21. Jane Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.
22. http://www.research.att.com/sw/tools/graphviz/.
23. E.A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002.
24. J. Lilius. The analyzer component framework version 0.1 – a tutorial. HTTP: `http://aiken.cs.abo.fi/acf/ACF.pdf`, Feb 2000. DRAFT: Revison 1.2.
25. Terence Parr and Russell Quong. Antlr: A predicated-ll(k) parser generator. *Journal of Software Practice and Experience*, 25(7):789–810, July 1995.
26. Paul Pettersson and Kim G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.
27. G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.
28. M.L. Puterman. *Markov Decision Processes.* John Wiley & Sons, 1994.
29. C. Rodrigues, J.P. Loyall, and R.E. Schantz. Quality objects (QuO): Adaptive management and control middleware for end-to-end qos. In *OMG's First Workshop on Real-Time and Embedded Distributed Object Computing*, 2000.
30. R.A. Sahner, K.S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems. An Example-Based Approach Using the SHARPE Software Package.* Kluwer, 1996.