Modeling and Verification of Network Protocol Specs using Timed Pi-Calculus

Kamal Barakat, Stefan Kowalewski
Embedded Software Laboratory
RWTH-Aachen University
Aachen, Germany
Email: {barakat,kowalewski}@embedded.rwth-aachen.de

Abstract—We present a toolkit based on our timed extension of
pi-calculus. The calculus is employed as a domain specific lan-
guage for modeling telecommunication protocols with real-time
properties. This model can be simulated, visualized, debugged
and transformed for model-checking using UPPAAL. This paper
introduces the general structure of our toolkit and provides a
unified running example.

Index Terms—timed pi-calculus, simulation, verification,
telecommunication protocols

I. INTRODUCTION

Pi-Calculus [1]–[3] is a process algebraic model that is
specifically tailored for modeling the mobility of concurrent
communicating processes. It employs the concept of link mo-
bility rather than process mobility [4]. The expressiveness and
rigorousness of π-calculus made it popular among scientists
from different application domains [5]–[12].

Modeling and verification of timed concurrent systems are
developed using various approaches, some of which are
synchronous languages [13], [14], timed automata [15]–[18],
Markov models [19], [20], and labeled transition systems
(LTS) [21]. Pi-Calculus, however, provides dynamic channel
and process setup in addition to hierarchical process structures.
Those features are in our opinion necessary to model com-
plex telecommunication protocols. Therefore, we extended the
calculus to model real-time capabilities with minimal impact
on its original congruence arrangements [22], [23]. Although
timers in our extension (π̄'-calculus) are mere input/output
actions, we prioritized them to exploit the analogy to interrupts
and timers in programming languages. Our approach separates
the passage of time from the execution of actions and hence
allows more flexibility in modeling various requirements.

In this paper we present the toolkit we developed based
on the calculus extension in [22], [23]. Our tools provide
simulation, visualization and debugging in addition to model-
checking of telecommunication protocol specs. The motiva-
tion is to support engineers and protocol architects that are
centered with the development of these specifications with an
integrated solution.

We start by briefly introducing the syntax of π-calculus
and our extension in Section II. The modeling platform
is presented in Section III. In Section IV we present our compiler.
In Section V, we describe the simulation engine. Section VI
outlines the visualization and debugging tool. Finally, Section
VII describes the transformation utility into UPPAAL. We
justify our design decisions and provide examples throughout
the paper.

II. INTRODUCTION TO π̄'-CALCULUS

We start with introducing the syntax. It is parametrized by
a set of agent identifiers, A, and a set of names, N. The
latter serves as both communication channels and data to be
transmitted along them. The set of process expressions is given
by the following context-free grammar.

\[ P ::= 0 \] empty process
\[ \alpha.P \] prefixing
\[ P + P \] choice
\[ P \parallel P \] parallel product
\[ (if \ x = y) P \] match
\[ (if \ x \neq y) P \] mismatch
\[ new \ x \ P \] restriction
\[ A\langle y_1, \ldots, y_n \rangle \] agent call

where \( A \in \mathcal{A} \) and \( x, y \in \mathcal{N} \). Here \( \alpha \) represents an action prefix, determined by

\[ \alpha ::= x(y) \mid \pi(x(y)) \mid \tau \]

where \( x, y \in \mathcal{N} \). Each agent identifier has to be defined by an equation of the form
\( A(x_1, \ldots, x_n) ::= P_A \) with \( x_i \neq x_j \) for \( i \neq j \). The empty process, 0, cannot perform any actions. A prefixed process, \( \alpha.P \), can perform action \( \alpha \) and then behaves as \( P \). Here \( x(y) \) denotes the input of name \( y \) along
channel \( x \), while \( \pi(x) \) stands for the complementary output
operation. Moreover, \( \tau \) denotes the unobservable (or silent)
action. The summation construct + represents nondeterminism
in the model where a process has several execution paths to
choose from. The parallel composition operator \( \parallel \) is used
to compose a process from several subprocesses that are
executing concurrently. In a restriction \( new \ x \ P \), the scope
of name \( x \) is limited to process \( P \), thus binding \( x \). Names
that appear in the argument positions of input actions or agent
definitions are also bound, every other occurrence of a name is
free. A (mis-)match behaves as the argument process if the
given (in-)equation on names is fulfilled. Otherwise it does
nothing. Finally agent calls can be used to specify recursive
behavior of processes. Sometimes we will use prefixes of the form \( x \) or \( \pi \) to denote a pure synchronization operation without name passing. Moreover, we use \( \alpha \) as an abbreviation for the process \( \alpha.0 \).

### A. Mobility support

Mobility in \( \pi \)-calculus is modeled using dynamic channel setup. Assume the simplified example depicted in Figure 1:

\[
T(f) := \pi(f).T(f) + f.(data).T(f), \quad (1)
\]
\[
M := c(f).\overline{f}(data).M, \quad (2)
\]
\[
SYS := M \parallel T_1(f_1) \parallel T_2(f_2), \quad (3)
\]

where the mobile set \( M \) moves freely in the domain of two cells \( T_1 \) and \( T_2 \). The mobile set constantly measures the signal and detects the available cell on a global channel \( c \). Each cell process is parameterized with a particular channel frequency \( f_1 \) and \( f_2 \) which it advertises on \( c \). If \( M \) is in the domain of a cell it uses the advertised frequency for sending data packets.

### B. Timed extension

A timer is a synchronization channel with a clock process. There are three types of timers in our timed extension of \( \pi \)-calculus. A passive timer, such as \( t_z \), executes instantly when it becomes unguarded causing its subscript parameter(s), \( z \), to assume the current value of time. Active timers, such as \( t^{x+y}_z \), execute if they become unguarded and if the current time reaches the value \( x+y \). If an active timer also contains a passive component, such as \( t^{x+y}_z \), it is then called bidirectional.

### III. The modeling platform

We use plain \LaTeX{} documents as a platform for producing specifications of telecommunication protocols in \( \pi \)-calculus syntax. The specification can contain text, graphics, tables, etc. in addition to \( \pi \)-calculus formulae that formally describe the protocol. To distinguish formulae that contain \( \pi \)-calculus specifications from other mathematical formulae in the specification, we redefined many math environments and commands in order to give them different names without changing their functionality. This allows our compiler to filter \( \pi \)-calculus content out of the specification document for post-processing. The \LaTeX{} compiler can still produce output in the usual way. Another advantage of using \( \pi \)-calculus for producing specifications is the improvement of their expressiveness. Misinterpreting text specifications is a common problem. The robust math-like syntax of \( \pi \)-calculus precisely models the behavior of protocols and reduces ambiguity. Figure 2 shows component dependency of our tool chain. The bisimulation block is planned for future work.

The choice of \LaTeX{} is beneficial because of its free availability and widespread use. Furthermore, formulae are produced in plain text, which simplifies post-processing the document after extracting those formulae.

### IV. The \( \pi \)-calculus compiler

The \( \pi \)-calculus compiler is the basic component of our SimPiCal toolkit. It is developed using ANTLR [24], [25] which employs LL grammar. The compiler accepts \LaTeX{} documents with specifications written in \( \pi \)-calculus syntax. The compiler extracts the relevant content of the \LaTeX{} specification by recognizing the redefined math environments. The rest of the specification is irrelevant to the compiler. The output is an ASCII metadata file in the form of semi-machine language. This metadata is the input of the simulation and visualization tools. The names in the \( \pi \)-calculus spec are mapped to unique numerical identifiers. The mapping is then kept in a hash file. This simplifies many aspects of simulation such as parameter passing. It also simplifies some aspects of the transformation process into the UPPAAL environment.

#### A. Metadata Structure

We broke the \( \pi \)-calculus syntax into the smallest basic components possible; type definition, process declaration, parameterized process instantiation/replication and process transitions. Alpha-conversion and restriction of names are also implemented. These components represent the complete classic \( \pi \)-calculus footprint. We also implemented timers to incorporate our extension to \( \pi \)-calculus. The following process declaration for example breaks down into two major parts; a type definition from the right side of the equation and a process declaration of this type:

\[
Q := t^{300ms}_x.\pi.Q + r.t_x.Q. \quad (4)
\]

The type itself comprises transitions that are created from the sequential compositions \( t^{300ms}_x.\pi \) and \( r.t_x \). The action
prefix $t_x^{300ms}$ is an bidirectional timer that executes at time $x + 300ms$ and saves the execution time to its subscript argument(s), while $t_x$ is a passive timer which only saves its execution time to $x$. $Q$ can execute $r$ as long as the current time did not reach the $x + 300ms$ threshold. If $r$ is chosen, $t_x$ will subsequently be executed and $Q$ will then re-initialize. However, if the threshold $x + 300ms$ is reached before executing $r$, the timer $t_x^{300ms}$ will fire forcing $Q$ to take the alternative execution path. In this path, $Q$ synchronizes over channel $a$ with some other process and then re-initializes. This choice construct is beneficial for modeling time-outs. Start and transient states are automatically created by the compiler. The generated metadata and hash table for this process are in Figure 3.

The simulator consists of two major parts; the metadata interpreter and the simulation engine. The interpreter builds object definitions from the numeric metadata file. It also creates lists of all process declarations and their parameters in order to perform name conversion when parameter passing occurs. These object declarations are the input of the simulation engine which starts by executing the system statement. Therefore, a $\pi$-calculus specification that is intended for simulation must contain a system statement such as (7). The system statement declares the layout of the whole system at kick-off time. This layout consists of previously defined processes and types. The following equations represent a complete system definition:

$$P := a.t_y.P + t_y^{500ms}.\pi.P', \quad (5)$$
$$P' := a.P + t_y^{300ms}.0, \quad (6)$$
$$SYS := P || Q, \quad (7)$$

where $Q$ is the process declared in (4). The whole system represents a fictitious liveness protocol where $Q$ reports to $P$ each 300ms on channel $a$. If $Q$ fails to report within time, $P$ will reset it over channel $r$ after 500ms, and then wait 300ms before terminating if failure persists.

### A. The simulation engine

The simulation engine operates in two modes; real-time and simulation. In the simulation mode, time advances only if all processes are blocked waiting for input/output synchronization or for timer events. If this condition holds the time will advance to the earliest scheduled timer and the owning process is notified from sleep. In this mode, input/output actions do not consume time and hence it is not Zeno free. In the real-time mode on the other hand, the engine uses timer APIs of the operating system to schedule the timers of the $\pi$-calculus model. Therefore, it is not possible to observe the Zeno behavior in this mode.

The simulation engine starts with instantiating all processes that are referenced in the system statement, i.e. $P$ and $Q$. All subsequent process instantiations, if any, are performed at particular locations and times as specified in the currently active processes. Active processes share a static list in which every process registers its currently enabled action prefixes. This list enables the synchronization and parameter passing between concurrent processes.

### B. Simulation output

The simulation engine produces user friendly output logs. It uses the original names of processes, types and action prefixes from the hash file to facilitate log reviewing. The log is divided into four columns; current time, line type (info, debug, error, etc.), an optional process ID and finally the log message. Figure 4 is the simulation log of the system declared in (7). Because of page limit, we only show the simulation log until the first liveness signal has arrived.

Our simulator enables the utilization of the full semantics of $\pi$-calculus to investigate particular properties of the specified system. The logs can be used to generate data for static analysis too. The simulator is developed using Visual C++ and it is engineered to make porting the software to other platforms as easy as possible.
VI. VISUALIZING AND DEBUGGING

We developed another tool [26] at our institute to visualize and debug \( \pi^- \)-calculus specifications. Although \( \pi^- \)-calculus has strong expressiveness because of its math-like syntax, it is still desirable to visualize complex specifications. The visualizer has two characteristics. It is firstly a document transformation utility that produces a HTML document from the original \( \LaTeX \) spec. Secondly, it allows the user to interactively step through the system and watch how processes interact and evolve. This feature represents a debugging facility. This is achieved by integrating a java applet in the HTML document which decodes the metadata, renders it in different views and steps through a visual representation of the compiled model. See Figure 5 for a view of the liveness Protocol introduced in (7).

A. The applet

The HTML document displays all the content of the original \( \LaTeX \) spec and embeds an instance of the Java visualization applet at the locations of \( \pi^- \)-calculus equations. The applet allows switching between the sequence diagram view, the automaton view and the equation view.

- The sequence diagram allows the user to step through the process interactions forwards and backwards and to choose the step size in terms of the number of interactions. If the specification contains nondeterminism, it is possible to choose between multiple action prefix synchronization pairs. In Figure 5 we see how \( P \) is forced by the user to take the timeout path instead of synchronizing with \( Q \) over channel \( a \). It is also possible to enter new values to the parameters of action prefixes and even change the prefixes themselves.
- Each process is internally modeled as an automaton, therefore it is possible to view it as such and to pinpoint its current state in the automaton view.
- It is desirable to switch back to the equations to view the formal definition of the system. The equation view displays the original equations of the \( \LaTeX \) spec that were introduced at the location of current instance of the applet.

Because of parameter passing and alpha-conversion operations, the names in the specification assume new values which could be numeric or other new names. The applet always displays the valuation of names under the scope of each process and allows manipulating their values. It also displays the value of the current time.

We chose HTML to make this tool supported by any Java-enabled web browser. The document transformation utility is also developed in Java to support as many platforms as possible.

VII. MODEL-CHECKING WITH UPPAAL

The XML model file of the UPPAAL platform is directly generated from the abstract syntax tree (AST) of the compiler without the need for metadata. Figure 2 shows that the UPPAAL transformation part is actually an extension to the compiler itself.

A. The transformation

Types are transformed to templates. Superscripts of active timers are boolean expressions that specify when the timer fires and hence are also transformed to guards. The names in the \( \LaTeX \) spec are transformed into UPPAAL variables. The subscript of a passive timer such as that in (4) are transformed to assignment labels. The names that appear in those subscripts assume the value of the current time. Assignment labels are also used to implement parameter passing between communicating process pairs. This is done using a shared array of integers that is used as a clipboard. A send transition with parameters will save its parameters in the clipboard using its assignment label. The corresponding input transition will use its assignment label to read the parameters from clipboard. Of course, an arity check is done to insure that only input/output channels of the same parameter arity synchronize with each other. Each process maintains its own name valuation using a local integer array. This simulates alpha-conversion in \( \pi^- \)-calculus and preserves scoping of name valuation. UPPAAL allows prioritizing transitions of choice. We use this feature to model timer transitions because timers in \( \pi^- \)-calculus have higher priority than input/output action prefixes. Constraints in \( \pi^- \)-calculus are conditions on action prefixes. Consider the following example:

\[
R(x) := (i f x \leq 3) \pi(x).0 + (i f x > 3) b(x).0
\]

(8)

The process \( R \) has a nondeterministic choice between sending the passed value \( x \) over channel \( a \) or channel \( b \). However, these two actions are prefixed with constraints that enable one of them according to the value of \( x \). Therefore constraints in \( \pi^- \)-calculus are transformed to guards in UPPAAL.

B. Compatibility issues

In UPPAAL however, it is not possible to query for deadlocks if priority channels were used. It is also not possible to assign the value of a clock or the difference between clocks to an integer variable because clocks are evaluated symbolically. On the other hand, constraints in \( \pi^- \)-calculus allow arithmetic operations on clock values. Our transformation utility therefore supports two formats:

- A format that does not create clocks in UPPAAL and maintains the current time as an integer value (compatibility mode),
- and a format that uses UPPAAL clocks but is therefore not fully compatible with \( \pi^- \)-calculus.

The drawback of the first approach is the extra state space and the limited verification time span that cannot exceed the size of integers. Our compiler models time with millisecond granularity. This will give the user a maximum of 32769mSec verification span in the 32bit version of UPPAAL. The second approach creates UPPAAL models that might not be executable by UPPAAL if the original model uses clock values in arithmetic expressions. Process instantiation in \( \pi^- \)-calculus
can happen at any time, while automata in UPPAAL are instantiated once at system kick-off. We intend to solve this issue by modifying the templates. A unique kick-off transition will be inserted before the init state. All processes in the UPPAAL model will still be simultaneously instantiated at the beginning. However, processes that are instantiated later during the π-calculus scenario will only be active after their kick-off transition is synchronized with the calling process. Replication is also not possible in UPPAAL but we can create an array of automata of the replicated template. This limits system verification w.r.t. replication to the number of automata in the array.

C. Verification of the liveness protocol

Figure 6 shows the liveness protocol after being transformed into UPPAAL in the compatibility mode. In the compatibility mode, an explicit clock template is automatically created by our compiler and a clock process is added to the system definition. The clock process increments the time and synchronizes it with P and Q. In P we have the transition S3 \xrightarrow{\pi} S4. We used UPPAAL’s requirement specification language to execute the query:

E<> P.S3

which evaluates to true. This shows that P eventually tries to reset Q. However, the query:

A[] P.S3 imply P.S4

fails which indicates that Q possibly reaches a state where it cannot synchronize with P over r. This happens after the timer in Q has fired, which puts Q in state R2. At this state Q looses its commitment to r and hence, it cannot be reseted by P.

π-Calculus is based on automata theory. Therefore, we believe that it is beneficiary to use a state-machine based approach to verify our π-calculus models. This approach is also widely adopted, thus, the use of UPPAAL offer more flexibility for the users.

VIII. RELATED WORK

A complete overview of formal verification and modeling approaches is not possible because of page limit (see some at [27].) We try here to cover the most popular and relevant ones and reflect on our calculus whenever applicable.

A recent contribution to timed π-calculus is the work of Jin et al. [7]. In the extended syntax there are new timed operators for expressing the passage of time and for modeling choice upon threshold crossing. Due to modeling time passing on the clock-tick level, all transition rules of π-calculus were revisited to provide delayed versions of each rule to represent how processes behave when time passes. This multiplies the total operational semantics of the calculus. We perceive a superfluity in introducing the threshold operator in the presence of the original choice operator “+” in π-calculus. We use a combination of timers and the choice operator to express time-out events as in (4).

The approach in [8] provides a type system for specifying locations which restrict services that agents can use during communication. TDπ-calculus combines the execution of actions with the passage of time in dedicated time operators. Message exchange occurs at defined locations over channels at these locations if the timer was not exceeded. The process then proceeds as declared in the operator. If the time-out value is exceeded, the process takes an alternative execution path. Our calculus can provide the same timed behavior using the timer and choice operators. The timed semantics of both approaches are equivalent. However, our calculus retains the advantage of separating the passage of time from the execution of actions, and thus provides more flexibility in specifying system behavior; our calculus allows state transitions based on pure time events.

Timed spi-calculus [6] approaches timing-out in a manner similar to [8]. The timed operational semantic is defined as a reduction relation on states with global time variable and a binder on all free names in these states. The operator allows the definition of a set of correspondences that can be terminated in the particular run, and also defines the process that remains to be executed. This reduction rule allows only
process replication and output actions to survive a period (or epoch) to the next one. All other syntactic forms expire with a clock-tick and degenerate to the null-process. Timed spi-calculus combines the notion of time passing and action execution and conveniently tailors the behavior of particular syntactic components of the calculus for modeling security and cryptographic issues, e.g. the asymmetrical timed behavior of input and output actions. This makes the calculus, however, inflexible in treating issues outside that domain.

Another approach in the domain of cryptography is the work of Blanchet [11]. A translation tool from applied π-calculus is provided to simplify the issue of creating Prolog rules that describe the cryptographic protocol, the attacker’s abilities and the facts about the initial knowledge of the attacker. A specific verification algorithm is also developed because ordinary Prolog systems do not terminate. Kremer and Ryan [12] use the same facilities developed in [11] to analyze a voting protocol. They exploit static and labeled observational equivalence to assess the knowledge processes make publicly available and hence be able to verify the privacy property of the protocol. However, this approach does not model timed properties of such protocols.

Berger and Yoshida [5] define types on channels which range in one dimension over being linear or affine and whether they are replicated or not. The other dimension depends on whether these names are input or output. From this type classification, they define constraints on names. The type system is then used to define message-loss and timers on process and network level. To model message-loss correctly, they solve the nondeterminism problem by using probabilistic automata. They proceed then by defining the impact of this nondeterminism on bisimulation by introducing timed approximate bisimulation. Nonetheless the analogy between [5] and [7] with respect to timed behavior becomes clear by considering their timeout and timestepper arrangements.

In the work of Lee and Žic [9] the time-out operator $P^T Q$ forces a process $P$ to proceed as $Q$ at time $t$ if no enabled action of $P$ was fired before $t$. Similar to previously mentioned approaches, a delay operator is defined. This operator takes place in the transition system of processes. Lee and Žic differentiate internal or invisible actions from those visible to the external observer by calling them uncontrollable and controllable respectively. In this work time is not allowed to pass during action execution and nondeterministic choices are only made by ordinary action prefixes. This means that the passage of time does not obscure the execution of an enabled action in a process. As in [7] the operational semantics explode due to different time passing properties of input/output actions and $\tau$ and their effect on combinatorial operators such as summation and parallel composition.

Another Prolog based approach is the work of Gilbert and Palamidessi [4]. In contrast to link mobility in π-calculus, they adopt the process mobility approach in which an agent is either copied or relocated to another location in a hierarchical structure. The implemented agent mobility allows only the local context to move with the agent but not its outside links. The context is migrated by adapting the concept of constraint store, which allows correct handling of visibility issues of local and global variables and procedures. An interpreter in Prolog is developed based on the operational semantics of agent migration, which enables the verification of migration scenarios. This model however, lacks the specification of timed requirements.

Kuttler et al. [10] introduce a variant of stochastic π-calculus. They model the stochastic properties using input patterns of functions with fixed arity. Inheritance is defined over classes of objects of the calculus in a way that substitutes recursive calls of the parent with calls of the child class, which facilitates modeling chemical reactions from one molecular specie to another. The π-calculus model is associated with a continuous-time Markov chain (CTMC) in order to draw the rates of each transition. Delay times are drawn from an exponential distribution with the rate of each transition at each state. They semantically distinguish time consuming transitions from non-time consuming ones and prioritize the latter. Our approach in contrast, prioritizes timed actions over ordinary input/output actions.

UPPAAL [15], [16] is the leading model checker for timed
automata. It enables the verification of bounded liveness properties of real time systems. It is well established in the academia as a reference because of its friendly GUI, simulation, verification and diagnostic utilities. Reachability checks are performed on combinations of control-nodes and constraints on clocks and integer variables. The model is a network of timed automata with guards on transitions. Automata can synchronize transitions using bilateral or broadcast channels, which could also be urgent. It is also possible to prioritize particular channels or states too.

RED [17] also employs timed automata for modeling and verification. The contribution here is the representation of clock and variable constraints using difference-reduced closure (DRC) format and their combination with binary decision diagrams (BDDs) into one data structure to enable full symbolic manipulation. A mapping for boolean variables is also defined in order to represent root variable values. This data structure fusion reduces the state space in comparison with other clock difference diagram (CDD) approaches.

The concept of timed hybrid automata as proposed by Lynch et al. [18] defines trajectories as the evolution of a collection of variables over an interval of time, and they are the set of functions that map a closed interval to the set of valuation functions. Hybrid automata (HA) are comparable if they have the same external interface (i.e., external variables and actions). They consider a simulation relation between different HAs if one implements the other in the sense of inclusion of sets of execution traces. Then they declare the composition and hiding operations on HAs, which in turn, respect simulation. The distinction between input and output variables enables the definition of hybrid I/O automata (HIOA). The receptiveness property defined on HIOAs allows the time to always pass for any sequence of input actions. The presented structures in this work are interesting because they correspond to native capabilities in π-calculus. However, hybrid automata do not posses the hierarchical structuring capabilities and the dynamic channel setup of π-calculus.

Prism [19], [20] is a probabilistic systems analysis tool that utilizes Markov chains and decision processes in addition to probabilistic timed automata (PTA) and priced PTA (PPTA) for its models. PRISM model-checks these systems against probabilistic temporal logic specifications. Markov based models are checked using several engines of different capacity and speed properties. Recent versions of PRISM provide PTA/PPTAs verification using quantitative abstraction refinement or digital clocks to answer queries about minimum/maximum probability of reaching a target state within time bounds or the minimum/maximum expected reward until a target is reached.

PAT [21] is a toolkit for system analysis under fairness that is defined over labeled transition systems (LTS). This contribution introduces verification of strong-global fairness in addition to strong-local and weak fairness. The importance of this work is the possibility to associate fairness to parts of the system or associate different parts with different fairness constraints by annotating particular actions and states. This in effect enables partial order reductions to actions that are irrelevant to the fairness annotation.

Esterel [14] is an imperative language which belongs to the family of synchronous languages [13]. This language is used to model synchronous threads using instantaneous communications and decisions. Threads communicate using signals that are emitted and received at interaction points. Statement predicates declare the behavior of the system by reacting or ignoring a signal. The syntax in Esterel contains flow control, parallel and sequential composition of actions. Esterel is widely used in modeling embedded systems. Compilers of Esterel utilized various techniques starting with automaton based compiling, to digital logic, symbolic state space traversal, and concurrent flow diagrams. There are automata based and symbolic BDD based model checkers available for Esterel.

IX. CONCLUSION AND FUTURE WORK

We developed a tool chain based on our timed extension of π-calculus in order to support engineers during the development of telecommunication protocol specs. Our contribution to the state-of-the-art can be summarized in the following points:

• We introduced a platform for producing telecommunication specs using the semi-mathematic syntax of πτ-calculus. This helps eliminate the problem of specification misinterpretation.
• These πτ-calculus specs can be compiled into metadata for post-processing.
• The simulation engine post-processes the metadata and produces simulation logs. These log files are helpful assets for analyzing the specs.
• The visualization and debugging tool supports the verification of complex models.
• Model-checking the specification after transforming it into UPPAAL enables answering critical questions about its correctness.

The ability to simulate, visualize, debug and model-check telecommunication protocol specs can be compiled into metadata for post-processing.

We are currently finalizing our transformation utility to UPPAAL. We are also working on a bisimulation engine that is able to verify whether two different πτ-calculus specifications are timely equivalent.

ACKNOWLEDGMENT

This work was funded by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

REFERENCES


