# Global Scheduler Properties derived from Local Restrictions

Thomas Arts*
IT-university
Box 8718, 402 75 Göteborg, Sweden

Juan José Sánchez Penas
LFCIA, Computer Science Department,
University of Corunha
Campus de Elvinha S/N. 15071, A Corunha,
Spain
juanjo@lfcia.org

## ABSTRACT

The VoDka server is a video-on-demand system for a Spanish cable company. We look at the distributed scheduler of this system. This scheduler enables that whenever a user agent is asking for a certain movie, this request is transferred through the system and a set of possible play-back qualities is returned to the agent. In case of a non-empty set, the agent selects one and the movie is streamed to the user.

The storage subsystem of the server is composed by a hierarchy of different storage systems, i.e. disks, CD players or tapes. These devices all have restrictions of which the process controlling the device is aware of. A second layer of processes controls a set of devices in one machine and has restrictions, for example, the bandwidth of its connection. A third layer may be further out in the network and serve as a cache to store more popular movies.

Every process in the scheduler of the system has a function determining local restrictions, given the configuration and present state of the system. We have built a tool to construct complete models of several configurations. With techniques from the area of formal methods (in particular model checking) these models are used to determine global properties of the system, such as the maximum number of a certain class of movies that can be served in parallel.

## 1. INTRODUCTION

In a design, either concurrent or distributed, where one has many processes that steer a certain functionality, one often finds global properties of the system hidden in several local properties of the running processes. This article proposes a method to reveal these global properties by a kind of exhaustive simulation of the system.

We demonstrate the method by means of a hierarchical distributed video-on-demand server [5, 4] developed using Erlang/OTP [1]. We concentrate on the implementation of the scheduler, where there is neither global state nor a global decision module. The local properties are restrictions (on bandwidth and number of connections of disk drives, CD players, tape storage devices and such), local scheduling functions (filtering and admission policies) and cost related functions (state of the component and resources still available).

Given only these local restrictions, and the rest of the configuration of the system (number of levels and components in each level), it is far from obvious to extract information about the behavior and performance of the system. Answering questions such as how many users can watch 'Star Wars' at the same time, is virtually impossible without building the actual configuration and testing this. Answers to such questions, however, are what both the operator of the video-on-demand server and the designers of the system are interested in. The former want to obtain information about the capacity of the system, and the later are more interested in knowing how the different distributed properties of the system influence its performance, in order to be able to know how to improve it (redesign and reconfiguration of the scheduler).

Many global properties of the system can be determined by testing, but testing all possible scenarios of users that request a movie is rather expensive. Moreover, one tests a certain configuration. Performing experiments with new drives, faster network connections and all that, increases the costs even more.

We used techniques from the area of formal methods to obtain a model of the scheduler from the Erlang source code of the system. Formal methods, used in a rather unconventional way, give us the possibility to construct a graph that represents the system load when users request all possible sequences of classes of movies. We developed tools to automatically translate the Erlang code into the process algebra $\mu$CRL [8], as an intermediate step for generating the performance graph of the system. This graph is a reduction of the state space generated (by existing tools [13, 14]) from the $\mu$CRL specification.

In this graph the failures of requests are visible and therefore, the shortest path to a failure. This answers the ques-

*Thomas was affiliated with Ericsson when the work described in this article was carried out. Juan José was during that time staying for a three months visit at Ericsson in Stockholm with an FPU grant from the Ministerio de Educación y Cultura of Spain.


*ACM SIGPLAN Erlang Workshop* '02 Pittsburg, PA USA

tion on how many users are guaranteed to be able to be served in parallel. Other questions, such as 'How many people can watch the movie A such that the system can still serve B?' or 'Where should we store the movie A for being able to serve it to N users?', can also be expressed as properties of the graph.

We designed a user interface to guide the whole process: choosing the parameters of the configuration, generating the model, constructing the graph and translating human understandable global properties of the system into a temporal logic formula. This formula is checked by model checking techniques using the Caesar/Aldebaran Development Package [7] (using the formula as a declarative way of asking for knowledge of the system and the checking techniques primarily as efficient graph search techniques).

The article is organized as follows. In Sect. 2 we present the scheduler architecture of the video-on-demand server. In Sect. 3 our methodology of verifying global properties of this scheduler is explained and described in detail. We perform three major steps from Erlang source code to automatically answering global properties of the system. These three steps are described in Sect. 3.1-3.3. We conclude in Sect. 4 with some remarks on the performance of the tools we used/developed and indicate what problems are open for future research.

## 2. THE VODKA SCHEDULER

The VoDka system is a hierarchical distributed multimedia server. By using a multimedia client, the user can request from the system a *media object (MO)* in several possible streaming protocols. The main goals during the system development were to create a low cost, scalable, very flexible and adaptable (to the underlying network architecture) solution, besides of the traditional requirements for this kind of systems, such as fault tolerance, massive storage capacity, and being able to serve high bandwidth objects to a a high number of concurrent users.

The system's flexible architecture is based on a hierarchy of specialized levels that can be combined in different ways, depending on the needs for a given deployment of the server. A common configuration of the system architecture would be: a massive *storage level*; one or more *cache levels*, that are going to reduce the performance requirements (i.e. the response time or the bandwidth) of the lower levels; and a *streaming level*, that implements the protocol adaptation between the server and the user client. The software of the system has been developed using Erlang/OTP, and it is deployed over an architecture of GNU/Linux based clusters of commodity computers.

In Fig. 1 the general architecture of the system for a linear configuration is shown. The boxes in this figure correspond to an Erlang process. Each of the levels is composed by a set of software components (most of them are Erlang *gen_server* processes) with a standard API. On top of the generic server based architecture, a supervision tree is constructed for providing fault tolerance. The system can roughly be divided in three levels: *storage level*, *cache level* and *streaming level*. Any *storage level* is composed by a *storage scheduler* and a hierarchy of storage devices grouped by one or more *storage groups*. The storage level is connected to the *streaming level*, but one or more *cache levels* may be in between them. The structure of the cache level is similar to that of a storage level; only logically they differ, since media objects are dynamically copied to it and removed after use. The *streaming level* has a hierarchy of components implementing the adaptation to the streaming protocols accepted by the system (HTTP, RTP, H.263, etc.). The processes in the streaming level create and supervise all the processes needed for performing the actual transmission of data through the system.

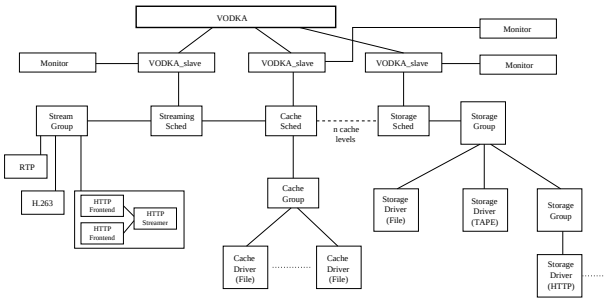


**Figure 1: Configuration example for the architecture of the video-on-demand system**

All the processes of the system have local restrictions, cost functions and decision algorithms, and all these *configuration* values are going to determine the distributed scheduling of the multimedia server.

Whenever a user requests a given MO with a concrete quality (bandwidth), this request is received in the streaming level and propagated trough all the levels of the system. If the MO can be provided by a given level, because the object is stored there and it has the resources left to provide it, then this information is returned. The scheduler is in charge of elaborating a list of candidate providers, with an associated *cost*, that is going to be sent back to the upper levels of the hierarchy. Finally, after filtering the options in the different levels on the way back to the user, either a *fail* or the opportunity to *play* the MO is replied.

In this paper a method is described to automatically derive the global scheduling performance properties of the system from the local restrictions on the scheduling subsystems. We concentrate on configurations of VoDka in which we have one streaming level and one storage level without any cache level. One may argue that a cache level behaves like a storage level in our performance analysis, since we consider in the storage level all possible distributions of the media objects over the devices (i.e., also copies of the same object on multiple devices). For an average-case scenario, the dynamic behavior of the cache may become interesting, but for a worst-case scenario, where all users start asking for a set of movies at the same time, the cache can be seen as static storage.

## 3. FROM LOCAL RESTRICTIONS TO GLOBAL PROPERTIES

The goal of our work is to implement a tool that analyzes an Erlang system in order to obtain knowledge on the behavior of the system. In particular we aim to derive information that is difficult or expensive to obtain by testing. In the case-study at hand, we are interested in the performance of several configurations of the video server, without actually building all possible configurations. Moreover, we would like to obtain some insight in the behavior, such that

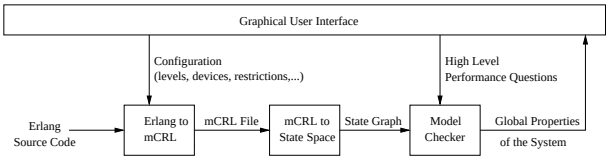


**Figure 2: Proposed three steps methodology: from Erlang to global properties**

we get an idea on how to improve the software.

The methodology we use is based on the principle of generating the full state space of the system, i.e., we generate a graph where the vertices represent all possible states of the system and the edges express which events cause the system to go from one state to the other.

We want to obtain this state space directly from the source code of the system. In more classical approaches in literature one often uses a formal specification of a system, normally a manually constructed abstraction of the system. The advantage of the use of Erlang is that we have a rather high-level of abstraction already, such that we can use the source code as our starting point. However, even the Erlang source code would contain too many details to make it feasible to generate the full state space. We really build upon the existence of design patterns, like the supervision tree and generic servers that hide a lot of the details.

We explore in particular the fact that the system consists of generic servers. Every generic server has an explicit state defined: the state is passed as a parameter in the call-back functions. Process state information, like the actual memory on the heap, the content of the message buffer and such is hidden for free. The generic server is an abstraction for debug features that production code typically is attached with. It is an abstraction for the handling of shutdown, code replacement and such. By concentrating on the call-back functions for handling messages, one concentrates on the basic functionality of the server and abstracts from a lot of standard features. The full state space of a system consists therefore of the combination of all possible states that can occur in the state parameter of a call-back function. This is a real reduction and in many practical examples it is a finite state space.

The events in the system are the messages the particular servers receive and reply/send. In our example the messages are requests for media objects that are passed from one level to the other, and a list of choices propagated in return. As long as the list of media objects is finite, this results in a finitely branching graph. However, in case of a realistic number of movies, this would be an enormous graph, impractical for our purposes. We realized that the level of detail on the specific movie is unimportant for the analysis of the system. What is important is that it is a media object that is stored on the first disk, or that it is an object stored on both disk and tape. Hence, we look at configurations of the system in which we instantiate our storage devices with abstract objects $m1$, $m2$, *etc.* Typically, we have one object per combination of devices. Thus, there is one object that is both on tape and on disk; one object only on tape, one only on disk, *etc.* The real distribution of movies is a function from media objects to abstract media objects. The user of our tools can therefore still ask the question: "Is it possible

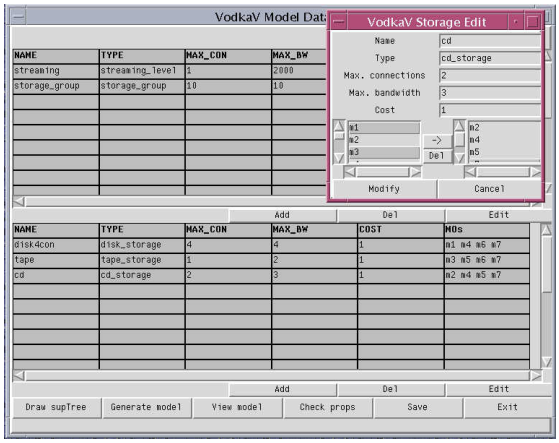


**Figure 3: The graphical user interface**

to have 30 users watch 'Star Wars' at the same time?". The abstract object is computed and the question translated.

As a real advantage of these abstract media objects, we get for free that we can answer the user: "No this is not possible, but if you put one extra copy on CD player 4, then this is possible". We just try to answer the question for all possible abstract media objects and determine the difference between the given abstract object and the abstract object for which it succeeds.

The analysis tool consists of three parts that are connected by a graphical user interface (GUI). First, the user interface lets the user select a configuration (cf. Fig 3), i.e. the levels in the system, the storage devices with their limitations and the media objects stored on each device. The interface then calls our translation tool that translates the system with this particular configuration to a $\mu$CRL specification (Sect. 3.1). Second, the user interface activates tools to efficiently generate the state space and reducing it (Sect. 3.2). Third, several properties are presented in the user interface that can be checked automatically using the underlying model checker and some gluing software (Sect. 3.3).

## 3.1 Erlang to $\mu$CRL

In order to generate a state space for an Erlang system, one needs to have a state space generation tool, i.e., a tool that can produce all possible runs of the system. Instead of creating such ourselves, we decided to use a tool that was demonstrated to be efficient for a language reasonably close to Erlang. These tools typically exist for specification languages like process algebras, such as LOTOS [12] and $\mu$CRL.

The advantage of a process algebra language over a language like Promela [11] is that the translation from Erlang to the process algebra is easier to establish: the data part of the process algebras is based on similar rewriting semantics as we see in functional languages. Moreover, in the process algebra attempt one is free to use unbounded data structures, like lists and natural numbers, in the specification. If the actual use of these data structures in the program turns out to have a bounded size, then one is able to generate a finite state space. However, one need not decide on beforehand what the maximal size of these data structures is. For Erlang programs it is quite often the case that lists have a

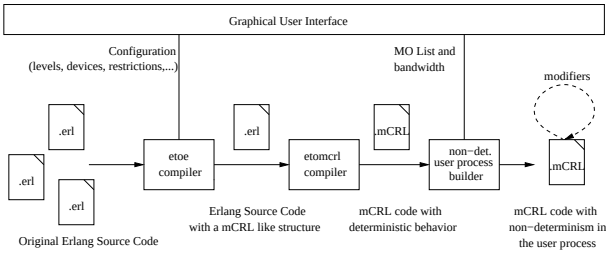


**Figure 4: From Erlang to $\mu$CRL**

fixed maximum length during all possible executions of the program, but that this length differs for different configurations in which the program is used. This fits seamlessly in the process algebra framework.

In order to use the state space generation tool for $\mu$CRL, a 'compiler' had been developed in an earlier project [2]. At the time this compiler was developed, we chose for $\mu$CRL over LOTOS, because we had more experience with it [3] and we were impressed by the little redundance it produced in the generation of the state space. The language $\mu$CRL and LOTOS are very similar though. The compiler has been constructed in such a way that it should be rather easy to create a LOTOS back-end for it.

In Fig. 4 the translation from Erlang to $\mu$CRL is sketched. The first two steps are performed with the compiler that was developed for the earlier project. The last step is a small modification to the created $\mu$CRL code to obtain a non-deterministic *user process*. Here we summarize the basic principles of the compiler and explain the last step in more detail.

Serious Erlang software is built upon the OTP design principles. Large Erlang systems have about eighty percent of their processes implemented as servers, using the *generic server* design pattern. All processes are put together in a *supervision tree*. Other frequently occurring processes are using the generic finite state machine or the event handler design pattern. The nice thing from a verification point of view is that the use of these design patterns make the state of the process explicit (in an argument of the function calls). Moreover, a lot of details can be omitted in the call-back functions, since they are provided once and for all in the generic implementation. For example, debug features are provided in the design pattern and need not be implemented in the call-back module; standard error handling is provided, and so on. Basically the few hundred lines of code that the generic server consists of would more or less be reused in all implementations, if such a generic solution was not provided. The semantics of the generic servers is embedded in the translation tool, such that a smarter translation can be produced than when we would analyze arbitrary Erlang code.

The Erlang to $\mu$CRL compiler is meant for systems that consists of servers that are implemented using the generic server design pattern and clients (that might be servers as well) that are restricted to communicate via the generic server API (i.e., *gen_server:call* etc.).

A set of Erlang modules implementing the servers and clients forms the input of our compiler. In Erlang the processes implemented by these modules can be dynamically created and normally a supervision tree will be used to initiate that. In $\mu$CRL we have to statically assign the processes that we consider, thus we need to know them on beforehand. This is achieved by demanding the natural design choice that all servers and clients are a child in a supervision tree that has to be provided as input as well. The first task for the compiler is to symbolically evaluate the supervision tree for a given set of arguments (determining the configuration) to find the processes and their initial arguments that are used in the system.

After computing the set of processes in the system, every module implementing one of these processes is translated. These processes are either simple clients or arbitrary complex generic servers. We use the fact that the message buffer of these servers is read in a *fifo* order. The language $\mu$CRL has synchronous communication and buffers are not provided in the language. The compiler constructs a process acting as a *fifo* buffer for each server in the system. Clients communicate with the buffer to send a message and the server communicates with the buffer to read the first message in the queue. In case two clients have the possibility to concurrently send a message to the same buffer, this will result in the state space as two different paths, one where the first client sends followed by a send of the second client and the other path where the second client sends followed by the first client. The server then executes the appropriate *handle_call*, *handle_cast* or *handle_info* and replies an answer to the client. This reply does not end up in the buffer; the generic server semantics prescribes that the client is blocked waiting for an answer from the server. This is implemented in Erlang by scanning the message buffer for a message with the right unique tag. We by-pass this scan of the buffer by translating a reply to a synchronous communication between client and server process.

For every server in the system we need to create another process besides the buffer process: the so called *call-stack*. The specification language $\mu$CRL strictly separates communication and computation. A computation cannot have communication as a side-effect. In general this separation of concerns results in clear, understandable specifications and we promote to use a similar strategy for Erlang programs as well. However, sometimes it is really clearer or more efficient to have an Erlang function that has communication as a side-effect. For example, the storage group process sends a message to all storage devices it has stored in a certain list. Assume the list contains tuples with name of the device and number of request one has send to it. As a result of sending a new request, one might want to obtain an updated list. It is more efficient (and probably clearer) to traverse the list only once and return the updated list as a result and sending the messages as side-effects. In order to translate such a construction to $\mu$CRL, the function sending values is translated into a function with accumulator that saves the updated list in the accumulator. Instead of returning this accumulator (which is impossible in $\mu$CRL) in the base case of the recursion, the function pushes the value on a stack by communicating it with the stack process. The calling process pops the value from the stack by a communication with the stack process and uses the value to continue computation.

The compiler analyzes the Erlang modules and divides the functions in it in two categories: those that are side-effect free and those that contain side-effects. A function is said to

have a side-effect whenever it (indirectly) sends or receives a message. Both parts are translated differently, because the side-effect free part is translated in rewriting rules in $\mu$CRL, whereas the part with side-effects is translated into processes.

Down to this point, Erlang has many features that are not supported by a process algebra, which is a specification language, not a programming language. Higher-order functions, records, list-comprehensions and all these things that make the Erlang code so readable have to be compiled to simple first-order functions[1], tuple-like data structures and so on. The compiler performs these transformations as Erlang to Erlang transformations. On the one hand this supports debugging of the compiler, on the other hand, it allows a flexible change of back-end.

Modules are not supported by $\mu$CRL either. Therefore, all imported functions have to be in-lined and name conflicts have to be resolved. Somehow it is strange that a relative modern specification language has so poor features for specifying large software systems on a high level. The language LOTOS is better in this respect, but also in that language support for higher-order functions is lacking.

After all these Erlang to Erlang transformations a single Erlang file is produced. This file is translated into $\mu$CRL by rather straight-forward syntactic manipulations. The only non-trivial part is that $\mu$CRL is a strongly typed language and Erlang is not. We generate one type in $\mu$CRL to represent all Erlang terms. All side-effect free computations are rewrite rules on terms of this type.

The obtained $\mu$CRL specification can be used to generate the state space of the Erlang system. However, in our scheduler we are not ready yet. As our main goal is to automatically obtain global properties about the performance and behavior of the system, we only have to analyze the control core of the video-on-demand server. But for 'activating' the system, a special *interface* that represents the possible users asking for media objects has to be added to the model. One direct solution would be to add to the Erlang source code an abstraction of the user process and to include this in the supervision structure of the system, thus introducing the number of users as a parameter in the same level as the number and configuration of the different devices. The problem of this approach is that it would be hard to explore all the different combinations of users, because the fact of including a lot of new processes would make the state space to grow exponentially. In the state space with many users one would get different paths for different orders in which the users ask for media objects. We are, however, not at all interested in the difference between user one asking for 'Star Wars' followed by user two asking for 'Star Wars' and the sequence in which they ask it the other way around. The only thing that is important for us is that two users asked for 'Star Wars' after each other. Another disadvantage with the solution of many user processes is that it is hard to tell how many of them will be needed to determine the capacity of the system.

The solution we choose lays in a different approach for modeling the users: in the Erlang source code, only one client process is used to model the *user pattern*. That process asks for a media object with a given quality, waits for the answer (the set of options provided by the server), and plays the object or asks again (if the request fails), repeating this loop all the time. Then, after creating the $\mu$CRL file, we use our tool for automatically adding non-deterministic behavior to the user process. Instead of asking for a concrete movie with a given quality, now the user is going to ask non-deterministically for one out of all possible media objects with one out of all the possible qualities. Thus, we use one non-deterministic user process for modeling an infinite set of users that constantly request media objects. Since the user process never releases a media object, this process will only receive 'fail' as a reply after that the system is overloaded. As such, there is a natural bound on the number of users in the system and the creation of an infinite state space is avoided.

[1]There is no general solution to translate higher-order functions to first-order functions, but the compiler supports *map*, *fold*, etc.

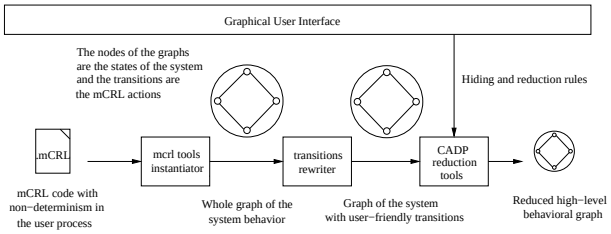


**Figure 5: From $\mu$CRL to state space of the system**

The tool set for $\mu$CRL comes with several other tools that can be used to modify the $\mu$CRL specification before generating the state space [9]. The aim of these modifications is to end up with a specification that gives less states and transitions (events) in the generated state space. This can theoretically be achieved because some events, like the communication with the *call-stack* or the *buffer* may be seen as internal actions and can be hidden in the state space. Tools like a confluence analyzer can be used to modify the source code in such a way that only one of the many confluent paths to a result is chosen in the generation of the state space, provided that the obtained state space is observational bisimular with the original one. We experimented with these tools as well, giving reductions of ten to twenty percent in the generated state spaces.

## 3.2 Generating a State Space from $\mu$CRL

After generating the $\mu$CRL code for the concrete configuration of our system, the second step of the proposed methodology is to create the state space for that configuration. This step is based on standard tools for $\mu$CRL [13], and tools for hiding and renaming labels as well as the reduction tools in the Cæsar/Aldébaran tool set [7]. Fig. 5 depicts which tools are used to create a reduced state space from the process algebra specification.

In the Erlang scheduler software there are many processes that communicate with each other. The communication is rather straight-forward, though. The user sends a request, this is passed from one process to the other and in the end a growing list of possibilities is passed back.

For the typical properties we are interested in, we are only concerned about the messages that the user sends to the system and the messages that are returned to the user. In that way, we can judge whether a user can be served and what the possible choice for the user are. The messages

between streaming level and storage level, between storage group and devices, are irrelevant for our purpose. However, the translation from Erlang to $\mu$CRL is such that when we use the *instantiator* tool on the specification directly, we obtain a state space in which all these irrelevant messages are visible as well. Therefore, the state space contains at least $4 * (2+\#\text{devices})$ times more events than we are interested in, whereas in practice this redundancy turns out to be even larger.

At the moment we generate the full state space. We then rename the labels of the events we are interested in and we hide the other events. By using reduction tools we can perform observational bisimulation reduction on the state space, obtaining in this way a much smaller state space in which only the relevant events are shown.

Renaming the labels is useful to create a better readable visualization of the graph. For small configurations such visualizations can be illustrative for the designers to look at. Moreover, the properties that we use later refer to the renamed labels. If we would not rename the labels, the properties would be harder to formulate. Hiding the irrelevant events has the two advantages that the properties can be formulated configuration and system implementation independent (they need not reflect internal behavior) and that model checking them is faster.

For small examples, the transformation from the Erlang code and the concrete configuration to the abstract behavioral state space is performed in a matter of seconds. As we complicate the system configuration, the state space grows and the computation time is much larger. As an example, the state space of a two level configuration, without cache nor restrictions in the upper levels, with four devices in the massive storage level and all the possible combination of movies distributed over the devices in two different qualities, contains up to a few million states. Its generation takes some hours and results finally in a reduced state space of about one thousand states.

Fig. 6 shows a very small illustrative example of a reduced state space obtained from a simple linear configuration. The system was configured with two linear levels (streaming level -with the local scheduler-, and massive storage level -with the scheduler and the storage group). The storage group was grouping two devices: a tape with 2 units of bandwidth, only able to handle one connection at the same time; and a CD with 3 units of bandwidth able to handle 2 simultaneous connections. No extra restrictions were placed in the hierarchy, other than trivial cost functions able to select the right providers for the users. In this example, we used the abstract approach for the media objects, placing in the devices all the possible combinations of MOs. That is, to have *m1* as the abstract kind of MOs that are in both devices, and *m2* and *m3* as the MOs that are only in one of them. For the quality of the MOs we chose to have only two possible qualities with 1 and 2 units of bandwidth respectively.

For the example configuration, from the original whole state space of the system, with a total of 2547 states and 2747 transitions, the reduction results in the 8 states and 48 transitions shown in Fig. 6. In the graph, the performance pattern for this kind of systems can be seen; from the initial state 0, the transitions explain which are the actions the users are able to perform (e.g. *play(tape,m3,1)* means to serve an MO from the group *m3* on the device called *tape* using 1 unit of bandwidth). After more users requesting

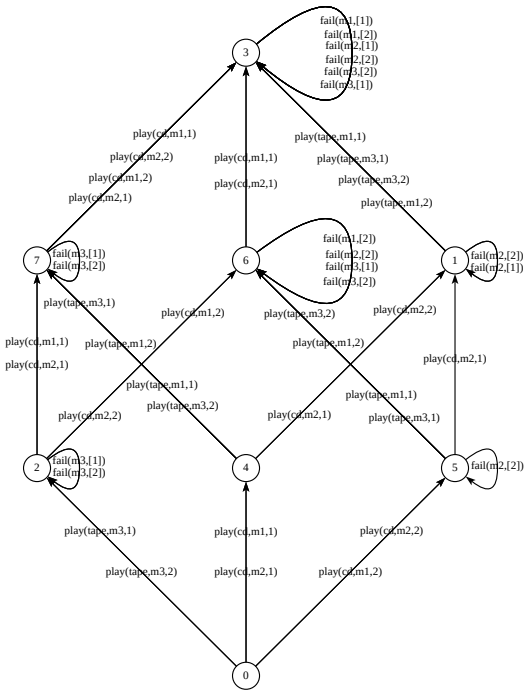


**Figure 6: Abstract graph for a simple configuration**

movies, the system is becoming more busy. Some resources are not available to handle some a user request (e.g. in state 5 the system is not able to provide the user an MO from group *m2* at a quality of 2 units bandwidth, because that media is only on the CD device, and the device only has 1 unit of bandwidth available at that moment). The last node of the graph is always representing the maximum load of the system, where all the resources are being used (the bottlenecks of the architecture are using their maximum capacity), and every user request results in a *fail* as reply.

The main performance limitations of the approach is the time it takes the state space generation tool of the *$\mu$CRL tool set*. As we said, our tool is able to handle reasonable sized real configurations, but when the goal is to analyze a really complex tree-like hierarchical architecture, with a big number of devices and restrictions, and thousands of media objects, the computation time and the use of resources (specially CPU but also memory handling the states) need to be reduced as much as possible. During the evaluation of the tool we also have found some performance problems when using big numbers in the model, due to the fact that the traditional $\mu$CRL representation of the natural numbers, based in the successor constructor, is not efficient when evaluating the model symbolically.

As improvements of this performance limitations, we are already exploring the use of a kind of theorem proving tools that are part of the $\mu$CRL tool set. As mentioned in Sect. 3.1, these tools are able to automatically prove certain properties of the $\mu$CRL specification, like confluence of a certain pair of actions. By exploring these properties the $\mu$CRL specification can be modified to a specification that results in a smaller, but observationally bisimular equivalent state space. The first experiments have shown promising results

in reducing the size of the whole state space of the system (both in number of states and transitions). A new more efficient representation of the natural numbers is also being explored as a solution to the performance limitation in numeric computations of the model.

## 3.3 Verifying Global Properties

Once we have generated the reduced state space, representing the behavior of the system from a black box point of view, the last step of the proposed methodology is to extract the performance properties from this graph by using model checking techniques.

An important goal in this step is to provide the users of our tool (both the designers trying to improve the performance and the cable operator evaluating the system) with an understandable set of properties presented in natural language via a graphical user interface. The user is given the possibility to ask questions like "How many movies can the system serve simultaneously?" instead of "What is the longest path in the state space to the point where only *fail* transitions are possible?". Another desirable feature when designing this part of the methodology is to provide the user with some kind of feedback information giving design related suggestions.

One of the main subsystems of the GUI developed in our prototype is devoted to this high level interface. The user can automatically check some properties, and can formulate others in a very easy way, giving, for example, a description of a concrete branching scenario in order to know if that can happen in a real execution of the system. Internally, these properties are converted to an alternation free $\mu$-calculus expression, that is going to be model checked using the CADP tools; therefore, the users do not need to deal directly with the logic.

The properties we offer the user to be analyzed are divided in three main groups, depending on the way the model checking techniques are used in order to obtain the information from the graph:

1. Counter-example based:

   With this method for extracting system information we propose a new way of using model checking tools. Instead of checking the property in order to know if it holds for the graph we are analyzing, we try to find the negation of the formula that represents the information we want to obtain. Therefore, a counter-example to the formula corresponds to an example that the global property holds, thus the counter-example gives us the knowledge about the system.

   One of the main properties a user would like to know about this kind of systems is its global capacity, but the abstract concept of *capacity* is really measured by a complex set of properties. One of them would be 'the worst case scenario in which the system reaches its maximum load', that is, translated to the graph language, 'the shortest path to the special node where the system only can *fail*'. A way of obtaining that information, using the counter-example based approach, is to use a property that we know is only false in the special node of the system where only fails can take place, viz. `[true*]<not 'fail.*'>true` (i.e. starting at any state of the system, it is always possible to have a transition that is not a *fail*). The counter-example for that property is going to be (with the tools we are using) one of the shortest paths to that node.

   Other properties for the systems, related with its performance and capacity, are obtained using the same approach. Instead of looking at the whole capacity of the system, we can focus on a concrete MO, and know the server performance for that concrete media object. By using the $\mu$-calculus property `[true*]<not 'fail(.*,m1,.*)'>true`, the tool can automatically obtain a counter-example with the shortest path to a fail for that media object. Some other similar expressions are used in order to check the worst case performance for the system where only plays are possible, e.g. the shortest path to any kind of fail (giving us a capacity idea about for how long the system is still able to always serve the user).

   Another interesting property is the maximum of media objects that can be served simultaneously in such a way that after serving these objects all possible requests can still be honored. Thus, the paths where all states only have successful successor states. In our little example in Fig. 6 this corresponds to the path from 0 to 4. Thus, in that example, the maximum number of simultaneous users after which a next requesting user always can be served is therefore 'one'. In general one can have several distinct paths in which all users can still be served. With this counter-example based approach we automatically determine the shortest of them by using the $\mu$-calculus formula:
   `[true*](<'fail.*'>true<true>['fail.*']false`.
   The longest of them can be detected using the logarithmic search approach described below.

   Thus, with the proposed technique the user can obtain automatically, by using a high level user interface, scenarios that fulfill global properties of the system behavior. All these properties can be combined with the bandwidth usage in order to extract more detailed information.

2. Logarithmic search:

   Complementing the counter-example based approach, a different set of global properties of the system can be obtained by using model checking in combination with a fast search algorithm with logarithmic complexity. With the counter-example approach some interesting results cannot be extracted from the graph, because of the limitation of the CADP model checker tools, that are giving always a shortest path to a node where the property does not hold.

   One kind of properties the users of our system are interested in are reflected by questions like: "What is the maximum number of users that can watch 'Star Wars' at the same time?" or "How many simultaneous users can the system provide such that it still is always able to play 'Star Wars'?". Expressed in a property over the graph these questions refer to the longest paths to some special nodes or situations.

   As an illustrative example we have the previously mentioned maximum number of simultaneous users, such that the next requesting user can always be served. The longest path to a state where all requests are successful is found by repeatedly proving properties of the

form: `<true*.'play.*'.true*...>['fail.*']false`, where the length of the sequence of the second `true*` occurrence is varied doubled until the property is false and then we search the exact point of failure by taking the middle between previous success and previous failure recursively.

These properties, extracted also from the graph but using a different approach, complement the information obtained with the counter-examples for giving the user global properties of the system. These properties are again obtained automatically, with only high level user interaction through the GUI.

3. Scenario based:

   Finally, as the third kind, the tool also provides to the user a more open interface for expressing scenario-like properties in almost natural language, that later are transformed internally into $\mu$-calculus expressions. Examples of this kind of properties are the *existential* properties, where the user can describe a concrete scenario (with MOs and bandwidths) and ask the system if that can happen; and the *eventually existential* properties, where the user can describe a more complex scenario in the form 'after looking at a given sequence of MOs with a given quality, it is still possible to have the following scenario: ...?'.

   With these kind of properties, the abstraction of the MOs placed in the system can effectively be used. If the user asks the tool whether a given scenario is possible, the tool checks this scenario for all abstract movies, instead of just the one that the user asked for.

   For example if the users ask whether the system can provide the following sequence of movies: movie 1, movie 3, and movie 2. Assuming that movie 1 and 3 are both belonging to group $m1$ and that movie m2 belongs to group $m3$, then to answer *yes* or *no* to this question, the property `<'play(.*,m1,.*)'.'play(.*,m1,.*)','play(.*,m3,.*)'>true` is checked. However, if the answer to the question is *no*, then we can also automatically check the property where we smartly exchange $m1$ and $m3$ by other groups. If one of these properties is true for the state space, we obtain a result that can be presented as an advice to the user. In this way, the tool can return a more complete answer like: "the scenario cannot occur, but moving movie 2 from the tape to the second CD makes this scenario possible in the system".

Thus, by combining three (complementary) ways of using of model checking techniques a method is given for automatically verifying global properties of the system.

# 4. CONCLUSIONS

In this paper, the scheduler of the VoDka system, a video-on-demand server, acts as a case-study for our methodology to verify global properties of a system. The behavior of the system is hidden in a complex distributed scheduler, based on component restrictions (bandwidth, number of connections), local policies (cost functions, filters), cost related functions (state of the components and resources still available), and a flexible hierarchical architecture. By means of this scheduler, we present a methodology for extracting

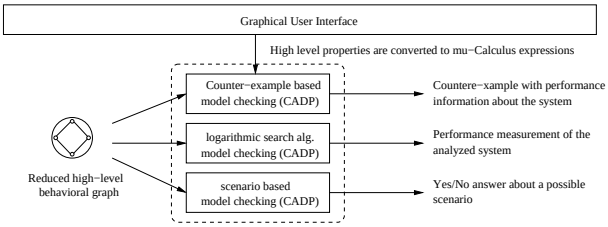


**Figure 7: From the behavioral graph to the global performance properties**

global properties of an Erlang system from the local restrictions hidden in its processes. The methodology is more generally applicable than only for schedulers. Many systems can be seen as a deterministic function over the input (in our case sequences of users demanding movies). However, these functions are composed of many small functions: the components. These components can have state and are therefore hard to statically analyze. However, with our approach, we simulate all possible runs of the system, as such obtaining the function between domain and range compacted as a graph.

The methodology is based on three main steps that are all performed in a completely automatic way. In the first step, we use our *Erlang to $\mu$CRL compiler* to translate the original Erlang source code modules into a $\mu$CRL specification. In the second step, we use the *$\mu$CRL tool set* for generating the whole state space, which then is reduced to a smaller one where only the information needed for extracting the performance properties is shown. Finally, we use the model checker of the *CADP tool set* in three complementary ways for extracting interesting properties for the user from the reduced state space. All these steps are performed by the user with a high level graphical user interface, developed with the goal of hiding the internal details of the approach. The tool is even able to analyze the system in order to provide the user some feedback information with suggestions about how to improve the system performance.

At the moment our approach is the only one that can automatically verify this kind of global properties of a system. Other code level model checkers for Erlang with this accuracy do not exist. For Java and C, however, there are similar approaches when it comes to model checking source code [6, 10, 11]. We are not aware of an attempt to use these source code model checkers for analyzing a flexible concurrent and distributed architecture in the way we explain in this paper. We apply ideas of simulation in a model checking framework.

There are many papers on the analysis of scheduling algorithms and simulation. We have not yet studied this material properly. However, we can state that most simulation tools use a simulation specification language that differs from the programming language the system is implemented in. We are not aware of an automatic translation from the source code to the simulation language.

The Erlang source code of the VoDka scheduler contains approximately 600 lines of code distributed over nine modules (plus the code of the generic behaviors, of course). This code implements the kernel part of the video server. We modified the code by abstracting away the parts that are not necessary for the performance analysis (e.g. low level

transmission protocols and processes). We also ignore the release of resources, since we want to look at the overloading of the system. This means that we are looking at the worst case performance scenario where users only request and do not release a media object.

With the approach described in this paper, we are able to handle configurations of the system that are as complex as the ones that are being used in the VoDka prototypes that have been deployed for the cable company (with the manual abstraction that there single cache level is seen as a storage level). This means that we are able to extract automatically performance information about the system from its source code and configuration parameters.

The proposed approach uses formal methods in a rather original way: using the $\mu$-calculus as a powerful declarative graph information extraction language; and using model checking tools as general, flexible and efficient graph search algorithms. We verified properties for several configurations. Most time is spent on the generation of the full state space. The compilation from Erlang to $\mu$CRL takes only a few seconds and similarly the reduction of the state space and verifying properties of it only takes a few seconds up to a minute. Generating the state space, though, may take a few hours for rather large configurations. For that reason, we are exploring the best way of using *$\mu$CRL tools* for converting the process algebra model in the system to a bisimular equivalent one that produces a smaller state space of the system. We can indicate already in the specification which action we want to hide and perform transformations on the $\mu$CRL level to obtain a specification that results in a reduced state space. The first results with these tools are very promising, being able to reduce more than ten percent of the size of the state space. Additionally, a new representation of natural numbers in the process algebra, in order to improve the performance of the symbolic computation of the model, is subject of study.

As future work, we plan to add a cache level in order to be able to handle the complete architecture of the system. The consequences of adding this level to the architecture, specially in the aspects related with the abstract analysis of media objects, is subject to study at the moment.

We also plan to inspect more parts of the whole state space of the system in order to give the user more complex feedback information. It is in particular interesting to be able to extract information about bottlenecks of the system, i.e. being able to give the user more complete answers (e.g., "Thirty users cannot request Star Wars simultaneously with the proposed configuration of the system, because the bandwidth communicating the CD device with the system is too narrow").

# 5. REFERENCES


[1] J. Armstrong, S. Virding, M. Williams, and C. Wikström. *Concurrent Programming in Erlang, 2nd edition*. Prentice Hall International, 1996.

[2] T. Arts and C. Benac Earle. Verifying Erlang code: a resource locker case-study. In *Int. Symposium on Formal Methods Europe*, volume 2391 of *LNCS*, pages 183–202. Springer-Verlag, July 2002.

[3] T. Arts and I. van Langevelde. Correct performance of transaction capabilities. In *Int. Conf. on Application of Concurrency to System Design*, pages 35–42. IEEE Computer Society Press, June 2001.

[4] M. Barreiro, V. M. Gulías, J. L. Freire, and J. J. Sánchez. An Erlang-based hierarchical distributed VoD. In *7th Int. Erlang/OTP User Conference (EUC2001)*. Ericsson Utvecklings AB, September 2001.

[5] M. Barreiro, V. M. Gulías, J. J. Sánchez, and S. Jorge. The tertiary level in a functional cluster-based hierarchical VoD system. In *Functional Programming and $\lambda$-Calculus Workshop*, volume 2178 of *LNCS*, pages 540–554. Springer-Verlag, February 2001.

[6] J. Corbett, M. Dwyer, and L. Hatcliff. Bandera: A source-level interface for model checking java programs. In *Teaching and Research Demos at ICSE'00*, June 2000.

[7] J. Fernández, H. Garavel, A. Kerbrat, and R. Mateesc. Caesar/Aldébaran development package: A protocol validation and verification toolbox. In *11th Int. Conf. on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, August 1996.

[8] J. Groote. The syntax and semantics of timed $\mu$CRL. Technical Report SEN-R9709, CWI, Amsterdam, The Netherlands, June 1997.

[9] J. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Technical Report SEN-R0117, CWI, Amsterdam, The Netherlands, 2001.

[10] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *Software Tools for Technology Transfer*, 2(4):366–381, March 2000.

[11] G. Holzmann. *The Design and Validation of Computer Protocols*. Edgewood Cliffs MA: Pretence Hall, 1991.

[12] ISO/IEC. Lotos – a formal description technique based on the temporal ordering of observational behaviour. In *International Standard 8807*, Information Processing Systems – Open Systems Interconnection. International Organization for Standardization, September 1988.

[13] SEN group. A language and tool set to study communicating processes with data. Technical report, CWI, `http://www.cwi.nl/~mcrl`, February 1999.

[14] A. G. Wouters. Manual for the $\mu$CRL tool set (version 2.8.2). Technical Report SEN-R0130, CWI, Amsterdam, The Netherlands, 2001.