Towards Type-based Optimizations in Distributed Applications using ABS and JAVA 8

VLAD SERBANESCU and CHETAN NAGARAJAGOWDA and KEYVAN AZADBAKHT and FRANK DE BOER and BEHROOZ NOBAKHT,

Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands

In this paper we present an API to support modeling applications with Actors based on the paradigm of the Abstract Behavioural Specification (ABS) language. With the introduction of JAVA 8, we expose this API through a JAVA library to allow for a high-level actor-based methodology for programming distributed systems which supports the programming to interfaces discipline. We validate this solution through a case study where we obtain significant performance improvements as well as illustrating the ease with which simple high and low-level optimizations can be obtained by examining topologies and communication within an application. Using this API we show it is much easier to observe drawbacks of shared data-structures and communications methods in the design phase of a distributed application and apply the necessary corrections in order to obtain better results.

Categories and Subject Descriptors: []: —

General Terms:
Additional Key Words and Phrases: cloud computing, programming models, distributed applications, formal methods, optimization

ACM Reference Format:

format *ACM Trans. Appl. Percept.* 2, 3, Article 1 (May 2010), 8 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The Java language is one of the mainstream object oriented programming languages that supports a programming to interfaces discipline. It has evolved into a platform to design and implement standards in several domains of both research and industry, along with supporting its community with new language features and standards. With application reaching exascale dimensions in terms of data volumes and requiring a lot of computing power, focus has increased in researching numerous libraries and frameworks with an attempt to provide distribution and concurrency at the level of Java language. However, it is widely recognized that the thread-based model of concurrency in Java that is a well-known approach is not appropriate for realizing distributed systems because of its inherent synchronous communication model. A powerful concept on the other hand is the event-driven actor model

DOI : http://dx.doi.org/10.1145/0000000.0000000

of concurrency introduced in [Hewitt 1971] which allows many applications to extend these actors to suit their behaviour. Examples of these domains include designing embedded systems [Geoffray et al. 2008], wireless sensor networks [Cheong et al. 2005], distributed web-services[Serbanescu et al. 2012], multi-core programming [Pop et al. 2008] [Karmani et al. 2009] and delivering cloud services through SaaS or PaaS [Pierre and Stratan 2012] [Nicolae et al. 2011]. Furthermore, it provides the basis for increasingly popular languages in parallel and distributed computing like Scala [Haller and Odersky 2009]. However, such a language uses an explicit mechanism at application level to support message passing and handling, which diminishes the general object-oriented approach of method look-ups that forms the basis of programming to interfaces.

We introduce Java 8 API [Nobakht] to program distributed systems and to formalize actor-based programming which implies asynchronous message passing together with the evergrowing object-oriented software engineering approach. Using asynchronous message passing and a corresponding actor programming methodology which abstracts invocation from execution (e.g. thread-based deployment), we want to fully support and emphasize the programming to interfaces discipline. The main research question of this paper is to demonstrate that using this API, several type-based optimizations can be achieved at the design phase as well as detecting possible bottlenecks in distributed applications using the simple example of The Sieve of Eratosthenes. This is the first step in researching how to use type-systems to automate optimizations in parallel and distributed applications.

2. THE ABS LANGUAGE

Our starting point for the actor programming model assumed in this paper is the Abstract Behavioral Specification language (ABS) introduced in [Johnsen et al. 2012]. ABS offers programmers several features such as asynchronous method calls, futures to control these calls, interfaces for encapsulation and cooperative scheduling of method invocations inside concurrent (active) objects. Specifically any object created in ABS represents an actor with encapsulated data. Similar to JAVA, their behaviour and state is defined by implementing interfaces with their corresponding methods. Thus they interact by making asynchronous calls to these methods which generate messages that are pushed into a queue specific to each actor. An actor progresses by taking a message out of its queue and processing it by executing its corresponding method. This feature combination results in a concurrent object-oriented model which is inherently compositional. The simplicity of ABS results from the fact that each actor is viewed as a separate processor making it very suitable for modeling distributed applications similar to MPI [Gropp et al. 1999], with the added benefit of specifying a distinct behaviour for each actor without the connectivity issue. Finally asynchronous method calls use futures as dynamically generated references to return values.

3. THE ABS-API LIBRARY

In this section we focus on the features in Java 8 that allow us to have an efficient and easy to use implementation of the actor model in ABS. First, methods in an interface are declared as Defender Methods using the **default** keyword. This allows actors to have a default behviour and optionally override this behaviour to suit a specific function. For instance, in Java 8 java.util.Comparator provides a default method reversed() that creates a reversed-order comparator of the original one. Such default method eliminates the need for any implementing class to provide such behavior by inheritance. Second, the introduction of Java Functional Interfaces and lambda expressions is a fundamental change in Java 8. All interfaces that contain only one abstract method are now functional interfaces that at runtime can be turned into lambda expressions. This means that the same lambda expression can be statically cast to a different matching functional interface based on the context. This is a fundamental new feature in Java 8 that facilitates application of functional programming paradigm in an object-

oriented language. This API makes use of these new features available in JAVA 8 because many of the interfaces found in the Java libraries are now marked as functional interfaces, most important of which in this context are java.lang.Runnable and java.util.concurrent.Callable. This means that a lambda expression can replace an instance of Runnable or Callable at runtime by JVM. Therefore a lambda expression equivalent of a Runnable or a Callable can be treated as a queued message of an actor and executed. Finally, Java Dynamic Invocation and execution with method handles enables JVM to support efficient and flexible execution of method invocations in the absence of static type information. This feature introduces a new API, available through java.lang.invoke.MethodHandles that allows translation of a lambda expression in Java 8 at runtime to be executed by JVM. Furthermore, this feature has been validated performance-wise over anonymous inner classes and the Java Reflection API. Thus, lambda expressions are compiled and translated into method handle invocations rather reflective code or anonymous inner classes.

The ABS-API library has a fundamental interface namely the Actor Interface. Using an interface for an actor allows an object to preserve its own interfaces, and also it allows for multiple interfaces to be implemented and composed. A Java API for the implementation of ABS models should have the following main features. First, one actor should be able to asynchronously send an arbitrary message in terms of a method invocation to a receiver actor. Second, sending a message can optionally generate the equivalent of an ABS future that the sending actor can use to refer to the return value. Finally, an object during the processing of a message should have a context reference to the sender of a message in order to reply to the message via another message. All these characteristics must co-exist without requiring any modification of the intended interface, for an object to act like an actor. The Actor interface provides a set of default methods, namely the run and send methods, which the implementing classes, as well as a queue that supports concurrent features of Java API 5. On one hand, the default run method takes a message from the queue, checks its type and executes the message correspondingly. On the other hand, the default (overloaded) send method stores the sent message in the corresponding queue. As mentioned before, in ABS we use futures to control synchronization. In the ABS-API we model messages that are expected to return a result as instances of Callable and a future is created by the send method which is returned to the caller, while those messages that need to run in parallel without a future reference to the outcome are modeled as instances of Runnable.

4. CASE STUDY

Using our solution, we present in this section a parallelized implementation of the Sieve of Eratosthenes [Bokhari 1987]. We aim to illustrate the benefit of using the Java language to program in an actor-based model while at the same time showing the performance improvement compared to other actor models, the benefit of observing certain behaviours in the programming phase, as well as showing that the actor-based model still performs well when compared to implementations that apply low-level optimizations. Generating prime numbers is a key factor in authentication algorithms. With distributed applications running on several cloud environments, the need to authenticate securely and transparently without a sizable overhead is constantly increasing. At the same time our case study is perfect for modeling partitions as actors as well as making it easy to simulate an application that can work on a multi-core platform using a shared memory or a distributed platform where communication between actors is key. The Sieve of Eratosthenes also allows us to illustrate several optimizations that result from the actor based model, as well as how certain well known optimizations are easy to apply in this model without significantly increasing the code size and therefore the design phase of a distributed application.

To model the algorithm using actors we use the well-known partitioning parallel algorithm and represent each partition as an actor. In this algorithm, the numbers are partitioned into smaller sequences

of numbers with the same size. Based on this algorithm, the size of each partition must be equal or greater than(except probably for the last partition) $\lfloor\sqrt{n}\rfloor$, and the number of partitions must be equal or less than $\lceil n\lfloor\sqrt{n}\rfloor\rceil$, where n is the target number. Following the above-mentioned constraints, the first partition contains all the prime number required to sieve, therefore the first actor in the model will be responsible for sending asynchronous messages to the others that will invoke the sieving process. With asynchronous messages written as regular method calls in Java, there is a significant improvement in the ease of programming compared to a similar solution that uses specialized directives like in MPI.

We decided to implement a data structure optimization, and therefore use a BitSet data structure and also half the amount of processing work by eliminating even numbers. These two optimizations clearly improve results and therefore needed to be applied before testing our model to other implementations which have at least these optimizations. We tested our solution on the SurfSara[Lysaght et al. 2013] cluster using a 16 CPU machine with 128GB of memory. A small example of a sieve invocation and using a future to synchronize on the result for checking the correctness of the prime numbers found at the end of the program is given below.

```
for (Actor s : actors) {
        //sieving process invocation for a new prime number
        Future<Object> r = this.send(()->{s.sieve(prime)});
        futures.add(r);
        }
```

Our main result in this paper is that with just the two standard optimizations, we obtained instant results for candidates up to 10^8 and 2.6 seconds when testing with 10^9 candidates. We decided to compare our results to the fastest sieving algorithm that further has cache-friendly memory management, wheel factorization and segmented sieve [Sieve]. Our model is only 10 times slower with the record program finishing for a target of 10^9 in 0.26 seconds. What we want to emphasize however is that the source code size for this record implementation is 505K compared to 30K, the size of the Actor-based model. This significantly improves the ease of programming even in a simple distributed application. Further comparisons with other Actor-based models will be discussed in the following section.

4.1 Type-based optimization

As discussed before, we aim to use this API to observe certain drawbacks or bottlenecks from the programming phase of the application. In this simple example it is easy to observe that the number of asynchronous messages sent between actors is very high. With the API exposed in the Java language we can easily use a shared data structure to eliminate the messages sent corresponding to each prime number used to sieve the partitions. While this is something very trivial, what we actually aim is to extract from this ABS-API is the possibility to detect and automate such optimizations depending on the application that is modeled. We want to be able to analyze several applications which can be CPU-intensive, IO-intensive or with multiple memory accesses and be able to detect performance penalties just like the one above.

5. EXPERIMENTAL METHODOLOGIES AND RESULTS

The development of multicore CPUs rapidly provides a bigger need for parallel and concurrent programming. Currently there are one more open source frameworks such as Akka, Erlang, Scala, Finagle, Storm, Hadoop, Ruby, Go Language, Hive, Pig available for distributed parallel and concurrent programming. Further Akka, Finagle, Storm and MapReduce are different elegant solutions for dis-

tributed computing and are based on functional programming languages. Pig programs are more complex, and can be compiled into an execution plan consisting of several stages of MapReduce jobs, some of which can run concurrently. Further Pig and Hive are script based data flow languages and thus more volatile and harder to debug during programming and provides a higher level of abstraction for MapReduce programming that is similar to SQL, but it is procedural code, not declarative. It can be extended with User Defined Functions (UDFs) written in Java, Python, JavaScript, Ruby, or Groovy and includes tools for data execution but was not ideal for implementing the Sieve of Eratosthenes case demonstrated in the proposed paper.

Also there are various actor oriented libraries and languages in the existing techniques for implementing some variant of actor semantics and are based on object oriented programming languages. The actor oriented languages includes but are not limited to Erlang, SALSA, E language and AXUM that are based on message passing. Further one of the important programming models based on message passing is the Actor model. The Actor model is an inherently concurrent model based on asynchoronous message passing . Moreover the Actor based model includes many important features such as encapsulation, fair scheduling , location transparency, locality of references which makes the actor model a suitable programming model for distributed parallel and concurrent programming. ABS is a concurrent , object-oriented modeling language that features functional data types. ABS model uses asynchronous method calls, interfaces for encapsulation, and cooperative scheduling of method activations inside concurrent objects. In specific the ABS language is a class-based object-oriented language that features algebraic data types and side effect-free functions. Also Actors implement a shared-nothing model for concurrency. A model represents a fragment of the state of sieve, specifically some subset of the primes discovered currently in the existing techniques. Further the existing open source frameworks are compared with the ABS model proposed in the paper for performance by implementing the Sieve of Eratosthenes case using the ABS API.

Currently there is a plurality of concurrent programming languages that use the Actor-based model approach for computing the primes using Sieve of Eratosthenes Algorithm. The results obtained from the existing concurrent programming languages such as Scala, Erlang and Go Programming Language are compared with the Actor based model approach implemented for Sieve of Eratosthenes Algorithm in the proposed paper. As discussed in the previous section, the Actor-based model approach proposed in the paper generates primes at 2.6 seconds until 10^9 on 16 CPU machines using Sieve of Eratosthenes Algorithm.

In the existing implementation for Sieve of Eratosthenes Algorithm in Ruby using JRuby and Akka, both the controller and model actors are defined as distinct classes. The message sent between the actors is a list with a leading symbol and a payload contained in the remainder of the list. The model only considers a value prime if it does not equal or divide evenly into any previously observed primes. The Sieve of Eratosthenes algorithm implemented in Ruby using JRuby and Akka computes primes until 10^4 in 77.114 seconds. This method was not effective and the performance was really slow once we got past an upper bound of about 10,000 numbers.

This follows a similar implementation [Tasharofi 2014] for the Sieve of Eratosthenes Algorithm in Erlang that uses tuples rather than lists for sending messages with actors. Erlang was one of the first prominent actors languages when it came out of Ericsson in 1993. It is a functional language, which extends to its native actors support. The Sieve of Eratosthenes algorithm implemented using Erlang calculates primes until 10^6 in 3.6 seconds. This method was effective for calculating primes until 10^6, but the performance was really slow for higher numbers between the range of 10^7 to 10^9. Further there are other actor-based languages, like Scala which closely follows the object-oriented model of programming though it has many functional programming features included to support message passing and

handling but this method diminishes the general object-oriented approach of method look-ups that forms the basis of programming to interfaces.

Further in the existing technique, the Sieve of Eratosthenes Algorithm is implemented using the Go programming language. Go programming language [Balbaert 2012] is a compiled language that combines some of the syntax of C with some more dynamic aspects to form a next generation systems programming language. One interesting feature of the Go language is the built-in multithreading feature which is based on channels and goprocesses. For the Sieve of Eratosthenes implementation using GoLangauge, each time a candidate makes it through the sieve and is returned as a new prime number. Further a new goroutine is created to check future candidates and reject them if they divide evenly by the new prime. The implementation is not useful as a standalone application since it includes no termination condition and also there are other disadvantages in the model as each goroutine knows only about the prime it contains and the channel where candidates should be sent if they pass. Once the goroutine is created its state does not change and also new state is added by creating a new goroutine for a newly-discovered prime and the state is never deleted. Moreover once a prime is discovered, removing it from consideration is non-sensical due to all states being completely distributed and no entity in the system knows about all discovered primes. The Sieve of Eratosthenes algorithm implemented using Go programming language calculates primes until 10^7 in 1m33.62s. This method was effective for calculating primes until 10^7 and could be further optimized using the Wheel Factorization optimization technique which in turn provided better time performance and calculated primes in 12 seconds for primes until 10^7.

Further we have compared the approach of the actor based model proposed in the paper on applications [Tipei] displaying different parallel patterns. The applications are namely Concurrent Sorted Linked-List (CSLL) and Prime Sieve (PSieve) that use a pipeline pattern to expose some parallelism. In the existing applications, the implementation maintains a list of helper actors with each actor responsible for handling request for a given value range for individual element operations. Also Collective operations, such as length or sum, are implemented using a pipeline starting from the head of list of the helper actors and only the tail actor returning a response to the requester. There are multiple request actors requesting various operations on the linked-list and non-conflicting requests are processed in parallel. The PSieve application represents a dynamic pipeline in which a fixed number of local primes are buffered in each stage. Every time the buffer overflows, a new stage is created and linked to the pipeline, thus growing the pipeline dynamically. This approach obviously affects the time performance and is slower compared to the Actor based model approach developed in the proposed paper.

Moreover by comparing the plurality of concurrent programming languages that use the Actor-based model approach for computing the primes using Sieve of Eratosthenes Algorithm, we have determined the Actor-based model approach implemented in the proposed paper provides better time performance and is efficient for computing primes at least until 10^9. The algorithm implemented using the Actor based model approach can be further optimized in the future work by having a single shared memory for storing the primes broadcasted from the Generator function. The algorithm proposed in the paper crosses off all multiples of a prime at once, we perform these crossings off in a lazier way: crossing off just-in-time. For this purpose, we will store a table in which, for each prime p that we have discovered so far, there is an iterator holding the next multiple of p to cross off. Thus, instead of crossing off all the multiples of 17 at once, we will store the first one (at 1717; i.e., 289) in our table of upcoming composite numbers. When we come to consider whether 289 is prime, we will check our composites table and discover that it is a known composite with 17 as a factor, and remove 289 from the table, and insert 306 (i.e., 289+17). In specific, we are storing iterators in a table by the current value of each iterator. Also a better data structure would be used for optimization as we only examine the table

looking for composites in increasing order. Thus we only need to check whether a candidate prime is the least element in the table (thereby finding it to be composite) or find that the least element in the table is greater than our candidate prime, revealing that our candidate actually is prime. In specific, a priority queue data structure would be used for optimization as it supports multiple items with the same priority (dequeuing in them arbitrary order) in the queue.

The actor based model can have further optimizations like wheel factorization and pre-sieving [Lauterburg et al. 2009]. For example, a third of our candidates are divisible by 3, and a fifth of them are divisible by 5. If, say, we avoid seeing multiples of 2, 3, 5, and 7, we can eliminate more than 77% of our work for large ”n” and even more for smaller ”n” in general. As we saw above in the case study, to produce numbers that are not multiples of 2, we simply begin at 3 and then keep adding 2. To avoid multiples of both 2 and 3, we can begin at 5 and alternately add 2 then 4. We can visualize this technique as a wheel of circumference 6 with holes at a distance of 2 and 4 rolling up the number line. In general, adding an additional prime p to the wheel multiplies the circumference of the wheel by p, and removes every p-th composite. This optimization of wheel factorization would be carried out for the future work using the Actor based model approach proposed in the paper for Sieve of Eratosthenes Algorithm.

6. RELATED WORK

Our Java ABS-API solution was constructed after looking at several works of research and development in the domain of actor modeling and implementation in different languages. We discuss a few languages at the level of modeling and implementation with more focus on Java and JVM-based efforts. Erlang [Armstrong et al. 1993] is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Its runtime system has built-in support for concurrency, distribution and fault tolerance. Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. The processes in Erlang communicate using message passing instead of shared variables, which removes the need for locks, but makes all synchronization explicit. Scala [Haller and Odersky 2009] is both a functional and object-oriented language that unifies thread-based and event-based programming model to fill the gap for concurrency programming. Like Java it provides the same features for handling concurrency, but it is not possible to to manage and schedule priorities on messages sent to other actors. We also compared our results to and Akka [Haller 2012] implementation of the actor model. This toolkit allows to build highly concurrent, distributed, and fault tolerant event-driven applications on the JVM based on actor model.

7. CONCLUSIONS

In this paper, we discussed an implementation of the actor-based ABS modeling language in Java 8 which uses the basic object-oriented mechanisms, principles of method look-up and programming to interfaces. We have used the API to model a simple distributed application that remains performant without applying specific optimization and fares much better than other actor-based models. We also showed the functionality of using Java to program distributed applications as well as making it possible to detect possible optimizations at the design phase.

The underlying modeling language has an executable semantics and supports a variety of formal analysis techniques, including deadlock and schedulability analysis. Further it supports a formal behavioral specification of interfaces to be used as contracts. As discussed in section 4.1 our research will focus on using type systems to automate optimization, extend our solution to identify resource usage of programs and communication topologies and apply a corresponding optimization table from which to eliminate drawbacks and bottlenecks during code generation. Our future work will also focus on

modeling more difficult distributed applications at testing important cloud features such as reliability, resource-provisioning, multitenancy and scalability. We also aim to automatically generate ABS models from Java code which follows the ABS design methodology. Model extraction allows industry level applications be abstracted into models and analyzed for different goals such as deadlock analysis and concurrency optimization.

REFERENCES

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1993. Concurrent programming in ERLANG. (1993).

Ivo Balbaert. 2012. *The Way To Go: A Thorough Introduction to the Go Programming Language*. IUniverse.

Shahid H Bokhari. 1987. Multiprocessing the sieve of Eratosthenes. *Computer* 20, 4 (1987), 50–58.

Elaine Cheong, Edward A Lee, and Yang Zhao. 2005. Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks. In *SenSys*, Vol. 5. 302–302.

Nicolas Geoffray, Gaël Thomas, Bertil Folliot, and Charles Clément. 2008. Towards a new isolation abstraction for OSGi. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*. ACM, 41–45.

William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

Philipp Haller. 2012. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. ACM, 1–6.

Philipp Haller and Martin Odersky. 2009. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2 (2009), 202–220.

Carl Hewitt. 1971. Procedural Embedding of knowledge in Planner.. In *IJCAI*. 167–184.

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*. Springer, 142–164.

Rajesh K Karmani, Amin Shali, and Gul Agha. 2009. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 11–20.

Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. 2009. A Framework for State-Space Exploration of Java-Based Actor Programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 468–479. DOI:http://dx.doi.org/10.1109/ASE.2009.88

Michael Lysaght, ICHEC Bjorn Lindi, Vit Vondrak, VSB John Donners, and SURFSARA Marc Tajchman. 2013. SEVENTH FRAMEWORK PROGRAMME. (2013).

Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. 2011. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.* 71 (February 2011), 169–184. Issue 2. DOI:http://dx.doi.org/10.1016/j.jpdc.2010.08.004

Behrooz Nobakht. https://envisage.ifi.uio.no:8080/jenkins/job/abs-api/javadoc/. (????).

Guillaume Pierre and Corina Stratan. 2012. ConPaaS: a Platform for Hosting Elastic Cloud Applications. *IEEE Internet Computing* 16, 5 (September-October 2012), 88–92. http://www.globule.org/publi/CPHECA_ic2012.html.

Florin Pop, Ciprian Dobre, and Valentin Cristea. 2008. Evaluation of Multi-Objective Decentralized Scheduling for Applications in Grid Environment. In *Proceedings of 2008 IEEE 4th International Conference on Intelligent Computer Communication and Processing, Cluj-Napoca, Romania, Published by IEEE Computer Society, ISBN: 978-1-4244-2673-7*. 231–238.

Vlad Nicolae Serbanescu, Florin Pop, Valentin Cristea, and O-M Achim. 2012. Web Services Allocation Guided by Reputation in Distributed SOA-Based Environments. In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*. IEEE, 127–134.

Fastest Sieve. http://primesieve.org. (????).

Samira Tasharofi. 2014. *Efficient testing of actor programs with non-deterministic behaviors*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Sever Tipei. COMPOSING WITH SIEVES: STRUCTURE AND INDETERMINACY IN-TIME. (????).