Petri Meta-Compiler — a Recursive Approach to System Design and Development

Piotr Chrząstowski-Wachtel[1], Michał Doleżek[2], Paweł Greipner[2], and Tomasz Wójcicki[2]

[1] Institute of Informatics, Warsaw University
pch@mimuw.edu.pl
[2] Devroom, Poland
michal,pawel,tomek@devroom.pl

Abstract. We propose a framework for the definition of Petri Net models in the Petri net approach. All the components: places, transitions, arcs are defined as Petri net objects. Assembled together they constitute a method to design business systems compositionally. Transitions are associated with a code, places contain data or are control places. Each entity (process) has a place, which serves as an interface to the external world (other processes).
The approach is implemented in such a way that it is possible to compile the codes of the designed systems within one framework (the same in which the model is defined). The approach enjoys the bootstrap compilation, so the system can compile itself. We have developed a method to make it in 3 phases. The first phase defines objects, the second one makes the net structure and the third one generates the code of the system itself.
The framework has been implemented and deployed in a few small and medium size internal business database projects. It provided great predictability of impact of changes and added flexibility to the system development. Characteristics of the resulting database structure with its immutable history of transactions seem compatible with blockchain concepts, thus we expect that such Petri Nets could become a convenient language for smart contracts.

1 Basics

Business processes, being quite complex and run by software, seldom run on reliable applications. To make application development manageable, the MVC approach separates three aspects of software — model, view and controller. When changes are made, the basic question is if the new environment is consistent with the old one. In MVC approach, the documentation serves as a tool for referencing and synchronizing the three aspects, but it is hard to maintain when changes happen often and sometimes it is confusing. Our approach synchronizes all the aspects of programming by using a common model for all of them. Such synchronization makes the approach more uniform. We gain a common language to talk about the control, the data and the view. The graphical charm of Petri nets made us eager to use a version of this model to combine all these aspects.

Petri nets have been considered an important model for business processes [5],[6]. They provide a scalable approach with well defined semantics. Also graphical charm makes them easy to explain for non-mathematicians. Our approach is based on the ideas of a hierarchical design and development of systems, raised among others in [4]. We were inspired by the paper [2], where the idea of a compiler that compiles itself was introduced.

We propose a language to define concurrent systems allowing hierarchical design in a Petri-net-like manner. The basic notions, from which we build the model are:

- Places
- Transitions
- Input arcs
- Output arcs
- Control places
- Processes

All of them satisfy normal Petri net assumptions. Places and transitions are disjoint and finite sets, arcs connect places and transitions. Places contain data, control places contain tokens. Processes are nets, which have been unfolded from one single place by means of structural refinement.

Arcs can also connect processes with transitions through special kind of places called handles. Transitions can change the state of the process they belong to, but also trigger changes of states of external processes through handles.

2 The Model

In what follows we define notions in one uniform framework. All the elements the system is built from (like places, transitions, arcs,...) will be defined in the language of Petri nets. The definition will be highly recursive, allowing in particular for bootstrap compilation.

A **system** is a set of processes $\mathcal{S} = \{S_1, \ldots, S_n\}$. A **process** is a Petri net

$$S_i = \langle P_i, T_i, \mathit{inarcs}_i, \mathit{outarcs}_i, C_i, h_i \rangle,$$

where P_i are places of S_i, T_i are transitions of S_i, $C_i \subset P_i$ is the set of control places, while $h_i \in P_i$ is the process handle — a place, which contains process id and which allows to communicate the interior of S_i with the external world allowing to use the data of S_i by transitions from external processes. We assume that places, which are neither control places nor handles contain the data of S_i. All elements of a process will be available through the dot notation by the id of the process. Let $H = \{h_1, \ldots, h_n\}$ be the set of handles, $P = \bigcup P_i$ be the set of all places, and $T = \bigcup T_i$ be the set of all transitions of the system. Let Idn be the set of all identifiers of objects including an element τ, which represents no id. There is also a requirement that the sets T_i are not empty and each of them contains at least one transition final_i. This transition is run in particular during the generation of objects — it puts then an identifier to the handle of this process. Places of any process can be handles of other processes. They will serve as links to external data.

Inarcs_S are input arcs for transitions of S. Formally they are partial functions $(P_S \cup H) \times T \to \mathit{Idn}$. If the value of $\mathit{inarcs}(p,t)$ is not τ (by which we mean that the arc exists) it is the id that will allow the transition t to access the data that flow through this arc during its firing. The only places, which can be inputs for a transition $t \in T_S$ are places from P_S and handles of external processes. So a transition cannot communicate with an external process by connecting to its internal place. If any requirements are defined for data to run a transition, then such requirements are stored as data within a process associated with the arc.

Similarly, $\mathit{outarcs}_S$ are output arcs for transitions of S. The only places which can be an output arc for a transition $t \in T_S$ are places of S and handles. So *outarc* is a function from $T_S \times (P_S \cup H) \to \mathit{Idn}$. If the value of an inarc or outarc is τ, we interpret it as no arc.

For the system $\mathcal{S}$ the underlying graph is a graph, whose nodes are elements of $\bigcup(T_S \cup P_S)$ and arcs connect pairs of nodes that are defined by *inarcs* or *outarcs*.

We assume here that in the underlying graph of each process S_i (so the underlying graph of the system with nodes restricted to places and transitions of S_i), for each transition t in T_i there exists a path from t to the handle h_i.

Every entity in a system (in particular all the objects created in any phase) must have a unique identifier from the set Idn .

A *marking* in a process S is a function from the set of places P_S to lists of strings. In our implementation numbers will also be denoted by strings. Handles can contain lists of identifiers only, while other places will use strings to denote names, numbers, program codes, etc. We will not distinguish here between control and data places: control places will have unique 0 or 1 (no multiple zeroes or ones), denoting a control token like in a simple Petri net.

The list of identifiers in a handle cannot be empty and once it is defined, we call such a marking an **object**. The first element on a list of identifiers in the handle must be unique in the whole system and is considered to be the object identifier. The remaining identifiers in the handle are references to objects of the same process. Usually some of the places of a process will be mandatory, so the existence of id in its handle will guarantee the presence of non-null data in such places. But in general an object can have missing data in some of its places.

A transition in S is *enabled* if there is a token on each control place connected to this transition by an inarc. The transition can fire at any moment provided it is enabled, but it must have a permission to do so. The permission can be granted either by an internal decision based on some logical condition or by an external action. One of the attributes of any transition will be its code. When a transition fires, the code of the transition is run and tokens are consumed from all input control places and, in the case of a proper termination of the code,

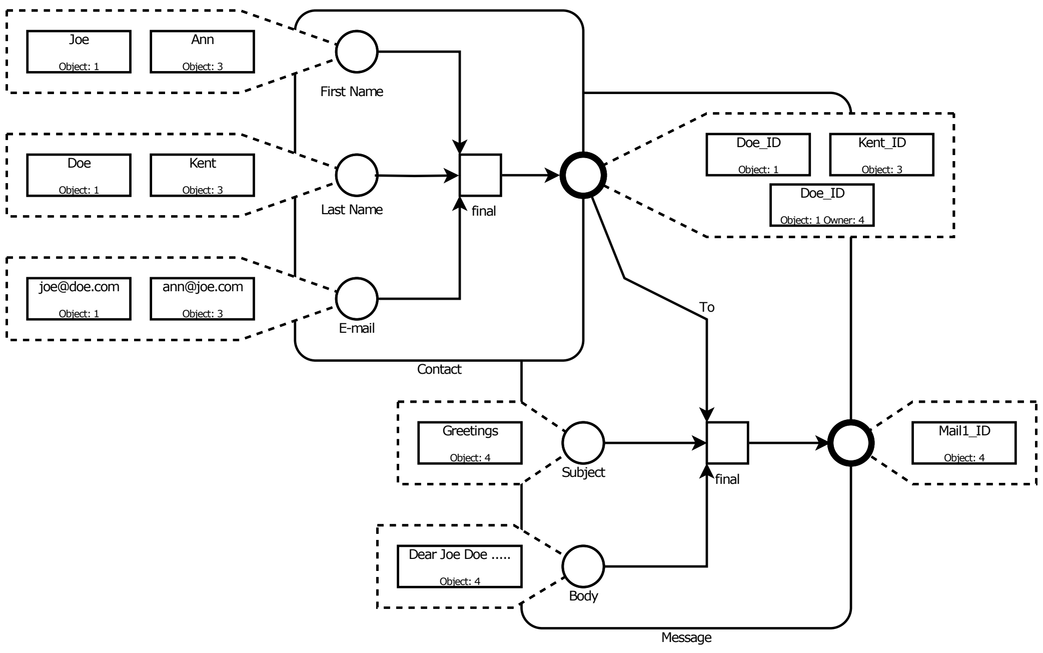


Fig. 1. Message storing example

the transition puts tokens on all output control places. If, by any reasons, a transition during its execution fails to be completed, the output control tokens will not be generated and the input control tokens will be restored.

If a transition is not the last transition of a process, and completes its execution successfully, it will update the content of the output places according to the code. Some of the output places can remain unchanged, if the code says so. A transition does not modify the content of its internal input places — it can only read their values. Similarly, a transition can only write on its output places and it can both read and write, if a place is both an input and an output for this transition.

Throughout the paper we adopt the following convention for the pictures. Every item on any figure presented with a solid line is an object. When the line is dashed or dotted it represents a notion; such items do not necessarily need to be realized within the model — it is a technical issue and can be implemented otherwise. We are defining our model within itself, so these items are compliant with it.

The frames with rounded corners are processes. The name of such process is at the bottom, outside the frame. Each of the processes has its handle on the right-hand side. Circles with bold borders are handles, normal borders mean data places, grey-filled circles are control places and rectangles are transitions. The names of items are located under the items and the names of arcs — over them.

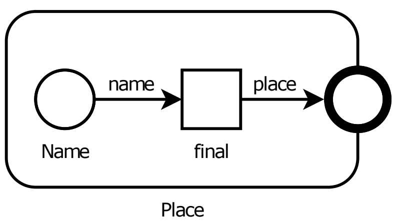
We show a simple example of a message storing application on Fig.1. The application consists of 2 processes which illustrate how data is stored and how processes and their objects can be linked.

Contact is a process which stores contact information. It consists of 3 places for contact's first name, last name and e-mail address, a transition and a handle for contact IDs of stored contacts. Adding a new contact is done by filling data in the 3 places and firing the transition. The transition has no code to run, but it can't be fired without data in the input places. It automatically generates a new object's ID upon successful firing and places it in the Contact handle. All data tokens of this object remain in their places and are automatically tagged with this object's ID. We can build a contact database by running this process repeatedly with different contact's data.

Message is a process that stores the actual message. Apart from similar places for data, transition and handle for message IDs, it has an arc that links Contact's handle to Message's transition and names it 'To'. That link means that a message's recipient is one of contacts stored in the Contact process and the information about which contact is linked to a message will be stored in that process' handle place, but attributed to Message's object.

On Fig.2-7 basic bricks, from which we construct nets, are described. Bold circles on the frames are handles, bold circles inside the frames are links to the external processes. Normal circles and boxes are internal places and transitions. If, for instance, a bold place **Transition** is inside the frame *Process*, such place will be identified with the handle of a process **Transition**. The identifiers, which reside in such a place denote the currently generated objects of the process Transition.

Below we describe the bricks from which every process is built. Mind that a common shape of all the Petri net components: places, transitions, arcs will be described in the same framework, and since all of them communicate with external world through the handle, and the handle is a place, you can find in the



```
Place       { "p" : 4, "o" : 10,
              "d2" : ["Name"] }

In_Arc      { "p" : 7, "o" : 11,
              "d2" : ["name"],  "l5" : [10] }

Out_Arc     { "p" : 8, "o" : 12,
              "d2" : ["place"],  "l6" : [14] }

Transition  { "p" : 9, "o" : 13,
              "d2" : ["final"],  "d3" : [" ... //code// ..."],
              "l7" : [11], "l8" : [12] }

Process     { "p" : 6, "o" : 14,
              "d2" : ["Place"], "l9" : [13] }
```

Fig. 2. Basic brick: place together with its JSON description

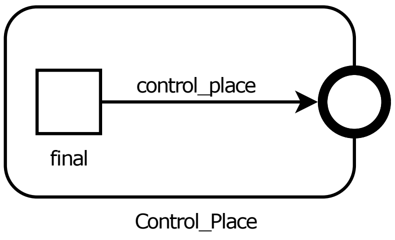


Fig. 3. Basic brick: control_place

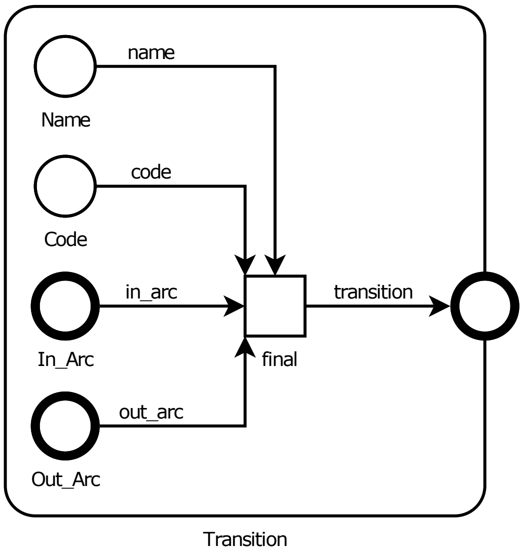


Fig. 4. Basic brick: transition

following diagrams places called "transition" or "arc", which can be considered bizarre, but is necessary to keep a uniform model for all the building bricks.

- The frame *Place* (Fig.2) describes a place (let's call it p) in the same framework. So a *Place* is a process consisting of one internal place with its name, one transition *final*, whose main aim is generating the identifier of this place in its handle. This transition will be fired during the every creation of a place object by the place process.
- *Control place* (Fig.3) need not have a name — it will not contain any data, only the control tokens, so no one will require any data access. In fact the identification of the control place will be done by the attribute *name* in the arc it is adjacent to.
- The frame *Transition*(Fig.4). We assume that transitions have attributes: *name, code, inarcs, outarcs.* We use inarcs and outarcs to match the transitions with neighbour places. The code will be run during transition firing.
- The frame *Inarc* (resp. *Outarc*)(Fig.5, resp. Fig.6) has *Name, Control_place, Place* and *Process* attributes. Since an *Inarc* connects exactly one place to a transition, we require here that in order to enable the transition *final* exactly

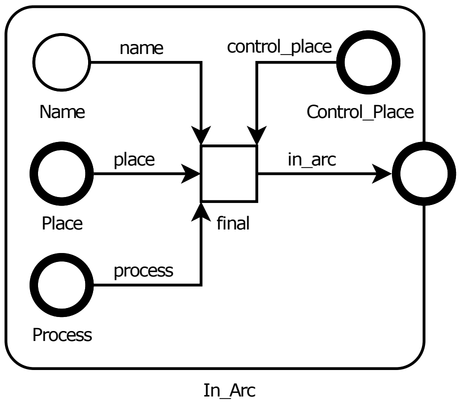


Fig. 5. Basic brick: inarc

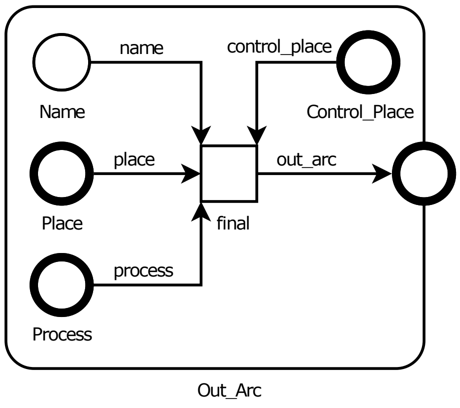


Fig. 6. Basic brick: outarc

one of these places must have data to make it being enabled together with the place name.

– The frame *Process* (Fig.7) has attributes: name, and a set of transitions, which reside in the place **Transition**.

We will describe the process of creating the framework for building systems in our approach. The overall scheme of the generation is depicted on Fig.16. It will consist of three phases. The first two are preparatory ones: they form atoms and molecules, from which we will next animate them to let them be alive and responsive. They are described in the next chapter and correspond to boxes appropriately labeled by 1 and 2 on Fig.16. The last phase — runtime — is described in another chapter and is depicted by the final loop in Fig.16. It allows for bootstrapping the whole system.

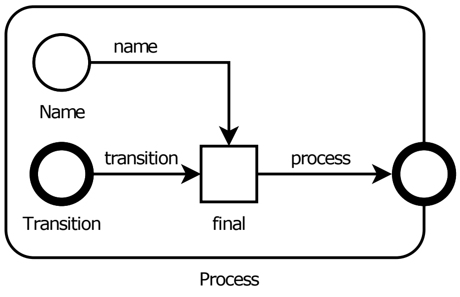


Fig. 7. Basic brick: process

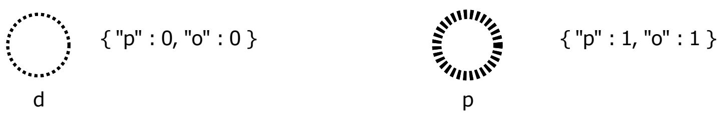


Fig. 8. Proto-atoms

3 Initial Loading and Generating

We start with defining the distinction between handles and normal data places. The proto-atoms (schemes) represented on Fig.8 do this. Places have an identifier 0, while handles have identifier 1, which is represented in the JSON notation. The JSON notation uses two additional fields to make it easier to read, we will use two mandatory fields: "o" to represent object's ID and "p" to represent the ID of handle where the object's ID is stored. Since object is a marking of a process, the id of this process can be treated as its type. So the type of object is explicitly indicated by the field "p" and its ID is in field "o" instead of being the first on the list in the handle indicated by "p".

Having distinction between places and handles, we define proto-types for our atoms: Name and Code (Fig. 9), having type 1 and having identifiers 2 and 3. Since they contain data they will be used in later phases as normal Petri net places and denoted on pictures by normal border. The next 6 items (Fig.10) are also of type 1 and have assigned consecutive integers — their identifiers will be used as handles. These are handles representing proto-types of the atoms, from which the net will be built.

The 8 circles from Fig.9 and 10 are data structures. To make their JSON notation easier to read, we will add a letter "d" to IDs of places, which contain lists of data tokens, and a letter "l" to IDs of handles which contain lists of references to objects, also referred to as links.

Once we define objects, which are parts of the net, we must initialize the computer memory to run the system. For this we need two phases.

3.1 Phase 1

We start with defining proto-class H, whose objects are proto-types of the atoms. The atoms themselves will be created in phase 2. The class H has 4 attributes,

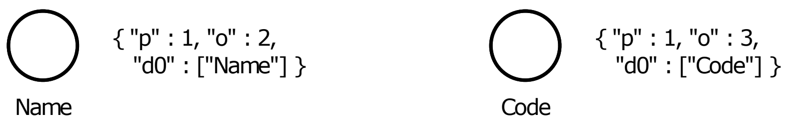


Fig. 9. Atoms

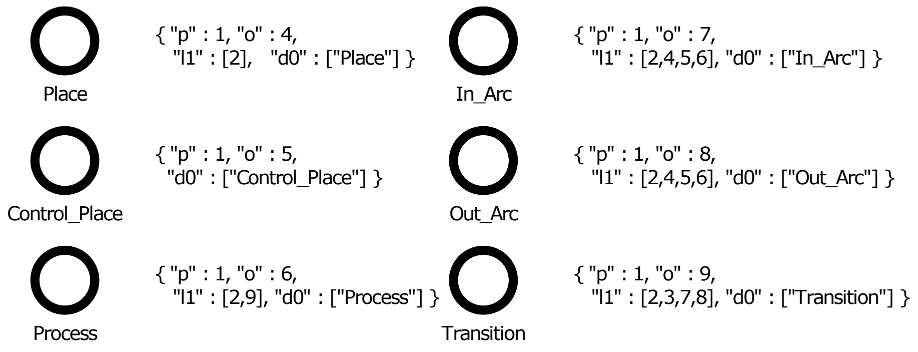

{ "p" : 1, "o" : 4,
"l1" : [2], "d0" : ["Place"] }
Place
{ "p" : 1, "o" : 7,
"l1" : [2,4,5,6], "d0" : ["In_Arc"] }
In_Arc
{ "p" : 1, "o" : 5,
"d0" : ["Control_Place"] }
Control_Place
{ "p" : 1, "o" : 8,
"l1" : [2,4,5,6], "d0" : ["Out_Arc"] }
Out_Arc
{ "p" : 1, "o" : 6,
"l1" : [2,9], "d0" : ["Process"] }
Process
{ "p" : 1, "o" : 9,
"l1" : [2,3,7,8], "d0" : ["Transition"] }
Transition

Fig. 10. JSON description of atoms

as in Fig.8–10. The basic brick is a list of JSON pairs (key, value). This can be done in any object oriented language. We have chosen C++ as our language of implementation.

In the first phase we define all the basic entities (Fig.9 – 10). We enumerate them as indicated in the figures. They are encoded in a JSON format and correspond to objects from Fig.2. We decided to use JSON just to facilitate readability of the object properties by humans. We begin with defining the primitives. There will be 8 basic bricks: Name, Code, Place, Control_Place, Inarc, Outarc, Transition and Process.

For instance the `Place` has only one field `name` and `Process` two fields `Name, Transition`. The other basic classes are generated analogously.

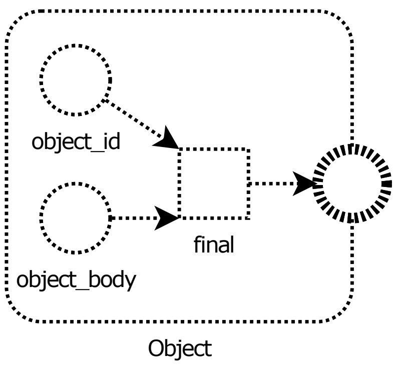


Fig. 11. Wrapper: Object

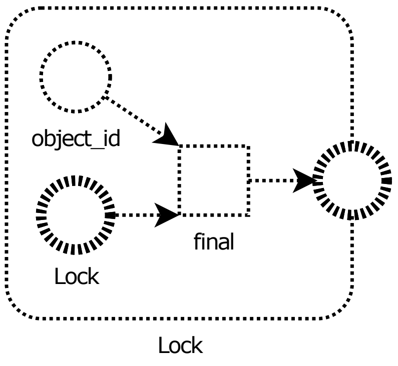


Fig. 12. Wrapper: Lock

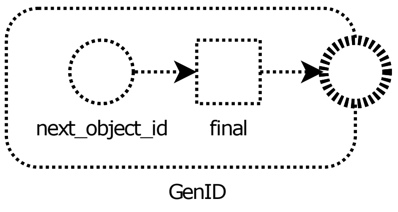


Fig. 13. Wrapper: GenID

3.2 Phase 2

Since we have decided to use C++, we generate in Phase 1 header files, which correspond to the the basic classes. After having generated the header files, we have tools for defining 6 basic processes (atoms) corresponding to components of a Petri net.

Below we present an example of defining process Place using tools which we made in Phase 1. As we can see on Fig.2 process Place consists of 5 objects: the handle (thick place), place "Name", transition "final", In_Arc "name" and Out_Arc "Place"(handle).

Under the net we present its JSON notation.

3.3 API examples

All the codes of phase 2 create objects necessary to generate the processes of phase 3. The JSON description of the objects provides full information necessary to build the atoms and molecules of the system. For example the process Place (Fig.2) consists of 5 objects: Place(Name), Inarc(name), Outarc(place), Transition(final) and Process(place). Similarily other 5 processes (Control place, Process, Inarc, Outarc, Transition) will have got the JSON code as a result of executing phase 2.

Phase 3 will be the normal run of a system. We need a process which will animate the processes.

The next step will be defining a process to run the transitions. We will call this process *Kernel* (Fig.14).

Each process in our model (excluding Kernel), in particular the 6 basic bricks, can be animated with the help of the process Kernel (Fig.14). In the API it means that we can call not only the transitions defined inside the process, but also the transitions of the Kernel (in C++ we exploit the inheritance).

For example the process Place has transitions (methods), which have not been yet defined (edit, set, destroy. . .). They do not belong to the process Place, but to the process Kernel. which is the central concept of the whole approach. This process will be executing the transitions, so running the whole system, being the sentinel of the whole model.

Each process of phase 3 in its API inherits from the class Kernel. We construct the process Kernel analogously to other processes, satisfying the structure from Fig.14.

The process Kernel, mentioned above, contains the life cycle of every object, which is an effect of process activity. The Kernel starts its execution by recognizing the process by its name taken from the place `process_name`, whose name resides in the place `Name` of the process. It executes the transition `start`, which just initiates the whole run. Start has two outcomes. The first of them is creating a token to activate the transition `close`. In general there will be many such transitions, allowing closing the run at any moment. The second one is activation of the transition `create` or `get`. Transition `create` makes an empty object of the chosen process.

On Fig.11 we have a process of accessing objects with given id. The transition final copies the content of its place `object_id` to its handle. The Kernel will be able to copy the content of place `object_body` of the process Object to the place `object_body` being an output place of the transition get.

The transition `get` in Kernel has `object_id` as input and has an access to the database of objects. It finds in the database an object whose identifier (object_id) is in its handle Object and the content of this found object (object_body) is deposited on its place `object_body`, as in Fig.11. The transition `get` of the process Kernel copies the content of the place `object_body` from the process Object to the place object_body in Kernel.

Analogously the database functions Lock and GenID (Fig.12 and 13) will be implemented. The first of them locking the object for editing, and GenID generating a unique identifier.

– All its transitions are written using API of the phase 3 and describe the behaviour consistent with the model defined before.
– Dashed handle place means that the objects of the process Kernel are never stored (they are temporary) and disappear once the process is finished.
– Filled (grey) places are control places (ControlPlace)
– Double-sided arrows mean inarc and outarc connecting a place and a transition.

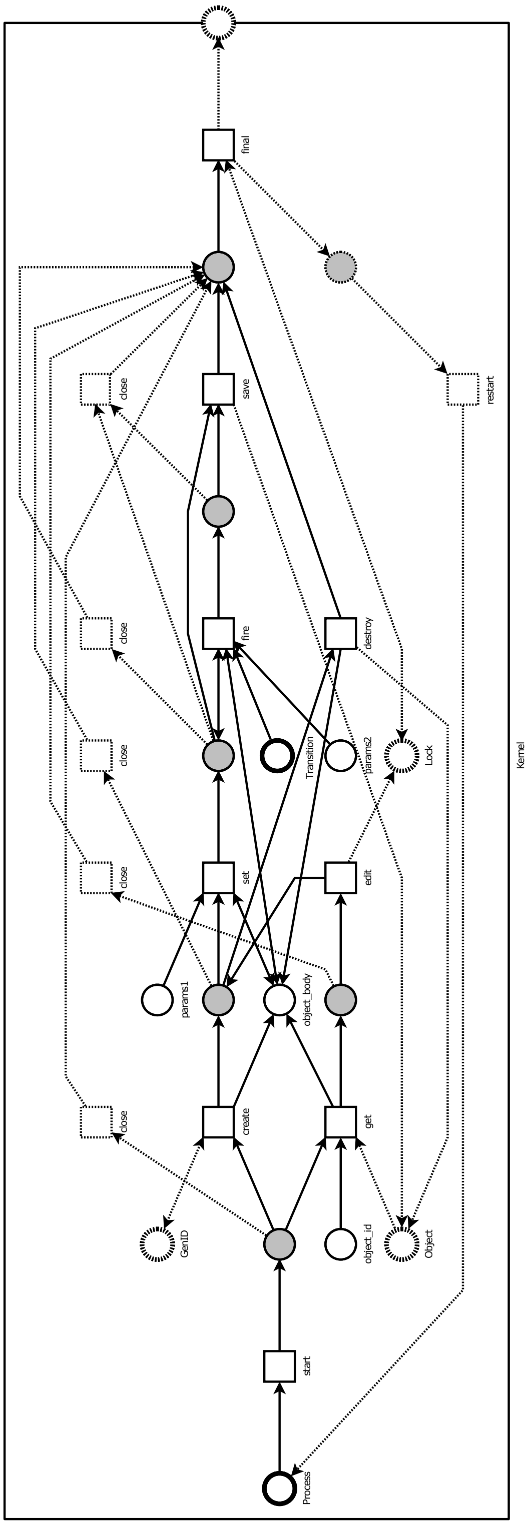


Fig. 14. Kernel

- Special processes are
  - Object — wrapper for database access with the key `object_id` and place `object_body`, in which we keep the body of the object in JSON format.
  - Lock — wrapper for a blocking mechanism, which locks an object with given `object_id` for editing.
  - GenId — wrapper for generation of unique identifiers, being consecutive integer of a number kept in the place `next_object_id`.

  The transitions `get` and `create` are mutually exclusive. The transition `create` uses `GenID` to generate a unique identifier. It gets the process identifier and writes the JSON header (the properties "o" and "p") to being properties `o` to `object_body`. So far only these two header values are known for this process. All these wrappers can be implemented in any way, because only Kernel uses them. They are rather technical details. In the following we refer to phase 3, which will be defined in the next chapter. The recursive nature of the bootstrap requires the assumption of existence of a working engine, which is currently being defined.
- The codes of transitions are texts written as if the phase 3 has already been compiled.
- We assume that the system processes (like Kernel) are generated differently than all the other ones. They are not run by `Kernel` and keep data directly in the C++ variables. When the object is completed, they disappear. The codes of transitions of such processes are just copied to the generated APIs. Our 6 basic objects in phase 3 will inherit the properties from Kernel.
- The process Process has an API generator in its code. Basing on the name of a process it decides, whether the API code should be generated or inherited from the class Kernel. All the processes except Kernel, will inherit from Kernel.

4 Phase 3

We come to the phase, which will become later a normal system behaviour (the loop in Fig.16). We are now in a model, which can compile itself. On Fig.15 we have an interpreter of the system. The header files of the 6 basic molecules (solid places) and Kernel are ready to use.

4.1 Bootstrap

Although the final code of the system could be created independently, so resulting in the final loop of Fig.16, we decided to create it step by step, using the two phases in order to comply with the model itself. We believe that such approach gives additional confidence that the result is compliant with the model and no logical errors has been introduced.

Whenever we wish to update the system and extend it adding some new properties, and changing its design, we need to be sure that it is consistent. Our solution is the bootstrap, so we will compile the system itself. The phases of the bootstrap are:

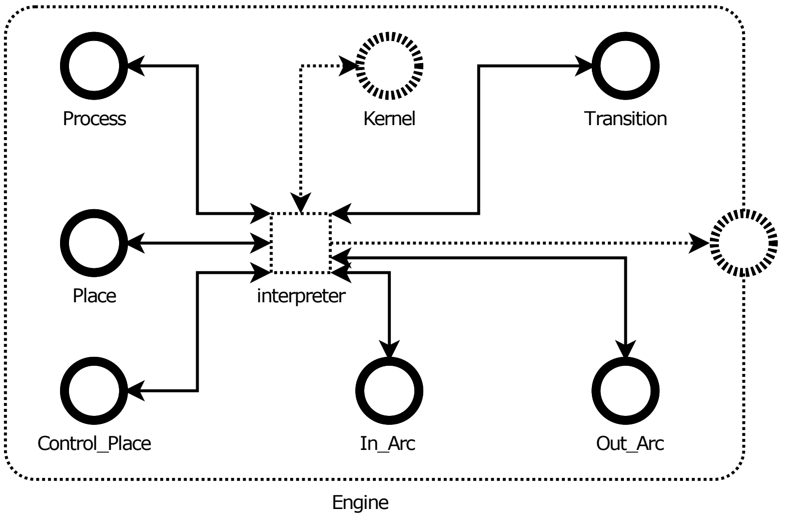


Fig. 15. Engine

- When the process Engine (Fig.15) is run, we substitute the folder with header files by a new one — initially empty — to let Engine create itself from the scratch. The new files *.h will be stored there.
- The script (with the desired changes) containing instructions building the system (almost the same as in phase 2, except that the transitions final are executable codes) is given as input. We assume here that a new version of the system is generated. It will differ from the previous version because the changes made, but yet another run of the same script will produce identical output.
- The newly generated header files *.h are stored in the new folder.
- The system is recompiled and restarted using the new header files
- The same script is provided for input once again.
- If the header files are identical as the ones just generated in the new folder, it means that we have successfully introduced the changes and our system is a Quine [3] (a self-reproducing program).

The basic advantage of bootstrap is that it verifies the model itself. Once it produces itself as an output, we have a certificate that no internal error has been introduced.

The other advantage is that we can use the methodology of that model by means of bootstrap in any Turing complete language or machine (not necessarily being C++, as it is in our case) e.g. Solidity on Ethereum Virtual Machine [9]. The only change will be seen in the language in which the low-level final code and API shape would be different but the framework will be identical.

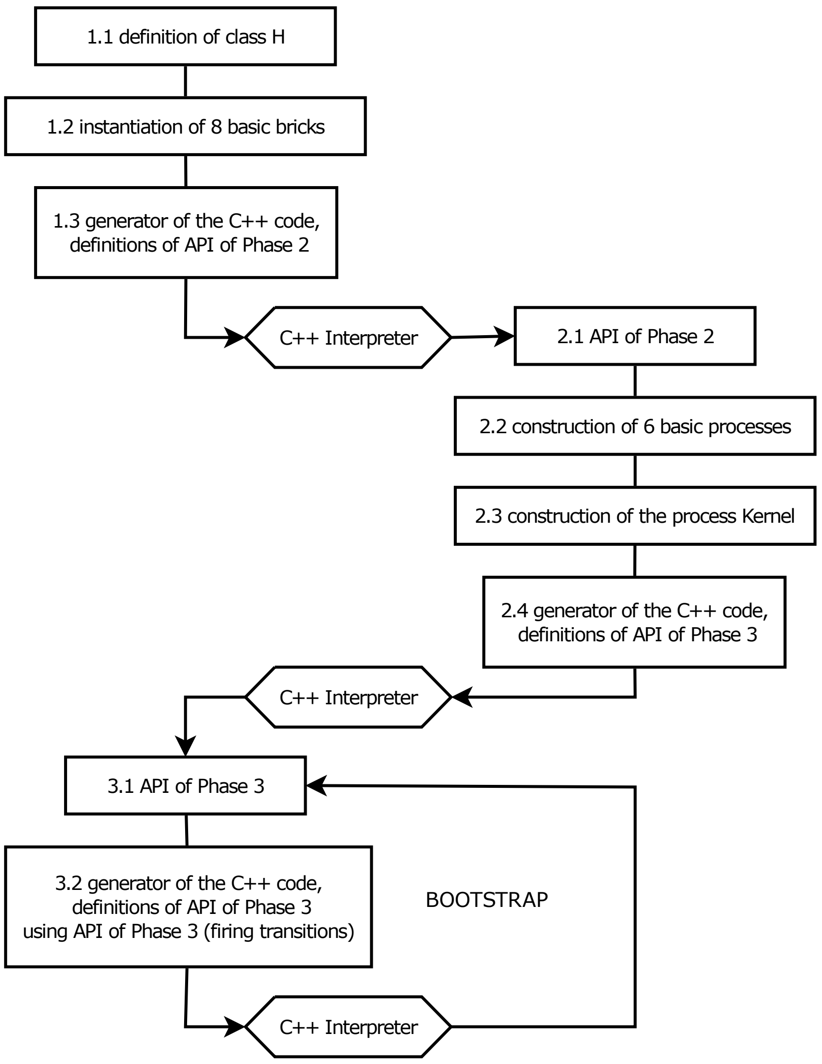

1.1 definition of class H
1.2 instantiation of 8 basic bricks
1.3 generator of the C++ code, definitions of API of Phase 2
C++ Interpreter
2.1 API of Phase 2
2.2 construction of 6 basic processes
2.3 construction of the process Kernel
2.4 generator of the C++ code, definitions of API of Phase 3
C++ Interpreter
3.1 API of Phase 3
3.2 generator of the C++ code, definitions of API of Phase 3 using API of Phase 3 (firing transitions)
BOOTSTRAP
C++ Interpreter

Fig. 16. General scheme

5 Our Experience with the Model

The first version of the model described informally was presented in [1] in the form of DCN (Directed Control Net). We have successfully deployed a few internal business systems based on the prototype of our approach. The prototype — though not completely bootstrapped — fully implements the described model. It allowed us to test its practical properties and confirm the most important advantages.

5.1 Positive experience

An application developed using this model is well structured. It is easy to see the relationships between processes and it's easy to understand the structure and intended purpose of a process. The code is limited in scope to its transition's inputs and outputs. This makes returning to old processes and transition codes easier for maintainers. It also facilitates introduction of new programmers to the project, as they have good overview of the structure and simple pieces of code to work with.

The structure of an application made using the presented model and the fact that data is kept in places following its structure, allows for simple SQL modification that navigates the data structure using dot notation instead of joining tables. This in turn enables programmers to focus on the structure of the process and avoid expressing relationships between data structures using SQL notation. Having metadata implemented with the same model, makes it easy to use the same modified SQL queries to find properties of the application. We can for instance specify, using the net structure, which processes are mutually dependent, or which transitions are allowed to edit objects of the given process. This provides detailed information on the impact that a change made to a process has in the whole application, thus allowing for changes to the application to be made quickly and safely.

What makes our approach useful is the bootstrap, which allows transferring any software written in our system to other platforms. It suffices to encode the phases 1 and 2, and the rest of the system complies itself. These two first phases are indeed very simple and programming them should consume very little time.

The fact that data is kept in places and relationships between processes are readily visible makes finding the origin of object data straightforward. The data can only be written as an effect of running the process that contains the places and that process can only be run according to the model. On top of that, versioning of objects provides a change log that, combined with the model, allows to precisely retrace steps taken by the user to write the resulting objects. This is very helpful in tracking down errors in the application which result in writing data that is flawed from a user's perspective but not excluded by the application. In such cases we can see the whole process that leads to writing of unwanted data and can explain it, so that the solution is agreed upon by the client. This method of finding the origin of object data is also useful for finding out that the flawed data was entered by a user, which usually means there is a need for some optimization in the user's interface so that they are able to better understand it.

The most natural database structure compatible with the model we found was simple ORM (Object Relational Mapping) with support for versioning. Processes are represented with Tables which contain objects with data in Places represented with Fields and few special Fields like object id and version id. Every table containing objects and all corresponding versions constitute an immutable history of the system including the evolution of its metadata.

Characteristics of the resulting database structure with its immutable history of objects appear compatible with the concepts like LevelDB, MVCC (Multi Version Concurrency Control), WAL (Write Ahead Logging) and blockchain [7],[8].

5.2 Future Work

During our journey with the deployment of a few internal business systems we have observed the rise of a strange idea called "blockchain". It was first popularized at scale by the Bitcoin project which was the world's first decentralized crypto-currency and a system which successfully introduces a concept of Byzantine Fault Tolerant Consensus. It means that the Bitcoin system by its protocol produces one version of history of transactions which every participant in the Bitcoin network agreed upon.

Bitcoin transactions are more then just an entry in distributed ledger. They contain simple programs written in a special programming language called *Script*, which is not Turing complete. This limitation is made on purpose because simple system provide better level of security. Early developers of Bitcoin soon realized, that this limitation is bad, because it limits very important applications of Blockchain technology in ex. financial markets.

This was the beginning of the Ethereum Platform which run programs called Smart Contracts in a decentralized way like in Bitcoin protocol, but the language in which they are written is Turing Complete.

Assuming that we have found the right model for the implementation of internal database systems using workflows and observing the fact that the resulting database is very similar to ones used in blockchain platforms we expect that our model could be very useful for smart contracts programming. At the time of writing the most popular blockchain platform using Smart Contracts is Ethereum [9].

We can establish hypothetical mapping between the presented model and the one used by Ethereum platform:

- System = DApp (Decentralised Application),
- Process = Smart Contract,
- Object = Transaction,
- Transition = Method of Smart Contract,

Places and Arcs limit the data access of Methods and define partial order of their execution.

We can translate our model by means of bootstrap to any Turing complete language or machine e.g. Solidity on Ethereum Virtual Machine. Systems consisting of SmartContracts written in compliance with the Model can be formally analyzed by known Petri Net methods. It can significantly improve quality of Smart Contract code and eliminate many errors which in the domain of Ethereum platform, can cost large amounts of money. In his book [8] Andreas M. Antonopoulos diagnoses a fundamental problem with the Ethereum platform: "One of the big challenges facing developers in Ethereum is the inherent contradiction between deploying code to an immutable system and a development platform that is still evolving. You can't simply "upgrade" your smart contracts. You must be prepared to deploy new ones, migrate users, apps, and funds, and start over.[. . .] Ironically, this also means that the goal of building systems with more autonomy and less centralized control is still not fully realized." We suspect that the problem is unsolvable without a right model directing expressive power of a Turing Machine in a productive way. We believe that the model proposed in this publication is fitting for Smart Contracts and can get traction in the future along with growing importance of Smart Contracts and Blockchains platforms.

6 Acknowledgements

The authors express gratefulness to an anonymous referee, whose comments helped to improve the paper substantially.

References

1. T. Wójcicki, M. Doleżek "System and method for specifying and implementing IT systems" United States Patent US8423580B2 Apr. 16, 2013
2. Schorre, D. V. [1964]. "Meta-II: a syntax-oriented compiler writing language" Proc. 19th ACM National Conf., D1.3-1 - D1.3-11, [1964].
3. David Madore "Quines (self-replicating programs)" [online] http://web.archive.org/web/20190102192930/ http://www.madore.org:80/~david/computers/quine.html
4. Piotr Chrząstowski-Wachtel, Boualem Benatallah, Rachid Hamadi, Milton O'Dell, Adi Susanto: A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling. Business Process Management 2003: 336-353 [2003]
5. van der Aalst,W., van Hee, K.,Workflow Management: Models, Methods, and Systems, MIT Press [2004]
6. Jan Mendling, Hajo A. Reijers, Marcello La Rosa, Marlon Dumas, Fundamentals of Business Process Management, Springer Verlag [2013]
7. Nakamoto, Satoshi (31 October 2008). "Bitcoin: A Peer-to-Peer Electronic Cash System". [online] http://web.archive.org/web/20100704213649/ http://www.bitcoin.org/bitcoin.pdf [2008]
8. Andreas M. Antonopoulos "Mastering Bitcoin" O'Reilly Media, Inc. 2014-12-01 First release, [2014]
9. Buterin. V. "A Next-Generation Smart Contract and Decentralized Application Platform" https://web.archive.org/web/20140723210954/www.ethereum.org/pdfs/EthereumWhitePaper.pdf [online] [2014]
10. Andreas M. Antonopoulos "Mastering Ethereum" O'Reilly Media, Inc. 2018-11-13 First release, [2018]