mathematics

MDPI

Article

Thread Algebra with Prospecting Services and Foresight Patterns

Jan Bergstra

Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands; j.a.bergstra@uva.nl or janaldertb@gmail.com

Abstract: Thread algebra is a domain-specific process algebra which may be used for semantic work on sequential systems, including systems based on deterministically scheduled multi-threading. Thread algebra is used in this capacity with the forecasting phenomenon for programs and machines as a domain of interest. Several new informal notions are proposed: prospecting services, foresight patterns for systems, and lookahead conditions as a mechanism for the specification of services. Some new prospecting services are proposed which facilitate the realisation of certain foresight patterns. Several negative results about the non-realisability of certain foresight patterns are provided.

Keywords: process algebra; thread algebra; program algebra

MSC: 68N30; 68Q10; 68Q58

1. Introduction

Thread algebra [1] is a domain-specific process algebra, which is tailor made for providing semantics for imperative programs in the form of instruction sequences written in the notation of the program algebra PGA of [2]. For introductions to program algebra, instruction sequences and thread algebra we refer to [3], and for more comprehensive information, to [4]. For more information on thread algebra we mention [5,6]. A comprehensive example of the use of notations from process algebra and thread algebra can be found in [7]. Below, the phrase "instruction sequence" will refer to an instruction sequence in one of the notations provided by program algebra, and thread will refer to an element of the domain of a thread algebra.

Thread algebra offers constants `S` (stop) for a properly terminating thread and `D` (deadlock/divergence) for an improperly terminating thread. With threads `P` and `Q` also $R \equiv \mathtt{P} \trianglelefteq \mathtt{f.m} \trianglerighteq \mathtt{Q}$ is a thread. A non-terminating (run of a) thread performs an infinite number of subsequent method calls. Non-termination differs from divergence in that divergence may involve an infinite number of subsequent internal steps, while non-termination involves an infinity of externally visible steps, viz. the subsequent method calls.

The idea is that `f.m` constitutes a method call, where a component accessible under the name (focus) `f` is asked to process a method call `m`. Processing a method will change the state of the component which performs that task, and moreover, it will produce a result in the form of a Boolean value. If `true` is returned, the thread `R` proceeds as `P` and if `false` is returned, the thread proceeds as `Q`. Components able to process method calls are called services. A focus `f` and a service `H` made accessible under focus `f` are combined as a service in focus `f.H`. A thread and a service in focus can cooperate in different ways: use (denoted $\mathtt{P/f.H}$), apply (denoted $\mathtt{P} \bullet \mathtt{f.H}$), and reply (denoted $\mathtt{P!f.H}$). In the case of use, the service is used for computing a thread; in the case of apply, the thread transforms the service which will contain inputs before a run of the thread is started and outputs upon termination of the run said thread; and in case of reply, a mere Boolean value is computed. Use and apply represent extremes of the different way a sequential process may deal with data: data may

check for updates

Citation: Bergstra, J. Thread Algebra with Prospecting Services and Foresight Patterns. *Mathematics* **2022**, *10*, 2232. https://doi.org/10.3390/math10132232

Academic Editor: William Sulis

Received: 13 May 2022
Accepted: 17 June 2022
Published: 26 June 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

support the computation of a more complicated process, and the sequential process is used for computing a function which is applied to the data. Below, we will only consider the case of use.

A representation of thread algebra terms of the process algebra ACP (as defined in [8]) is given in [4,9]. The idea is that a thread P is represented by a process $\natural(\mathtt{P})$ and a service $\mathtt{H}$ is represented by a process $\natural(\mathtt{H})$. $\natural$ is the thread/service to process projection, which may be alternatively be denoted by $\mathtt{ts2p}$. In particular, using ACP notation for actions for sending and reading along a port, and communication thereof (as used in [10] and later):

$$\natural(\mathtt{P} \trianglelefteq \mathtt{f.m} \trianglerighteq \mathtt{Q}) = \mathtt{s_f(m)} \cdot (\mathtt{r_f(true)} \cdot \natural(\mathtt{P}) + \mathtt{r_f(false)} \cdot \natural(\mathtt{Q}))$$

$\mathtt{P} \bullet \mathtt{f.H}$ is represented by a parallel composition $\natural(\mathtt{P} \bullet \mathtt{f.H}) = \partial_{\mathtt{RS}}(\natural(\mathtt{P})||\natural(\mathtt{f.H}))$ of a representation $\natural(\mathtt{P})$ of $\mathtt{P}$ and a representation $\natural(\mathtt{f.H})$ of $\mathtt{f.H}$. Here $\partial_{\mathtt{RS}}$ is the encapsulation operator which guarantees that corresponding send and receive actions, together collected in the set $\mathtt{RS}$, will synchronize. A state of a service $\mathtt{H(s)}$ then takes the following form:

$$\natural(\mathtt{H(s)}) = \sum_{\mathtt{m} \in \mathtt{J}} (\mathtt{r(m)} \cdot \mathtt{s}(\theta(\mathtt{m},\mathtt{s})) \cdot \natural(\mathtt{H}(\eta(\mathtt{m},\mathtt{s},\theta(\mathtt{m},\mathtt{s})))))$$

Here $\theta(\mathtt{m},\mathtt{s})$ is a Boolean condition which determines the reply value of the service on the call of method $\mathtt{m}$: if $\theta(\mathtt{m},\mathtt{s}) = \mathtt{true}$, then the reply value is $\mathtt{true}$ and if $\theta(\mathtt{m},\mathtt{s}) = \mathtt{false}$, then the reply value is $\mathtt{false}$. The function η determines the state of the service after a reply has been returned. The follow-up state depends on the original state $\mathtt{s}$, the method involved and the reply value $\theta(\mathtt{m},\mathtt{s})$ that has been returned.

For an ordinary service $\mathtt{H}$, $\theta(-,-)$ the so-called reply condition, and $\eta(-,-,-)$, the so-called effect function, are functions. In the case of an infinite state space, it is plausible that both functions are computable. Throughout the paper, we will assume that η is a function, while θ may depend on other attributes of a system in which the service operates. If $\theta(-,-)$ is a function, i.e., it depends on $\mathtt{m}$ and $\mathtt{s}$ only, the service is called ordinary.

The following equivalent form is more suggestive about the fact that for each method call, the corresponding reply conditions must be evaluated just once. Here, $\mathtt{x} \lhd \phi \rhd \mathtt{y}$ is the "if ϕ then x else y" function with the condition written as the argument in the middle. When located under focus $\mathtt{f}$ the send and receive actions are renamed, the focus $\mathtt{f}$ being used as a port name for the process algebra notation, we obtain $\natural(\mathtt{f.H(s)}) = \sum_{\mathtt{m} \in \mathtt{J}}(\mathtt{r_f(m)} \cdot ((\mathtt{s_f(true)} \cdot \natural(\mathtt{H}(\eta(\mathtt{m},\mathtt{s},\mathtt{true}))))) \lhd \theta(\mathtt{m},\mathtt{s}) \rhd (\mathtt{s_f(false)} \cdot \natural(\mathtt{H}(\eta(\mathtt{m},\mathtt{s},\mathtt{false})))))))$ A trivial manner in which the reply condition can be non-functional is to have it dependent of the focus by which the service is made accessible (using multiline notation for better readability and subscripts for the reply condition and effect function so that unambiguous reference to these from outside the architerm can be made).

$$\begin{aligned}\natural(\mathtt{f.H(s)}) = \sum_{\mathtt{m} \in \mathtt{J}}(\mathtt{r_f(m)} \cdot & \\ & ((\mathtt{s_f(true)} \cdot \natural(\mathtt{H}(\eta_{\mathtt{H}}(\mathtt{m},\mathtt{s},\mathtt{true})))) \\ & \quad \lhd \theta_{\mathtt{H}}(\mathtt{m},\mathtt{s}) \rhd \\ & \quad (\mathtt{s_f(false)} \cdot \natural(\mathtt{H}(\eta_{\mathtt{H}}(\mathtt{m},\mathtt{s},\mathtt{false}))))) \\ &)\end{aligned}$$

Multiple services, $\mathtt{H_1},\ldots,\mathtt{H_n}$ accessible under pairwise different foci $\mathtt{g_1},\ldots,\mathtt{g_n}$ are combined into a single service using the service composition operator $\oplus$: $\mathtt{g_1.H_1} \oplus \ldots \oplus \mathtt{g_n.H_n}$. For the details of service composition, we refer to [4]. In terms of the process algebra, one finds $\natural(\mathtt{g_1.H_1} \oplus \ldots \oplus \mathtt{g_n.H_n}) = \natural(\mathtt{g_1.H_1})|| \ldots ||\natural(\mathtt{g_n.H_n})$, at least in case the $\mathtt{g_i}$ are pairwise different. Thread algebra allows different representations in terms of process algebra. For instance, for some applications, an interpretation into a discrete time process algebra (see, for example, [11]) is preferable. Below, the thread algebra notation is used unless the additional detail of how a basic action is realised via a pair of atomic actions matters.

A thread $\mathtt{P}$ may be defined in different ways. First of all, threads can be defined, like processes in process algebra, by means of recursion equations or systems of recursion equations; for instance $\mathtt{P} = \mathtt{S} \trianglelefteq \mathtt{f.m1} \trianglerighteq (\mathtt{P} \trianglelefteq \mathtt{g.m2} \trianglerighteq \mathtt{D})$. Otherwise, comparable with the use of a transition system as a process definition in process algebra, a thread, say $\mathtt{P}$, can be defined by means of thread extraction from an instruction sequence, say $\mathtt{X}$: $\mathtt{P} = |\mathtt{X}|$. An expression of the form $\partial_{\mathtt{RS}}(\mathtt{P} \bullet \mathtt{g}_1.\mathtt{H}_1 \oplus \ldots \oplus \mathtt{g}_n.\mathtt{H}_n)$ is referred to as an architerm, it describes a so-called execution architecture for $\mathtt{X}$, in the terminology of [12].

Process algebras contain behaviours as elements and provide composition operators for these. Many different process algebras have been designed, and process algebras can be tailor made for certain application areas. Conventionally, process algebras have been studied in the context of parallel processes so that parallel composition is the primary composition operator. In thread algebra, behaviours represent sequential systems, including deterministic systems that result from scheduled concurrent compositions of sequential systems (see [1,5]).

Instruction sequences, as well as threads, allow, as much as possible, the separation of data and control. For no datatype, not even for Booleans, the presence is taken for granted in the setting of program algebra. Only Boolean feedback from operations on data (i.e., method calls), which are located in services, is used to influence the progression of control.

1.1. Prospecting, Foresight and Lookahead Conditions

We will discuss prospecting in the context of thread algebra. Prospecting allows an agent participating in a system to survey (aspects of) the potential future behaviour of one or more other agents in the system. We model prospecting as a capability of a service. We use the term prospecting rather than the perhaps more appealing term lookahead in order to diminish the connotation of statistically sound inference from available data. Dealing with prospecting in a process algebra setting seems to be harder but may follow suit just as, for instance, strategic interleaving was first defined in the context of thread algebra in [1] and only thereafter in a setting of process algebra (see, for example, [13]).

Foresight constitutes the apparently successful ability of an agent in a system to take future options of behaviour of the entire system into account. We use the term foresight rather than forecasting, also in order to weaken the connotation of inference based on scientifically rigorous methods. Indeed prospecting, as meant below, may be achieved by way of magic, and forecasting may be based on unexplained or non-rigorous forecasting. Foresight is a phenomenon which concerns a system as a whole because it combines the intention of an agent in the system to take expectations or knowledge about the entire system (including the agent) into account as well as some assessment of the success of the agent in that respect.

Prospecting may allow successful foresight. One may think of an agent as a robot $\mathtt{R}$ the software of which is known to another agent $\mathtt{P}$ in the form of a compiled source program. Somehow $\mathtt{P}$ may be able to make useful predictions about the behaviour of $\mathtt{R}$ on the basis of that information. We will not look into this particular scenario, and it is mentioned only in order to indicate that it is not inconceivable that agent $\mathtt{P}$ uses intelligence to obtain meaningful information regarding the future behaviour of another component $\mathtt{R}$.

An example of prospect and foresight is as follows. With $\mathtt{I}_{\mathsf{method}}(\mathtt{H})$, We will denote the method interface of $\mathtt{H}$, and a detailed description of interfaces is given in Section 2.4. Let $\mathtt{m1} \in \mathtt{I}_{\mathsf{method}}(\mathtt{K}_1)$ and $\mathtt{m2} \in \mathtt{I}_{\mathsf{method}}(\mathtt{K}_2)$. Now consider the following architerm: $\mathtt{A} = ((\mathtt{g}_2.\mathtt{m2} \circ \mathtt{S}) \trianglelefteq \mathtt{g}_1.\mathtt{m1} \trianglerighteq \mathtt{S}) \trianglelefteq \mathtt{h.find} \trianglerighteq (\mathtt{S} \trianglelefteq \mathtt{g}_1.\mathtt{m1} \trianglerighteq (\mathtt{g}_2.\mathtt{m2} \circ \mathtt{S}))) \bullet \mathtt{h.H} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$.

The task is to design a service $\mathtt{H}$ with method interface $\mathtt{I}_{\mathsf{method}}(\mathtt{H}) = \{\mathtt{find}\}$ which works in such a manner that in all circumstances, $\mathtt{A} = \mathtt{A}(\mathtt{H}, \mathtt{K}_1, \mathtt{K}_2)$ will perform the basic action $\mathtt{g}_2.\mathtt{m2}$. Suppose that $\mathtt{H}$ is an ordinary service, say $\mathtt{H}(\mathtt{s}_0)$ with state space $\mathtt{S}_\mathtt{H}$ and $\mathtt{s}_0 \in \mathtt{S}_\mathtt{H}$, and with reply condition $\theta_\mathtt{H}(\mathtt{find}, \mathtt{s})$ and effect function $\eta_\mathtt{H}(-,-,-)$. Now suppose

$\theta_{\mathtt{H}}(\mathtt{find}, \mathtt{s}_0) = \mathtt{true}$, then on the first call of a method (in this case $\mathtt{h.find}$) a reply $\mathtt{true}$ is received so that the next state of the computation will be

$$\mathtt{A}^{\star} = ((\mathtt{g}_2.\mathtt{m2} \circ \mathtt{S}) \trianglelefteq \mathtt{g}_1.\mathtt{m1} \trianglerighteq \mathtt{S}) \bullet \mathtt{h.H}(\eta_{\mathtt{H}}(\mathtt{s}_0, \mathtt{find}, \mathtt{true})) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$$

Now assume that service $\mathtt{K}_1$ produces reply $\mathtt{false}$ on the first call of method $\mathtt{m1}$. Then (writing $\mathtt{t}_0$ for the initial state of $\mathtt{K}_1$) the subsequent state is

$$\mathtt{A}^{\star\star} = \mathtt{S} \bullet \mathtt{h.H}(\eta_{\mathtt{H}}(\mathtt{s}_0, \mathtt{find}, \mathtt{true})) \oplus \mathtt{g}_1.\mathtt{K}_1(\eta_{\mathtt{K}}(\mathtt{t}_0, \mathtt{m1}, \mathtt{false})) \oplus \mathtt{g}_2.\mathtt{K}_2$$

Clearly, $\mathtt{A}^{\star\star}$ will not perform basic action $\mathtt{g}_2.\mathtt{m2}$. A similar argument can be given if it is assumed that $\theta_{\mathtt{H}}(\mathtt{find}, \mathtt{s}_0) = \mathtt{false}$. It follows that an ordinary service $\mathtt{H}$ cannot have the required property that always a call $\mathtt{g}_2.\mathtt{m2}$ will take place.

Next, with a leap of imagination, one may allow the reply condition $\theta_{\mathtt{H}}(\mathtt{find}, \mathtt{s}_0)$ to include $\mathtt{K}_1$ as an additional argument and to allow $\mathtt{H}$ to experiment with the (hypothetical) future behaviour of $\mathtt{K}_1$. Now consider the following reply condition for $\mathtt{H}$: $\widehat{\theta}_{\mathtt{H}}(\mathtt{find}, \mathtt{s}_0) \equiv [\mathtt{K}_1(\mathtt{m1}) = \mathtt{true}]$ (i.e. $\theta_{\mathtt{K}_1}(\mathtt{m1}, \mathtt{t}_0) = \mathtt{true}$). This new condition is permitted to acquire prospective information about another service in the architerm $\mathtt{A}$. Let $\mathtt{H}(\widehat{\theta}_{\mathtt{H}})$ denote the service obtained from $\mathtt{H}$ by replacing the reply condition $\theta(\mathtt{find}, \mathtt{s}_0)$ by $\widehat{\theta}(\mathtt{find}, \mathtt{s}_0)$ then the execution architecture $\mathtt{A}(\mathtt{H}(\widehat{\theta}), \mathtt{K}_1, \mathtt{K}_2)$ has the required property.

We will say that $\widehat{\theta}$ is a lookahead condition which allows $\mathtt{H}(\widehat{\theta})$ to perform prospecting, a state of affairs which then explains the fact that in configurations of the form $\mathtt{A}(\mathtt{H}(\widehat{\theta}), \mathtt{K}_1, \mathtt{K}_2)$ a phenomenon of foresight can be observed. In this paper, we will demonstrate some examples of lookahead conditions that may serve as non-ordinary reply conditions. The design of a most general class of such conditions is non-trivial so it seems, and the following problem is left open.

Problem 1. *Is there a plausible definition of which conditions may occur as (non-ordinary) reply conditions, so that prospection can be "exploited" to its limits?*

This question applies in particular to configurations where more than one service may use lookahead conditions.

1.2. Configurations and Intermediate Configurations

Configurations capture both initial stages and intermediate stages for computations in an obvious manner: step after step, the basic actions of the thread are performed and the states of various services are updated. Final states of such computations are mere service families from which the thread has become trivial: either $\mathtt{S}$ or $\mathtt{D}$.

The term configuration refers to a threads/service configuration. A typical (generic) configuration has the form $\mathtt{C} \equiv \mathtt{P} \bullet \mathtt{g}_1.\mathtt{K}_1 \oplus \ldots \oplus \mathtt{g}_n.\mathtt{K}_n$. In the setting of thread algebra configurations compute by performing steps which correspond to (i) a method call being made, (ii) a reply being computed, and (iii) the (side) effect on the state of the service which is processing the call that is computed. Upon looking at the same run from the perspective of process algebra, additional precision can be obtained, and intermediate states and configurations appear. We will need some notation for intermediate configurations and components thereof.

For a thread $\mathtt{P}$, $\ddagger(\mathtt{P})$ will denote that same thread, though with its first atomic action (when present) missing. This idea can be made formal via the translation to processes: $\natural(\ddagger(\mathtt{P}))$ is the unique process p such that for some atomic action $\mathtt{a}$ it is the case that $\natural(\mathtt{P}) = \mathtt{a}.\mathtt{p}$ if such a decomposition of $\mathtt{P}$ atomic action exists and $\natural(\mathtt{P}) = \delta$ (the inactive process in ACP) otherwise. In particular, one finds: $\natural(\ddagger(\mathtt{P} \trianglelefteq \mathtt{f.m} \trianglerighteq \mathtt{Q})) = \mathtt{r}_{\mathtt{f}}(\mathtt{true}) \cdot \natural(\mathtt{P}) + \mathtt{r}_{\mathtt{f}}(\mathtt{false}) \cdot \natural(\mathtt{Q})$.

Moreover for two services $\mathtt{H}_1, \mathtt{H}_2$ with the same method interface $\mathtt{J}$, let $[\mathtt{H}_1] \lhd_{\mathtt{f}} \theta \rhd_{\mathtt{f}} [\mathtt{H}_2]$ denote a service, present under focus $\mathtt{f}$, in the state just after having received a method call and going to reply $\mathtt{true}$, then moving on as $\mathtt{H}_1$ if the reply condition θ turns out to be

true and going to reply $\mathtt{false}$, while continuing as the service $\mathtt{H}_2$ otherwise. Upon reading threads as process with $\natural$, one finds

$$\natural([\mathtt{H}_1]\lhd_{\mathtt{f}}\theta\rhd_{\mathtt{f}}[\mathtt{H}_2]) = (\mathtt{s}_{\mathtt{f}}(\mathtt{true})\cdot\natural(\mathtt{f}.\mathtt{H}_1))\lhd\theta\rhd(\mathtt{s}_{\mathtt{f}}(\mathtt{false})\cdot\natural(\mathtt{f}.\mathtt{H}_2))$$

Now consider the configuration $\mathtt{A} \equiv (\mathtt{P}\trianglelefteq\mathtt{f}.\mathtt{m}\trianglerighteq\mathtt{Q})\bullet\mathtt{f}.\mathtt{H}(\mathtt{s})\oplus\mathtt{g}_1.\mathtt{K}_1\oplus\ldots\oplus\mathtt{g}_{\mathtt{n}}.\mathtt{K}_{\mathtt{n}}$. As a process, $\natural(\mathtt{A})$ takes smaller steps (one atomic action at a time) than as a thread/service configuration (one basic action at a time). After performing one atomic action (sending $\mathtt{m}$ to $\mathtt{H}$ via port f), the following configuration $\mathtt{A}'$ is obtained:

$$\begin{aligned}&\mathtt{A}' \equiv \ddagger(\mathtt{P}\trianglelefteq\mathtt{f}.\mathtt{m}\trianglerighteq\mathtt{Q}))\bullet\\ &\mathtt{f}.([\mathtt{H}_1(\eta_{\mathtt{H}_1}(\mathtt{s},\mathtt{m},\mathtt{true})]\lhd_{\mathtt{f}}\theta(\mathtt{s},\mathtt{m})\rhd_{\mathtt{f}}[\mathtt{H}_2(\eta_{\mathtt{H}_2}(\mathtt{s},\mathtt{m},\mathtt{false})])\oplus\mathtt{g}_1.\mathtt{K}_1\oplus\ldots\oplus\mathtt{g}_{\mathtt{n}}.\mathtt{K}_{\mathtt{n}}.\end{aligned}$$

The advantage of considering $\mathtt{A}'$ is that it provides a precise picture of the situation in which a reply condition determines which reply is to be returned after a method call has been received and the corresponding reply has been computed by the service at hand, and how the service at hand will proceed. This picture allows an informal answer to Problem 1.

Definition 1 (Informal definition of lookahead conditions)**.** *In the configuration* $\mathtt{A}'$*, a lookahead condition* $\theta(\mathtt{s},\mathtt{m})$ *may depend on*

(i) all focus and method names which play a role in $\mathtt{A}'$*,*
(ii) the threads $\mathtt{P}$ *and* $\mathtt{Q}$*, and*
(iii) the services $\mathtt{K}_1,\ldots,\mathtt{K}_{\mathtt{n}}$*.*

What matters is that dependence on the services $\mathtt{H}_1$ and $\mathtt{H}_2$ is not permitted. The reason to be so cautious about $\mathtt{H}_1$ and $\mathtt{H}_2$ lies in the observation that it is more likely than not that the same lookahead condition $\theta(\mathtt{m},\mathtt{s})$ occurs in these services as well. That may happen, for instance, if $\mathtt{H}_1$ and $\mathtt{H}_2$ are chosen as both continuations of a single service $\mathtt{H}$ upon having received method call $\mathtt{f}.\mathtt{m}$ in state $\mathtt{s}$ and if after a number of steps, the same state is visited once more during the computation of $\mathtt{A}'$. A complicated self-reference might result, which is now excluded, however.

In this manner, a reasonable upper bound to the notion of a lookahead condition in the context of $\mathtt{A}'$ is obtained. A meaningful restriction, however, is that $\theta(\mathtt{s},\mathtt{m})$ is built up by logical connectives and quantifiers over primitive assertions, which can be understood as expressing outcomes of hypothetical experiments with configurations of the form

$$\mathtt{A}' \equiv \ddagger(\mathtt{P}\trianglelefteq\mathtt{f}.\mathtt{m}\trianglerighteq\mathtt{Q}))\bullet\mathtt{f}.([\mathtt{L}_1]\lhd_{\mathtt{f}}\theta(\mathtt{s},\mathtt{m})\rhd_{\mathtt{f}}[\mathtt{L}_2])\oplus\mathtt{g}_1.\mathtt{K}_1\oplus\ldots\oplus\mathtt{g}_{\mathtt{n}}.\mathtt{K}_{\mathtt{n}}$$

for various ordinary services $\mathtt{L}_1$ and $\mathtt{L}_2$ with the same method interface $\mathtt{J}$ as is required for $\mathtt{H}_1$ and $\mathtt{H}_2$ in the configuration $\mathtt{A}'$.

Below, we will make use of the following notation: $\mathtt{P}\bullet\mathtt{f}_1.\mathtt{H}_1\oplus\cdots\oplus\mathtt{f}_{\mathtt{n}}.\mathtt{H}_{\mathtt{n}}\ \underline{\mathtt{sat}}\ \mathtt{CALL}(\mathtt{f}_{\mathtt{i}}.\mathtt{m})$ expresses that after zero or more intermediate steps (counting atomic actions as steps), the computation starting with the architerm $\mathtt{B}\equiv\mathtt{P}\bullet\mathtt{f}_1.\mathtt{K}_1\oplus\cdots\oplus\mathtt{f}_{\mathtt{n}}.\mathtt{K}_{\mathtt{n}}$ will perform a call $\mathtt{f}_{\mathtt{i}}.\mathtt{m}$ (or more precisely, $\natural(\mathtt{B})$ will perform an action $\mathtt{c}_{\mathtt{f}}(\mathtt{m})$ which constitutes the synchronisation of a send action and a receive action with content $\mathtt{m}$ along port $\mathtt{f}$ (in ACP style process algebra terminology). Taking $\mathtt{A}'$ for $\mathtt{B}$ the condition $\mathtt{A}'\ \underline{\mathtt{sat}}\ \mathtt{CALL}(\mathtt{f}.\mathtt{m})$ is an example of an assertion which can be understood as expressing information concerning outcomes of experimentation with runs of $\mathtt{A}'$ and, for that reason, as a lookahead condition.

1.3. Thread Algebra, an ACP Representable Domain Specific Process Algebra

An action $\mathtt{f}.\mathtt{m}$ in the context of program algebra and thread algebra is called a basic action. Seen from the viewpoint of process algebras, basic actions are not atomic actions. Instead, a basic action can be explained in terms of synchronous communication of atomic actions. As was outlined above, thread algebra, services, and service composition can be interpreted in process algebra, and in fact, different such interpretations exist. We

prefer to understand this kind of interpretation as a representation. The process algebra provides a well-known semantic standard based on bisimulation semantics, while the thread algebra is to some extent a matter of ad hoc design. By thinking in terms of representations of thread algebra in process algebra, the conceptual primacy of process algebra is emphasised. Upon adopting a specific representation, e.g., as indicated above, the use of thread algebra instead of the underlying process algebra becomes merely a matter of convenience. Thread algebra does not come with the ambition to acquire any new semantic insights or principles for concepts, such as sequential composition, concurrent composition, atomicity of actions, abstraction from internal (silent) steps, and the proper definition and use of infinite state systems. Thread algebra is a domain-specific process algebra that features two kinds/classes of domain-specific processes: threads and service families. Thread algebra offers several domain-specific combinators on these domain-specific classes of processes (apply, use, and reply), each of which are instances of reactive concurrent composition when understood from the perspective of process algebra. Thread algebra is ACP representable in the sense that there is a preferred representation ($\natural$) of thread algebra into ACP-style process algebra.

Thread algebra is one of several domain-specific process algebras that have been developed on the basis of representation in ACP. Process algebra based on asynchronous communication constitutes a domain-specific process algebra which may be further specialised for use with dataflow networks. Alternatively, systolic arrays may be modelled with a highly concurrent version of ACP. The notation μCRL [14] and its successor mCRL2 [15] may be understood as domain-specific process algebras which exploit data as parameters of processes rather than to have as separate semantic entities wrapped in a dedicated process, or as parameters of an additional operator (viz. the state operator of [16]).

For threads and services, the notion of an interface is vital. Interface calculus is a "sister" of the alpha–beta calculus for ACP in [17]. As it turns out, due to the built-in symmetries in ACP, the idea of an interface boils down to a set of actions, which is relatively simple compared with the notion of an interface, as it is needed for the setup of thread algebra.

Thread algebra involves threads and services, representing two types of agents with threads having a focus on control, and services a focus on data. Thread algebra is more specific for sequential, deterministic, and reactive systems and, therefore, less general than process algebra, which is meant to capture arbitrary concurrent systems.

Different forms of process algebra come into play for different themes. In this paper, we will look at prospecting, both phenomena which one may at least hypothetically ascribe to the behaviour of an agent, and foresight, a capability which, again hypothetically, may be ascribed to the participation of an agent in a system.

For software engineering, the idea of prospecting is rather uncommon, though very common in processor pipelines. Prospecting is usually expected of rational agents. Prospecting and foresight play a role already in sequential systems, and for that reason, the investigation of these topics may well start with thread algebra rather than in process algebra.

Process algebra is a tool for analysing the options for mutual communication and understanding in a multi-agent network. Such networks are thematic for AI. Thread algebra, and an underlying program algebra, provide specialised algebraic approaches for dealing with sequential subsystems of such networks.

Prospecting services were studied in [12] (then named forecasting services) and in [4], where it was shown that such services cannot correctly forecast halting when giving replies in a two-valued logic only. In [4], different forms of the halting problem are formalised in the context of thread algebra. Apparently, thread algebra is better suited for analysing such applications, where control and data are to be treated on equal footing than process algebras, where control is the dominant feature and data are an "add on" feature to some extent.

1.4. Process Algebra, Thread Algebra, and AI

Process algebra can help to analyse the boundaries of what can be computed. What Turing machines do for computability, allowing techniques for analysing what can and

what cannot be done, can be repeated and adapted in a plurality of models of concurrency when it comes to analysing the notion of algorithmic impossibility for interacting agents. Process algebra, as an approach with a plurality of instances, each of which serves as a model for concurrency, allows some uniformity for the analysis of such models. Analysing what cannot be done in qualitative terms is of relevance for AI as much (or even more) as it has been for computer science.

When it comes to proving the impossibility of computational phenomena, arguments often depend on diagonalisation. At face value, the liar paradox and the Russel paradox on the one hand and undecidability of the halting problem and the incompleteness of the axioms of Peano arithmetic (following Gödel) are similar phenomena. However, there is a difference: the liar paradox and the Russel paradox show that certain combinations of features are clearly inconsistent. These are paradoxes. The unsolvability of the Halting problem and the incompleteness of Peano arithmetic show that certain problems are just too complex to be solved by certain limited means, and these express unfeasibility rather than a paradox. Admittedly, this discrepancy is gradual; provability logics in arithmetic bridge this kind of gap, and in [4], it was analysed how Turing incompleteness can take the form of a paradox-like manifest incompatibility when working in a setting of program algebra and thread algebra.

The famous observation by Cohen dating back to 1984 [18] (see also [19,20]) that an agent cannot decide whether or not its behaviour will be harmful, brings to the surface a disparity of the first kind: a certain combination of features, including a form of forecasting is paradoxical, i.e., leads to a manifest contradiction, while at closer inspection, a somewhat different casting of the same theme brings it closer to the halting problem and Gödel incompleteness. What is tried cannot be achieved in full generality because it is too difficult, not because it is a paradoxical (self-contradictory) idea as such.

In this article, we elaborate on these matters with the following general question in mind.

Problem 2. *Where are the boundaries and what are the options concerning self-reflection about the future of a computation, the self-reflection being understood as a part of the same computation?*

In light of the objectives of this special issue, it matters contemplating how and why this particular problem might matter for AI.

1. Contemplating the Cohen impossibility result, several follow-up questions arise. To begin with, one may ask how the software engineering life-cycle for certain safety critical components would change (would have changed) if the result would have been the opposite to what it has been. How would the tests (which, after all, do not exist) would be used in practice.
   The software engineering life-cycle is a computational model where computed phases and human actions take place in an interleaved manner. However, as a result of progress in AI, increasingly, many of the human steps are becoming incorporated in the computed phases. The question arises if the availability of the program/machine that Cohen showed not to exist would, hypothetically, show up in systems design for automated programming. Apparently, an attempt to incorporate a component which is blessed with successful foresight about whether or not another component is viral (assuming a particular definition of that) into a larger system induces a change of the requirements on said component. As a consequence of that change, instead of being paradoxical, it becomes merely unfeasible (as in the case of the halting problem) to determine virality.
   Unfeasible problems may have useful automated approximations, however, which then could be used for imperfect but still practical automated improvements of an ordinary software engineering life-cycle. By working with approximations of software components that provably do not exist, hypothetical foresight may be transformed into plausible forecasting.

2. The Cohen impossibility may be considered, like the halting problem, to constitute merely a conceptual matter. However, one may consider the notion of a program fault. The formidable literature on program faults provides some principled approaches, e.g., in [21,22], where the idea is that a fault must have two properties, it can be the cause of a failure during a run and it allows a (preferably provable, otherwise at least evidence based) improvement such that said cause is removed. For a survey of approaches to the notion of a program fault, we refer to [23]. Faults may be understood as options for forecasting certain failures, and in a fully automated software engineering life-cycle, one can imagine tools for dealing with, i.e. profiting from, such forms of foresight.
3. Another link between AI and forecasting arises when contemplating robot morality. Without assuming any form of lookahead, it is hardly possible to imagine any moral judgement regarding the control software for a military robot. For some reflection on this matter, we refer to [24].
4. Another area where lookahead plays a role is garbage collection, where, given a pool of linked objects which is used for the processing of a single instruction sequence only, the notion of garbage may be made dependant on the future behaviour of said instruction sequence. This idea gives rise to the notion of shedding (see [25]), which involves run-time foresight by a thread about its own future behaviour. Comparable to the case of virus detection, it takes care to avoid self-contradictory requirements, and indeed AI methods for automated self-reflection might become useful for garbage collection.

1.5. Survey of the Paper

Thread algebra is introduced as a domain-specific process algebra. In particular, thread algebra is ACP representable. The use of process algebra facilitates the introduction of so-called intermediate states, which proves very helpful for introducing the informal notion of a lookahead condition (in Definition 1). Two more informal notions, prospecting service and foresight pattern, are introduced. The relevance of non-paradoxical foresight patterns is motivated.

A fairly complete account is presented of interfaces in the context of thread algebra. That is done both in intuitive and semantic terms and by way of an axiomatisation which is informative, though lacking known formal properties.

The new results are as follows: Proposition 3 demonstrates a case where foresight cannot be realised, while this is not entirely obvious. Proposition 5 introduces a new foresight mechanism, which indicates that the Cohen impossibility result is shown in a context where the sought forecasting pattern is not paradoxical. Theorem 2 indicates that for the same requirements, paradoxical circumstances arise in the presence of a second service with prospecting capability. The basic action foresight pattern is introduced, and in Proposition 4, a prospecting service is obtained which can implement this pattern.

2. Threads, Services, and Architerms

At risk of repeating some elements that were covered in the introduction, we will proceed with some additional remarks on threads, services and service families. We will use without further reference the elements of program algebra, thread algebra, and instruction sequence theory which are surveyed in [3,26], and which have been set out in detail in the papers referenced therein. In these papers, the focus method notation `f.m` provides a basic instruction, where the focus `f` represents the name of (a link to) a service and `m` names a method to be applied to (as seen from an instruction sequence) and processed by (as seen from the service) the service, under the assumption that (i) the service will send a Boolean reply back to the instruction sequence which made the method call `f.m` in such a manner that the continuation of the computation as prescribed by the instruction sequence may (but need not) depend on the reply value, (ii) as a consequence of processing the method `m`, the service may update its state, and (iii) in between the processing of subsequent method calls, say `f.m1` and `f.m2`, the service may interact with external components working as

an independent process, concurrently with other services. The architerm, also termed configuration, which will mostly be used below, is thus:

$$\partial_{\mathtt{U}}(\mathtt{P} \bullet \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n)$$

with $\mathtt{P}$ a thread and $\mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n$, a service family which offers access to service $\mathtt{K}_i$ under focus $\mathtt{g}_i$ for $1 \leq i \leq n$, and with $\mathtt{U} \subseteq \{\mathtt{g}_1, \ldots, \mathtt{g}_n\}$. More often than not, the encapsulation operator will be omitted from an architerm.

The service combination operator $\_ \oplus \_$ is assumed to be commutative and associative so that the ordering of the services in an expression for an execution architecture does not matter. $\partial_{\mathtt{U}}(-)$ performs restriction to the services with a focus outside $\mathtt{U}$. The representation of restriction in ACP-style process algebra satisfies $\natural(\partial_{\mathtt{U}}(\mathtt{P} \bullet \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n)) = \partial_{\mathtt{RS(U)}}(\natural(\mathtt{P} \bullet \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n))$ with $\mathtt{RS}(\mathtt{U})$ containing the communication actions along port $\mathtt{f}$, that is

$$\mathtt{RS}(\mathtt{U}) = \{\mathtt{c}_\mathtt{f}(\mathtt{m}), \mathtt{c}_\mathtt{f}(\mathtt{b}) \mid \mathtt{f} \in \{\mathtt{g}_1, \ldots, \mathtt{g}_n\}, \mathtt{m} \in \mathtt{I}_{\mathsf{method}}(\mathtt{H}_i), \mathtt{b} \in \{\mathtt{true}, \mathtt{false}\}\}$$

Here $\mathtt{c}_\mathtt{f}(\mathtt{m}) = \mathtt{s}_\mathtt{f}(\mathtt{m})|\mathtt{r}_\mathtt{f}(\mathtt{m})$ and $\mathtt{c}_\mathtt{f}(\mathtt{b}) = \mathtt{s}_\mathtt{f}(\mathtt{b})|\mathtt{r}_\mathtt{f}(\mathtt{b})$ with $-|-$, the communication function of ACP. Services with a focus in $\mathtt{U}$ may be used as containers for inputs or as containers for auxiliary data only of use during a computation. All outputs are located as states in services with a focus outside $\mathtt{U}$. For this particular notion of an execution architecture for instruction sequences and for threads, we refer to [12] and to the further development thereof in [4]. Below, we will use architerm as an abbreviation of execution architecture.

The equality of architerms is denoted with $\equiv$. Architerm equality is a somewhat informal motion because it depends on the context as to how much abstraction comes with $\equiv$. Writing $\mathtt{A} \equiv \partial_{\mathtt{U}}(\mathtt{P} \bullet \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n)$, it is confirmed that $\mathtt{A}$ denotes an architerm involving restriction (removal of auxiliary services from the result by $\partial_{\mathtt{U}}(-)$), a thread $\mathtt{P}$, which is applied to (processing determined by $\bullet$) $\mathtt{n}$ services which are accessible via foci $\mathtt{g}_1, \ldots, \mathtt{g}_n$, respectively. No further structural information is given (via $\mathtt{A}$) regarding $\mathtt{P}$ and the respective services. We will always assume that the $\mathtt{g}_i$ are pairwise different. With $\mathtt{U} = \{\mathtt{g}_{j_1}, \ldots, \mathtt{g}_{j_k}\}$, the service family $\mathtt{g}_{j_1}.\mathtt{K}_{j_1} \oplus \cdots \oplus \mathtt{g}_{j_1}.\mathtt{K}_{j_k}$ contains the services with either a pure input status or an auxiliary status (or a combination of both) for the architerm $\mathtt{A}$. With $\mathtt{B} \equiv |+\mathtt{f.m1};\#3;-\mathtt{g.m2};\mathtt{f.m3};\mathtt{X};!| \bullet \mathtt{f.K} \oplus \mathtt{g.L}$, it is required that a thread is applied which is obtained via thread extraction from an instruction sequence beginning with $+\mathtt{f.m1};\#3;-\mathtt{g.m2};\mathtt{f.m3}$ and ending with !. The idea is then that said instruction sequence $(+\mathtt{f.m1};\#3;-\mathtt{g.m2};\mathtt{f.m3})$ constitutes a part of the architerm. By substitution for variables, the architerms can be refined.

2.1. Reply Function Services

Each service $\mathtt{H}$ comes with (i) an input alphabet $\mathtt{J} = \mathtt{I}_{\mathsf{method}}(\mathtt{H})$ and (ii) a reply function $\alpha\colon \mathtt{J}^+ \to \{\mathtt{true}, \mathtt{false}\}$. When a method call $\mathtt{m} \in \mathtt{J}$ takes place, the reply is $\alpha(\mathtt{m})$ and the new reply function is $\lambda\mathtt{u} \in \mathtt{J}^+\alpha(\mathtt{m}\,\mathtt{u})$. A reply-only service has implicit states, and the reply function serves as its state.

We write $\mathtt{RFS}(\mathtt{J},\alpha)$ for the service with method interface $\mathtt{I}_{\mathsf{method}}(\mathtt{RFS}(\mathtt{J},\alpha)) = \mathtt{J}$ and reply function $\alpha\colon \mathtt{J}^+ \to \{\mathtt{true}, \mathtt{false}\}$. When available under focus $\mathtt{f}$, the (single-service) service family $\mathtt{f.RFS}(\mathtt{J},\alpha)$ has provided interface $\mathtt{I}_{\mathtt{provided}}(\mathtt{f.RFS}(\mathtt{J},\alpha)) = \{\mathtt{f.m} \mid \mathtt{m} \in \mathtt{J}\}$. $\mathtt{f.RFS}(\mathtt{J},\alpha)$ will process method call $\mathtt{f.m}$ by asking $\mathtt{RFS}(\alpha, \mathtt{J})$ to process the call $\mathtt{m}$.

2.2. Stateful Services

A service may also have an explicit state space. Then, $\mathtt{H}$ comes with a state space $\mathtt{S}$ containing an initial state $\mathtt{s}_0$ and a reply condition function $\theta\colon \mathtt{S} \times \mathtt{J} \to \{\mathtt{true}, \mathtt{false}\}$ and an effect function $\eta\colon S \times J \times \{true, false\} \to S$. Upon method call $\mathtt{m} \in \mathtt{J}$ and when in state $\mathtt{s}$, the condition $\theta(\mathtt{s},\mathtt{m})$ is evaluated. Let $\mathtt{b} \in \{\mathtt{true}, \mathtt{false}\}$ be the result of evaluating $\theta(\mathtt{s},\mathtt{m})$, then $\mathtt{b}$ is returned to the calling thread as a reply, and the next state is $\eta(\mathtt{s},\mathtt{m},\mathtt{b})$.

Such services are called stateful. We write $\mathtt{SFS}(\mathtt{J},\mathtt{S},\mathtt{s}_0,\theta,\eta)$ (for stateful service) for the service with method interface $\mathtt{J}$, state space $\mathtt{S}$, with initial state $\mathtt{s}_0 \in \mathtt{S}$ reply condition function θ, and effect function η.

The architectural primitive $-\bullet-$ indicates that thread $\mathtt{P}$ is run in the context of the service family $\mathtt{g}_1.\mathtt{K}_1(\mathtt{s}_1) \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}(\mathtt{s}_\mathtt{n})$, with $\mathtt{s}_\mathtt{i}$, assuming encapsulation $\partial_\mathtt{U}$ with $\mathtt{U} = \{\mathtt{g}_{\mathtt{i}_1},\ldots,\mathtt{g}_{\mathtt{i}_k}\}$, a state of $\mathtt{K}_\mathtt{i}$, with as an output, in case of termination, a service family $\mathtt{g}_{\mathtt{i}_1}.\mathtt{K}_{\mathtt{i}_1}(\mathtt{s}'_{\mathtt{i}_1}) \oplus \cdots \oplus \mathtt{g}_{\mathtt{i}_k}.\mathtt{K}_\mathtt{m}(\mathtt{s}'_{\mathtt{i}_k})$. In case of non-termination as well as in the case of divergence after a number of method calls have been processed, the result is $\mathtt{g}_{\mathtt{i}_1}.0_{\mathsf{service}} \oplus \cdots \oplus \mathtt{g}_{\mathtt{i}_k}.0_{\mathsf{service}}$, where 0_{service} is the unique service which accepts no methods.

2.3. Threads and Thread Extraction

A thread $\mathtt{P}$ is either $\mathtt{S}$ (stop, termination), or $\mathtt{D}$ or $\mathtt{P}_1 \trianglelefteq \mathtt{g.m} \trianglerighteq \mathtt{P}_2$ for some method call (basic action) $\mathtt{g.m}$ and some threads $\mathtt{P}_1$ and $\mathtt{P}_2$. $\mathtt{P} \trianglelefteq \mathtt{f.m} \trianglerighteq \mathtt{P}$ can be abbreviated to $\mathtt{f.m} \circ \mathtt{P}$.

Instruction sequences and threads are connected via the thread extraction operator $|\mathtt{P}|$, for instance: $|\mathtt{f.m};\mathtt{g.m1};\mathtt{h.m2};!| = \mathtt{f.m} \circ \mathtt{g.m1} \circ \mathtt{h.m2} \circ \mathtt{S}$, $|+\mathtt{f.m};\#2;\mathtt{g.m1};\mathtt{h.m2};!| = (\mathtt{h.m2} \circ \mathtt{S}) \trianglelefteq \mathtt{f.m} \trianglerighteq (\mathtt{g.m1} \circ \mathtt{h.m2} \circ \mathtt{S})$, and $|\mathtt{f.m};+\mathtt{g.m1};!;-\mathtt{h.m2};\#0;!| = \mathtt{f.m} \circ (\mathtt{S} \trianglelefteq \mathtt{g.m1} \trianglerighteq (\mathtt{S} \trianglelefteq \mathtt{h.m2} \trianglerighteq \mathtt{D}))$, where $\mathtt{g.m} \circ \mathtt{P} = \mathtt{P} \trianglelefteq \mathtt{g.m} \trianglerighteq \mathtt{P}$.

For the construction of an apply architerm, it is required that the interfaces of the thread and service family match, i.e., that every method call can be processed by one of the services. Expressing these requirements rests on a notion of interface, which we will discuss in some detail.

2.4. Interfaces: Required Interfaces and Provided Interfaces

Interfaces serve as abstractions of components which are made in order to facilitate reasoning about the design and composition of components. Consider the instruction sequence, $\mathtt{X} \equiv +\mathtt{f.m1};\mathtt{g.m2};!$. The processing of $\mathtt{X}$ works as follows: (i) apply method $\mathtt{m1}$ to the service accessible with focus $\mathtt{f}$; (ii) if the reply $\mathtt{false}$ is returned, skip the second method call and terminate; and (iii) if instead $\mathtt{true}$ is replied, perform the method call $\mathtt{g.m2}$ and then terminate, irrespective of the reply which has been returned. $\mathtt{X}$ requires of an execution environment that it provides a service accessible under focus $\mathtt{f}$ which is able to process a method named $\mathtt{m1}$ and another service accessible under focus $\mathtt{g}$ which will process a method named $\mathtt{m2}$. This information can be combined in the interface $\mathtt{J} = \{\mathtt{f.m1},\mathtt{g.m2}\}$. The interface $\mathtt{J}$ is a required interface, it conveys what an environment must provide for $\mathtt{X}$ to be properly effectuated. Now consider $\mathtt{Y} = !;+\mathtt{f.m1};\mathtt{g.m2};!$. $\mathtt{Y}$ will terminate at once and make no method calls. It is plausible to say that its required interface is empty. If, however, a jump from outside is allowed and a run may not start with the first instruction (e.g., in $\mathtt{Z} \equiv \#3;\mathtt{Y}$), then it is not plausible to think of $\mathtt{Y}$ as having an empty interface. With $\mathtt{I}_{\mathsf{required}}(\mathtt{P})$, we will denote the required interface of an instruction sequence $\mathtt{P}$, which contains a mere collection of all basic actions ($\mathtt{f.m}$), which occur in $\mathtt{X}$ (either as a void basic instruction $\mathtt{f.m}$, as a positive test instruction $+\mathtt{f.m}$ or as a negative test instruction $-\mathtt{f.m}$). Thus, with $\mathtt{Y}$ as above, $\mathtt{I}_{\mathsf{required}}(\mathtt{X}) = \mathtt{I}_{\mathsf{required}}(\mathtt{Y}) = \{\mathtt{f.m1},\mathtt{g.m2}\}$.

Given the task to design an instruction sequence, say $\mathtt{X}$, which computes some given function or otherwise performs some specified task, there may be constraints (requirements on) on the required interface of $\mathtt{X}$ for instance one may wish that $\mathtt{I}_{\mathsf{required}}(\mathtt{X}) \subseteq \mathtt{J}$ for some interface $\mathtt{J}$. For the programming task at hand, $\mathtt{J}$ is an access-constraining interface. Access-constraining interfaces play a key role in complexity questions about instruction sequences when looking for the shortest, or otherwise best, instruction sequence in some format (say PGLB of [2]) that achieves a certain given task.

Complementarily to the required interface of an instruction sequence, one may consider the method interface $\mathtt{I}_{\mathsf{method}}(\mathtt{H})$ of a service, say $\mathtt{H}$, e.g., $\{\mathtt{m1},\mathtt{m2},\mathtt{m3}\}$. A method interface of a service is a provided interface. If a service family $\mathtt{H}$ offers only $\mathtt{S}$ accessible under focus $\mathtt{f}$, then the provided interface of $\mathtt{H}$ is $\mathtt{I}_{\mathtt{provided}}(\mathtt{H}) = \{\mathtt{f.m1},\mathtt{f.m2},\mathtt{f.m3}\}$. If one has the task to design a service family, say $\mathtt{L}$, then it may be required that a certain interface $\mathtt{J}$ is provided by $\mathtt{L}$: $\mathtt{I}_{\mathtt{provided}}(\mathtt{L}) \supseteq \mathtt{J}$.

2.5. Some Naming Conventions; Equations for Interfaces

The paper involves a range of different types of entities, and we will follow some informal, and non-strict rules for the naming of variables for these types. We will use $\mathtt{X}, \mathtt{Y}, \mathtt{Z}$ for instruction sequences, $\mathtt{P}, \mathtt{Q}, \mathtt{R}$ for threads, $\mathtt{I}, \mathtt{J}$ for interfaces, $\mathtt{H}, \mathtt{K}, \mathtt{L}$ for services, $\mathtt{S}, \mathtt{T}$ for state spaces of (stateful) services with $\mathtt{s}, \mathtt{t}$ for states in the state space of stateful services, and $\mathtt{V}, \mathtt{W}$ for service families. Methods often are given names involving $\mathtt{m}$ or $\mathtt{n}$, and foci often have names involving $\mathtt{f}$, $\mathtt{g}$, or $\mathtt{h}$.

Using these conventions, some equations that loosely specify the various types of and operations about interfaces are given in Tables 1 and 2. Concerning these tables, some comments are in order.

- Interfaces are by default sets of focus method pairs. Interfaces may serve as the required interface for a thread, as the provided interface for a service family, or as the constraining interface for a (forthcoming) thread. $\emptyset_{\mathsf{interface}}$ is the empty interface. $0_{\mathsf{service-family}}$ is the empty service family. Its interface is empty as well.
- Services have a method interface which consists of method names only. $\emptyset_{\mathsf{method-interface}}$ is the empty method interface. 0_{service} is the unique service without any method.
- The service family $\mathtt{f}.0_{\mathsf{service}}$ differs from $\emptyset_{\mathsf{interface}}$ because it involves a focus $\mathtt{f}$. This complication is useful because it allows $\oplus$ to be a total, commutative, and associative operator. Suppose $\mathtt{m} \in \mathtt{I}_{\mathtt{methods}}(\mathtt{H}_1) \cap \mathtt{I}_{\mathtt{methods}}(\mathtt{H}_2)$ the result of a call $\mathtt{f}.\mathtt{m}$ to a service family $\mathtt{f}.\mathtt{H}_1 \oplus \mathtt{f}.\mathtt{H}_2$ is hard to define. Non-determinism is to be avoided, and $\oplus$ is supposedly commutative. A simple way out is to avoid service families with multiple services under the same focus. Adopting $\mathtt{f}.\mathtt{H} \oplus \mathtt{f}.\mathtt{K} = 0_{\mathsf{service-family}}$ fails, as it renders $\oplus$ as being non-associative. However, adopting $\mathtt{f}.\mathtt{H} \oplus \mathtt{f}.\mathtt{K} = \mathtt{f}.0_{\mathsf{service}}$, blocks ambiguous method calls and allows $\oplus$ to be commutative and associative, though not idempotent (i.e. $\mathtt{V} \oplus \mathtt{V} = \mathtt{V}$ fails on any service family $\mathtt{V}$ with $\mathtt{I}_{\mathtt{provided}}(\mathtt{V}) \neq \emptyset_{\mathsf{interface}}$).
- The notion of an interface occurs in quite different ways in the literature on software technology. Three clusters of use of the term "interface" may be distinguished as follows:
  - Operational interfaces: collections of primitives with definite meaning.
    - API: an application programmer interface is a toolkit in a software engineering environment;
    - Processor instruction sets;
    - Collections of message passing primitives (e.g., MPI).
  - Requirements of specification-oriented interfaces (interface elements come with limited information regarding semantics).
    - Modal interfaces (see, for example, [27]);
    - Interface automata (see, for example, [28]);
    - Algebraic specifications as interface of data types.
  - Syntactic interfaces. (comprising declarations of syntactic elements; interface elements come without a predefined semantics, but may have suggestive names or symbols).
    - Similarity types in universal algebra (interfaces for first order theories);
    - Signatures as interfaces for algebraic (abstract data type) specifications;
    - Sets (alphabets) of atomic actions as process interfaces in process algebra (see, for example, [17]);
    - Sets of basic actions as interfaces in program algebra and thread algebra (following, for example, [26].

  In the context of thread algebra, interfaces are used in the capacity of syntactic interfaces as well as in the capacity of operational interfaces (though in a theoretical manner).

Table 1. Equations for provided interfaces.

$$\mathtt{x} \cup \emptyset_{\mathsf{interface}} = \mathtt{x} \tag{1}$$
$$\mathtt{x} \cup \mathtt{y} = \mathtt{y} \cup \mathtt{x} \tag{2}$$
$$\mathtt{x} \cup (\mathtt{y} \cup \mathtt{z}) = (\mathtt{x} \cup \mathtt{y}) \cup \mathtt{z} \tag{3}$$
$$\mathtt{x} \cup \mathtt{x} = \mathtt{x} \tag{4}$$
$$\mathtt{f}.\{\mathtt{m}\} = \{\mathtt{f}.\mathtt{m}\} \tag{5}$$
$$\mathtt{f}.\emptyset_{\mathsf{method-interface}} = \emptyset_{\mathsf{interface}} \tag{6}$$
$$\mathtt{f}.(\mathtt{h} \cup \mathtt{u}) = \mathtt{f}.\mathtt{h} \cup \mathtt{f}.\mathtt{u} \tag{7}$$
$$\mathtt{H} \oplus 0_{\mathsf{service-family}} = \mathtt{H} \tag{8}$$
$$\mathtt{H} \oplus \mathtt{L} = \mathtt{L} \oplus \mathtt{H} \tag{9}$$
$$\mathtt{H} \oplus (\mathtt{K} \oplus \mathtt{L}) = (\mathtt{H} \oplus \mathtt{K}) \oplus \mathtt{L} \tag{10}$$
$$\partial_{\emptyset_{\mathsf{focus-collection}}}(\mathtt{H}) = \mathtt{H} \tag{11}$$
$$\partial_{\mathtt{U}}(0_{\mathsf{service-family}}) = 0_{\mathsf{service-family}} \tag{12}$$
$$\partial_{\{\mathtt{f}\}}(\mathtt{f}.\mathtt{R}) = 0_{\mathsf{service-family}} \tag{13}$$
$$\mathtt{f} \neq \mathtt{g} \to \partial_{\{\mathtt{f}\}}(\mathtt{g}.\mathtt{R}) = \mathtt{g}.\mathtt{R} \tag{14}$$
$$\partial_{\{\mathtt{f}\}}(\mathtt{H} \oplus \mathtt{K}) = \partial_{\{\mathtt{f}\}}(\mathtt{H}) \oplus \partial_{\{\mathtt{f}\}}(\mathtt{K}) \tag{15}$$
$$\partial_{\mathtt{U} \cup \mathtt{V}}(\mathtt{H}) = \partial_{\mathtt{U}} \circ \partial_{\mathtt{V}}(\mathtt{H}) \tag{16}$$
$$\mathtt{f}.\mathtt{H} \oplus \mathtt{f}.\mathtt{K} = \mathtt{f}.0_{\mathsf{service}} \tag{17}$$
$$\mathtt{I}_{\mathsf{method}}(0_{\mathsf{service}}) = \emptyset_{\mathsf{method-interface}} \tag{18}$$
$$\mathtt{I}_{\mathsf{method}}(\mathtt{H}) = \{\mathtt{m} \mid \mathtt{m}\ \mathtt{is\ a\ method\ of}\ \mathtt{H}\} \tag{19}$$
$$\mathtt{I}_{\mathtt{provided}}(0_{\mathsf{service-family}}) = \emptyset_{\mathsf{interface}} \tag{20}$$
$$\mathtt{I}_{\mathtt{provided}}(\mathtt{f}.\mathtt{H}) = \mathtt{f}.\mathtt{I}_{\mathsf{method}}(\mathtt{H}) \tag{21}$$
$$\mathtt{I}_{\mathtt{provided}}(\mathtt{f}.\mathtt{H} \oplus \partial_{\{\mathtt{f}\}}(\mathtt{K})) = \mathtt{I}_{\mathtt{provided}}(\mathtt{f}.\mathtt{H}) \cup \mathtt{I}_{\mathtt{provided}}(\partial_{\{\mathtt{f}\}}(\mathtt{K})) \tag{22}$$
$$\mathtt{I}_{\mathsf{focus-collection}}(0_{\mathsf{service-family}}) = \emptyset_{\mathsf{focus-collection}} \tag{23}$$
$$\mathtt{I}_{\mathsf{focus-collection}}(\mathtt{f}.\mathtt{H}) = \{\mathtt{f}\} \tag{24}$$
$$\mathtt{I}_{\mathsf{focus-collection}}(\mathtt{V} \oplus \mathtt{W}) = \mathtt{I}_{\mathsf{focus-collection}}(\mathtt{V}) \cup \mathtt{I}_{\mathsf{focus-collection}}(\mathtt{W}) \tag{25}$$

$$\mathtt{x} \subseteq \mathtt{y} \iff \mathtt{x} \cup \mathtt{y} = y \tag{26}$$

Table 2. Equations for the required interfaces of threads and instruction sequences.

$$\mathtt{I}_{\mathsf{required}}(\mathtt{S}) = \emptyset_{\mathsf{interface}} \tag{27}$$
$$\mathtt{I}_{\mathsf{required}}(\mathtt{D}) = \emptyset_{\mathsf{interface}} \tag{28}$$
$$\mathtt{I}_{\mathsf{required}}(\mathtt{P} \trianglelefteq \mathtt{f}.\mathtt{m} \trianglerighteq \mathtt{Q}) = \mathtt{I}_{\mathsf{required}}(\mathtt{P}) \cup \{\mathtt{f}.\mathtt{m}\} \cup \mathtt{I}_{\mathsf{required}}(\mathtt{Q}) \tag{29}$$

$$\mathtt{I}_{\mathsf{required}}(!) = \emptyset_{\mathsf{interface}} \tag{30}$$
$$\mathtt{I}_{\mathsf{required}}(\#\mathtt{n}) = \mathtt{I}_{\mathsf{required}}(\backslash\#\mathtt{n}) = \emptyset_{\mathsf{interface}} \tag{31}$$
$$\mathtt{I}_{\mathsf{required}}(+\mathtt{f}.\mathtt{m}) = \mathtt{I}_{\mathsf{required}}(-\mathtt{f}.\mathtt{m}) = \mathtt{I}_{\mathsf{required}}(\mathtt{f}.\mathtt{m}) = \{\mathtt{f}.\mathtt{m}\} \tag{32}$$
$$\mathtt{I}_{\mathsf{required}}(\mathtt{X};\mathtt{Y}) = \mathtt{I}_{\mathsf{required}}(\mathtt{X}) \cup \mathtt{I}_{\mathsf{required}}(\mathtt{Y}) \tag{33}$$
$$\mathtt{I}_{\mathsf{required}}(\mathtt{X}^!) = \mathtt{I}_{\mathsf{required}}(\mathtt{X}) \tag{34}$$

2.6. Thread Service Interaction, Instruction Sequence Processing Operators

In PGA-style program algebra and the corresponding theory of instruction sequences, the general idea of an execution architecture is as follows: given a instruction sequence $\mathtt{X}$, thread extraction produces a thread $|\mathtt{X}|$. Threads are processes captured in a thread algebra which is a simplified process algebra. Threads have three forms: $\mathtt{S}$, the terminating (stopping) thread, $\mathtt{D}$ the deadlocked (diverging, improperly terminating) thread and $\mathtt{P} \equiv \mathtt{Q} \trianglelefteq \mathtt{f.m} \trianglerighteq \mathtt{R}$, the basic action form of a thread. The stopped thread may be equipped with a result value, for instance, a bit: $\mathtt{S}_0$ and $\mathtt{S}_1$. Threads in basic action form can perform a unique basic action, say, for $\mathtt{P}$: $\mathtt{f.m}$, which works as follows:

(step 1) Let the service in focus $\mathtt{f}$ (say the service name $\mathtt{H}$, which at the time of the method call is in state $\mathtt{s}$, or more precisely $\mathtt{H}(\mathtt{s})$) performs the (its) method named $\mathtt{m}$, the service is offered from the context as given by the execution architecture.

(step 2) A Boolean reply is expected, at reply $\mathtt{true}$, the run proceeds with $\mathtt{Q}$, and at reply $\mathtt{false}$, the run proceeds with $\mathtt{R}$, and the reply is found by evaluating the reply condition $\theta(\mathtt{s},\mathtt{m})$ for method $\mathtt{m}$ to state $\mathtt{s}$, thereby obtaining Boolean value $\mathtt{b}$.

(step 3) In both cases the state $\mathtt{s}$ of $\mathtt{H}$ is updated with the effect function η to $\mathtt{H}(\eta(\mathtt{s},\mathtt{m},\mathtt{b}))$. Three operators connecting instruction sequences and service families come into play:

(i) The reply operator: $|\mathtt{X}|\,!\,\mathtt{H}$, which produces a bit in $\{\mathtt{true},\mathtt{false}\}$ and in which $\mathtt{X}$ is understood as a way to define (compute) a function from (states of) service families to bits. To allow definition of the reply operator, two different termination instructions are needed: $!{:}\mathtt{true}$ (terminate with result $\mathtt{true}$) and $!{:}\mathtt{false}$ (terminate with result $\mathtt{false}$).

(ii) The apply operator $|\mathtt{X}| \bullet \mathtt{H}$ in which $\mathtt{X}$ is supposed to describe a transformation from $\mathtt{H}$ to a new state of it (i.e., to a new state for each of the services contained in it).

(iii) The use operator $|\mathtt{X}|/\mathtt{H}$, which produces a thread rather than a value or a service family (carrying the states of the various services). For $|\mathtt{X}|/\mathtt{H}$, the services combined in $\mathtt{H}$ act as local resources, to begin with memory sources, which support $\mathtt{X}$ in computing thread $|\mathtt{X}|/\mathtt{H}$.

Typically, one may imagine compositions $(|\mathtt{X}|/\mathtt{V}) \bullet \mathtt{W}$ where it is plausible though not necessary that the service families $\mathtt{V}$ and $\mathtt{W}$ have no focus in common (i.e., $\mathtt{I}_{\mathsf{focus-collection}}(\mathtt{V}) \cap \mathtt{I}_{\mathsf{focus-collection}}(\mathtt{W}) = \emptyset_{\mathsf{focus-collection}}$). Encapsulation drops services from a service family: with $\mathtt{U}$, a collection of foci, $\partial_{\mathtt{U}}(\mathtt{V})$ removes from $\mathtt{V}$ each service accessible under a focus in $\mathtt{U}$. Assuming that $\mathtt{V}$ and $\mathtt{W}$ involve disjoint collections of foci results in the following useful identity:

$$\mathtt{I}_{\mathsf{focus-collection}}(\mathtt{V}) \cap \mathtt{I}_{\mathsf{focus-collection}}(\mathtt{W}) = \emptyset_{\mathsf{focus-collection}} \rightarrow$$

$$(|\mathtt{X}|/\mathtt{V}) \bullet \mathtt{W} = \partial_{\mathtt{I}_{\mathsf{focus-collection}}(\mathtt{V})}(|\mathtt{X}| \bullet (\mathtt{V} \oplus \mathtt{W})))$$

3. Prospecting Services: Endowed with a Non-Ordinary Capability

Below, further lookahead conditions are introduced, as a follow up on the ones mentioned in Section 1.1. Prospecting refers to the capability of a service to evaluate one or more lookahead conditions. That is a non-ordinary capability. Foresight is a property of a system (here, an execution architecture) where one or more services perform as if they were able to forecast the actual and potential future behaviour of the system. Proper foresight is present if no ordinary service meets the given requirements.

The basic action foresight pattern, when realised, allows a system to use a certain service (here, $\mathtt{H}$) to operate in such a manner that a given basic action (viz. $\mathtt{g_i.m}$) will certainly be performed provided that basic action might occur with non-deterministic behaviour for $\mathtt{H}$. For an explanation of the conditions of the form $\mathtt{A}\ \underline{\mathtt{sat}}\ \mathtt{CALL(h.m)}$, see Section 1.2 above.

Definition 2 (Basic action foresight pattern). *Let* $\mathtt{n} > 1$ *and let* $\mathtt{J}_1,\ldots,\mathtt{J}_\mathtt{n}$ *be method interfaces. Further, let* $\mathtt{H}$ *be a service with* $\mathtt{I}_{\mathsf{method}}(\mathtt{H}) = \{\mathtt{find}\}$ *then an open architerm* $(\mathtt{X}_1 \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{X}_2) \bullet \mathtt{h.H} \oplus \mathtt{g}_1.\mathtt{Y}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{n}.\mathtt{Y}_\mathtt{n}$ *realises the* basic action foresight pattern *for*

some $\mathtt{i} \in \{1,\ldots,n\}$ *and method* $\mathtt{m} \in \mathtt{J_i}$ *if for all ordinary (but arbitrary) services* $\mathtt{K_1},\ldots,\mathtt{K_n}$ *with* $\mathtt{I_{method}(K_1)} \subseteq \mathtt{J_1},\ldots,\mathtt{I_{method}(K_n)} \subseteq \mathtt{J_n}$, *and for all threads* $\mathtt{P},\mathtt{Q}$ *with* $\mathtt{I_{required}(P)} \cup \mathtt{I_{required}(Q)} \subseteq \mathtt{I_{provided}(h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n})$, *the following holds:*

if $\mathtt{P} \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ $\underline{\mathtt{sat}}$ $\mathtt{CALL(g_i.m)}$ *or* $\mathtt{Q} \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ $\underline{\mathtt{sat}}$ $\mathtt{CALL(g_i.m)}$ *then also,* $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ $\underline{\mathtt{sat}}$ $\mathtt{CALL(g_i.m)}$.

The example in Section 1.1 above (taking $\mathtt{i} = 2$ and $\mathtt{m} = \mathtt{m2}$) provides a proof of the following proposition.

Proposition 1. *No ordinary service* H *satisfies the requirements of the basic action foresight pattern.*

In Theorem 4 below, a prospecting service named MCP is shown to satisfy the requirements of the basic action foresight pattern. From the latter observation, it follows that the basic action foresight pattern is not self-contradictory.

3.1. Forecasting versus Prospection: Forecasting Patterns and Prospecting Services

Proposition 1 asserts that a certain foresight pattern will not emerge unless a service with prospecting capability is available. A foresight pattern may also be inconsistent (paradoxical, self-contradictory).

Foresight is reflexive in the sense that it works as if an agent (service) is well-informed about its own future behaviour, as well as about the behaviour of other agents (services). Prospection is non-reflexive, and refers to the capability of a service to acquire knowledge about the future behaviour of threads and other services in the same system. Lookahead conditions which occur in a service will work under the hypothesis that the service proceeds as an ordinary service (even if, in fact, it does not). For this limitation, see the preliminary definition of lookahead conditions and the remarks in Section 1.2 above. Using these conventions, prospection realised in a service by way of lookahead conditions will not have any paradoxical consequences, however hypothetical the evaluation of lookahead conditions in practice may be.

Proposition 2. *Let* $\mathtt{i} \in \{1,\ldots,n\}$, *and let* $\mathtt{K_1},\ldots,\mathtt{K_n}$ *be ordinary (i.e., non-prospecting) services and let the interface* J *be given by* $\mathtt{J} = \{\mathtt{f.ok}\} \cup \mathtt{I_{provided}(g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n})$. *Further let* TERM *be a lookahead service with* $\mathtt{I_{method}(TERM)} = \{\mathtt{ok}\}$.

Now, it is not possible that for all threads P *of the form* $\mathtt{P} \equiv \mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q'}$, *with* $\mathtt{I_{required}(P)} \subseteq \mathtt{J}$, *the following holds: the first call along focus* f *of the computation* $\mathtt{P} \bullet \mathtt{f.TERM} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *returns* true *if and only if the computation of* $\mathtt{P} \bullet \mathtt{f.TERM} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *properly terminates.*

Proof. Suppose otherwise, and take $\mathtt{Q} \equiv \mathtt{D}$, $\mathtt{Q'} \equiv \mathtt{S}$. Now the computation $(\mathtt{D} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{S}) \bullet \mathtt{f.TERM} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ terminates properly (ends in S) if and only if the first (and only) method call f.ok returns true if and only if the same computation diverges (ends in D). □

Proposition 2 is a rephrasing of the argument on the non-existence of malware detection by Cohen [18]. The latter result works with an action do:harm instead of termination. We will discuss this forecasting pattern in more detail in Section 3.3 below. It is informative to phrase Proposition 2 in terms of a foresight pattern.

3.2. More Primitive Lookahead Conditions R

Primitive lookahead conditions have the quality that these may be understood as outcomes of experimenting with an architerm as outlined in Definition 1 and subsequent comments. Besides the lookahead condition $\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{CALL(f.m)}$, which is introduced below Definition 1, the following lookahead conditions (for an architerm A) are used below:

- A $\underline{\mathtt{sat}}$ PERP holds if the computation from A does not terminate (i.e., perpetuates).

- $\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{TERMINATION(ok)}$ if the computation starting from $\mathtt{A}$ properly terminates.
- $\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{TERMINATION(not\_ok)}$ asserts that the computation starting from $\mathtt{A}$ improperly terminates (i.e. ends in $\mathtt{D}$).
- $\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{TERMINATIONat(n,ok)}$ asserts that in the $\mathtt{n}$'th step proper termination occurs.
- $\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{TERMINATIONat(n,not\_ok)}$ asserts that at the $\mathtt{n}$'th step improper termination occurs.
- $\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{noCALL(f.m)}$ asserts that the computation from architerm $\mathtt{A}$ at no stage performs a method call $\mathtt{f.m}$. Here, the computation may not terminate or may terminate properly or improperly and if $\mathtt{f} \notin \{\mathtt{g_1},\ldots,\mathtt{g_m}\}$, $\neg(\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{noCALL(f.m)})$ holds.
- $\mathtt{A}$ $\underline{\mathtt{sat}}$ $\mathtt{CALLat(k,f.m)}$ asserts that the computation from architerm $\mathtt{A}$ performs a method call $\mathtt{f.m}$ in step number $\mathtt{k}$ for a natural number $\mathtt{k} > 0$.

Definition 3 (Termination foresight pattern A)**.** *Let* $\mathtt{n} > 1$ *and let* $\mathtt{J_1},\ldots,\mathtt{J_n}$ *be method interfaces. Further, let* $\mathtt{H}$ *be a service with method interface* $\mathtt{I_{method}(H)} = \{\mathtt{ok}\}$ *then an open architerm* $\mathtt{X_P} \bullet \mathtt{f.H} \oplus \mathtt{g_1.Y_1} \oplus \cdots \oplus \mathtt{g_n.Y_n}$ *realises the* termination foresight pattern A *if the following holds:*

let $\mathtt{K_1},\ldots,\mathtt{K_n}$ *be ordinary services such that* $\mathtt{I_{method}(K_1)} \subseteq \mathtt{J_1},\ldots \mathtt{I_{method}(K_n)} \subseteq \mathtt{J_n}$*. Then, for all threads* $\mathtt{Q},\mathtt{Q'}$ *with* $\mathtt{I_{required}(Q)} \cup \mathtt{I_{required}(Q')} \subseteq \mathtt{I_{provided}(f.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n)}$*, it is the case that:* $(\mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q'}) \bullet \mathtt{f.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ $\underline{\mathtt{sat}}$ $\mathtt{TERMINATION(ok)}$*, if and only if the first call of* $\mathtt{f.ok}$ *in the computation* $(\mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q'}) \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *returns* $\mathtt{true}$*.*

Proposition 1 entails that the termination foresight pattern A cannot be realised, not even with a prospecting service. That state of affairs is hardly surprising. Here is a similar foresight pattern for which it is less obvious that it cannot be realised.

Definition 4 (Termination foresight pattern B)**.** *Let* $\mathtt{n} > 1$ *and let* $\mathtt{J_1},\ldots,\mathtt{J_n}$ *be method interfaces. Further let* $\mathtt{H}$ *be an SFS service (see Section 2.2 above) with method interface* $\mathtt{I_{method}(H)} = \{\mathtt{ok}\}$ *and with state space* $\mathtt{S}$ *and initial state* $\mathtt{s_0} \in \mathtt{S}$*, then an open architerm* $\mathtt{X_P} \bullet \mathtt{f.H} \oplus \mathtt{g_1.Y_1} \oplus \cdots \oplus \mathtt{g_n.Y_n}$ *realises the* termination foresight pattern B *if the following holds:*

let $\mathtt{K_1},\ldots,\mathtt{K_n}$ *be ordinary services such that* $\mathtt{I_{method}(K_1)} \subseteq \mathtt{J_1},\ldots \mathtt{I_{method}(K_n)} \subseteq \mathtt{J_n}$ *and assume that* $\mathtt{m} \in \mathtt{I_{method}(K_i)}$*. Then for all threads* $\mathtt{Q},\mathtt{Q'}$ *with* $\mathtt{I_{required}(Q)} \cup \mathtt{I_{required}(Q')} \subseteq \mathtt{I_{provided}(f.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n)}$*, and for each state* $\mathtt{s} \in \mathtt{S}$*, it is the case that* $\mathtt{Q} \bullet \mathtt{f.H(s)} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ $\underline{\mathtt{sat}}$ $\mathtt{TERMINATION(ok)}$ *if and only if the first call of* $\mathtt{f.ok}$ *in the computation* $(\mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q'}) \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *returns* $\mathtt{true}$*.*

Termination foresight pattern B is unrealisable, irrespective of any conceivable prospecting capabilities of $\mathtt{H}$:

Proposition 3. *Termination foresight pattern B cannot be realised by any ordinary or non-ordinary service* $\mathtt{H}$*.*

Proof. Let $\mathtt{S}$ be the state space of $\mathtt{H}$ so that $\mathtt{H} = \mathtt{H(s_0)}$ and choose $\mathtt{Q}$ and $\mathtt{Q'}$ as follows: $\mathtt{Q} = \mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{S}, \mathtt{Q'} = \mathtt{S}$.

If $\theta_{\mathtt{H}}(\mathtt{s_0},\mathtt{ok}) = \mathtt{false}$ then $\mathtt{Q} \bullet \mathtt{f.H(s_0)} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ $\underline{\mathtt{sat}}$ $\mathtt{TERMINATIONat(2,ok)}$ which contradicts the assumptions of the proposition. It follows that $\theta_{\mathtt{H}}(\mathtt{s_0},\mathtt{ok}) = \mathtt{true}$, Now consider $\mathtt{s_1} = \eta_{\mathtt{H}}(\mathtt{s_0},\mathtt{ok},\mathtt{true})$. For $\mathtt{s_1}$ it follows in the same way that $\theta_{\mathtt{H}}(\mathtt{s_1},\mathtt{ok}) = \mathtt{true}$. In this way, an infinite sequence $\mathtt{s_0},\mathtt{s_1},\mathtt{s_2},..$ results such that for all $\mathtt{i}$, $\theta_{\mathtt{H}}(\mathtt{s_i},\mathtt{ok}) = \mathtt{true}$ and $\mathtt{s_{i+1}} = \eta_{\mathtt{H}}(\mathtt{s_i},\mathtt{ok},\mathtt{true})$. It follows that $\mathtt{Q} \bullet \mathtt{f.H(s_0)} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ $\underline{\mathtt{sat}}$ $\mathtt{PERP}$ so that termination does not occur, which contradicts the assumption that the reply to the first method call equals $\mathtt{true}$, i.e., $\theta_{\mathtt{H}}(\mathtt{s_0},\mathtt{ok}) = \mathtt{true}$. □

3.3. SHRD: A Hypothetical Service for Security Hazard Risk Detection

We contemplate a service $\mathtt{SHRD}$ (security hazard risk detection), according to which the occurrence of method call $\mathtt{g_i.do{:}harm}$ constitutes a security hazard, and which supports running a thread in such a way that the risk of a security hazard is detected and an

alternative option for proceeding the computation is taken. The following result states the impossibility result of Cohen in terms of threads and services. This formalisation is based on [29].

Malware detection in practice, and in theory, amounts to much more than the mere detection of the future occurrence of a single basic action. For instance, in [30], the occurrence of runs of certain instruction sequences as subcomputations, perhaps interspersed with other steps, of the run of an instruction sequence is considered an indication/confirmation of a virus infection.

Proposition 4. *Let* $\mathtt{i} \in \{1,\ldots,n\}$ *and let* $\mathtt{K_1},\ldots,\mathtt{K_n}$ *be ordinary services with* $\mathtt{do{:}harm} \in \mathtt{I_{method}(K_i)}$*, and let the interface* $\mathtt{J}$ *be given by* $\mathtt{J} = \{\mathtt{f.ok}\} \cup \mathtt{I_{provided}}(\mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n})$*. Let* SHRD *(for security hazard determination) be a (possibly non-ordinary) service with method interface* $\mathtt{I_{method}(SHRD)} = \{\mathtt{ok}\}$*.*

Now, it is not possible that for all threads $\mathtt{P}$ *of the form* $\mathtt{P} \equiv \mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q'}$ *with* $\mathtt{I_{required}(P)} \subseteq \mathtt{J}$*, the following holds: the first call along focus* $\mathtt{f}$ *of the computation* $\mathtt{P} \bullet \mathtt{f.SHRD} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *returns* `true` *if and only if the computation of* $\mathtt{P} \bullet \mathtt{f.SHRD} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *does not involve a method call* $\mathtt{g_i.do{:}harm}$*.*

Proof. Suppose otherwise, and take $\mathtt{Q} \equiv \mathtt{do{:}harm} \circ \mathtt{S}$, $\mathtt{Q'} \equiv \mathtt{S}$. Then, one finds that the computation $((\mathtt{do{:}harm} \circ \mathtt{S}) \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{S}) \bullet \mathtt{f.SHRD} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ avoids performing $\mathtt{g_i.do{:}harm}$ if and only if the first (and only) method call $\mathtt{f.ok}$ returns `true` if and only if the same computation performs $\mathtt{g_i.do{:}harm}$ (and then terminates). □

Now assume, irrespective of Proposition 4, that an SHRD-like service is available as suggested in the proposition. Plausibly, SHRD would be used in the following manner: in a context $(\mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q'}) \bullet \mathtt{f.SHRD} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_m.K_m}$ call $\mathtt{f.ok}$ in order to prevent a run of $\mathtt{Q}$ leading to a method call $\mathtt{g_i.do{:}harm}$ and instead to engage in a run of $\mathtt{Q'}$, thereby hopefully avoiding $\mathtt{g_i.do{:}harm}$. Although SHRD as indicated above in Proposition 4 cannot exist, a lookahead service which allows the particular use just mentioned is conceivable. In the following paragraph, such a service is defined under the name SHRAT. SHRAT will be a service of the form $\mathtt{L}_\theta^{\mathtt{ok,s}}$ according to the following definition.

Definition 5. *Let the service* $\mathtt{L}_\theta^{\mathtt{m,s}}$ *has singleton state space* $\{\mathtt{s}\}$ *and* $\mathtt{I_{method}}(\mathtt{L}_\theta^{\mathtt{m,s}}) = \{\mathtt{w}\}$ *and when accessible under focus* $\mathtt{f}$ *in an architerm* $\mathtt{A}$ *it works as follows: on method call* $\mathtt{f.m}$ *evaluate* $\mathtt{b} = \theta(\mathtt{s},\mathtt{m})$*, then return* $\mathtt{b}$*, without changing the state.*

3.4. SHRAT: A Prospecting Service for Security Hazard Risk Assessment

In [31], the possibility of a service SHRAT (security hazard risk assessment) service is contemplated, which, supposedly, determines during the operation of an instruction sequence X, the text of which is known to the service, and can determine whether or not the thread created by the instruction sequence will in the future perform a method call do:harm to an unnamed service.

A method call ok to SHRAT aims to determine whether or not "the future is ok", i.e., no do:harm action will be performed. We will rephrase this idea with more emphasis on interfaces than in the presentation in [31]. To begin with the unnamed service is given name K and is present via the interface under focus g.

We imagine a service SHRAT with a single method ok and a single state s. In slight contrast with SHRAT one may imagine a service $\mathtt{K_i}$ with method interface $\mathtt{I_{method}(K_i)}$ containing a method do:harm. Most methods from $\mathtt{I_{method}(K_i)}$ constitute useful (friendly) operations on the state space of K, while the method do:harm brings its state in disarray, so that it may be called only under special circumstances.

Let X be an instruction sequence such that

$$\mathtt{I_{required}(X)} \subseteq \mathtt{J} =_{\mathtt{def}} \mathtt{I_{provided}}(\mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1(s_1)} \oplus \cdots \oplus \mathtt{g_n.K_n(s_n)})$$

The application of $\mathtt{X}$ to a service family $\mathtt{g_1.K_1(s_1)} \oplus \cdots \oplus \mathtt{g_n.K_n(s_n)}$ while making use of the support of the (possibly non-ordinary) service SHRAT amounts to application of the thread $|\mathtt{X}|$ extracted from the instruction sequence $\mathtt{X}$ to the service family $\mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1(s_1)} \oplus \cdots \oplus \mathtt{g_n.K_n(s_n)}$ followed by forgetting f.SHRAT, thereby obtaining

$$\partial_{\mathtt{f}}(|\mathtt{X}| \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1(s_1)} \oplus \cdots \oplus \mathtt{g_n.K_n(s_n)})$$

Now the idea is that in a thread $\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}$, when effectuated in the context of a service family $\mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$, the reply of SHRAT on the call ok is positive if the computation of $\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ will not involve any call of the form $\mathtt{g_i.do{:}harm}$ while the reply is negative otherwise. Here, it is assumed that P and Q are threads arising from thread extraction of states of X so that the required interface of both P and Q is included in J as introduced above.

For example, taking $\mathtt{n} = 1$, if P contains a single occurrence of $\mathtt{g_1.do{:}harm}$, it still may be the case that this particular method call will certainly not be performed so that a positive reply of f.ok can be given by SHRAT. For instance if $\mathtt{K_1(s)}$ returns false on m1 then in

$$((\mathtt{g_1.do{:}harm} \circ \mathtt{S} \trianglelefteq \mathtt{g_1.m1} \trianglerighteq \mathtt{g_1.m2} \circ \mathtt{S}) \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{S}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1(s)}$$

the call of f.ok returns true and thus:

$$\partial_{\mathtt{f}}(((\mathtt{g_1.do{:}harm} \circ \mathtt{S} \trianglelefteq \mathtt{g_1.m1} \trianglerighteq \mathtt{g_1.m2} \circ \mathtt{S}) \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{S}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1(s)}) =$$

$$(\mathtt{g_1.m1} \circ \mathtt{g_1.m2} \circ \mathtt{S}) \bullet \mathtt{K_1(s)} = \mathtt{K_1(effect_{m2}(effect_{m1}(s)))}$$

With this idea in mind, a definition of the functionality of the lookahead service SHRAT can be given as follows. (RFS is defined in Section 2.1 above).

Definition 6. *The assertion* $\theta_{\mathtt{SHRAT}}$ *with arguments* $\mathtt{Q}, \mathtt{K_1}, \ldots, \mathtt{K_n}$ *is as follows:* $\theta_{\mathtt{SHRAT}} \equiv \exists\beta\colon \{\mathtt{ok}\}^+ \rightarrow \{\mathtt{true},\ \mathtt{false}\}[\mathtt{Q} \bullet \mathtt{g.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}\ \underline{\mathtt{sat}}\ \mathtt{noCALL}(\mathtt{g_i.do{:}}\mathit{harm})]$

Definition 7. SHRAT *determines its reply to a method call* $(\mathtt{Q} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}') \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *as follows: first determine the truth value (say* b *of the assertion* $\theta_{\mathtt{SHRAT}}$ *as in Definition 6) with arguments* $\mathtt{Q}, \mathtt{K_1}, \ldots, \mathtt{K_m}$*; then the reply given by* SHRAT *to the call* f.ok *equals* b *while the state is left unchanged.*

In this definition, the form of Φ matters. In particular, if one or more of the services $\mathtt{K_1}, \ldots, \mathtt{K_n}$ have prospecting capability and SHRAT is informed about that fact, then when evaluating $\theta_{\mathtt{SHRAT}}$, it is "known" that when evaluating $\mathtt{Q} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ the services $\mathtt{K_i}$ operate on the basis of the information that the service accessible via focus f behaves as an ordinary service, thus reducing the number of non-ordinary services in the service family at hand. It follows that replies of SHRAT are well defined also without the assumption that the services $\mathtt{K_1}, \ldots, \mathtt{K_n}$ are ordinary services.

3.5. Obtaining Lookahead Functionality with SHRAT

We first formalise with Proposition 5 the intuition that the sought foresight pattern cannot be provided by an ordinary service. Then Theorem 1 captures how SHRAT realises the required foresight pattern.

Proposition 5. *Let* $\mathtt{i} \in \{1, \ldots, \mathtt{n}\}$ *and let* $\mathtt{K_1}, \ldots, \mathtt{K_n}$ *be ordinary (i.e., non lookahead) services with* $\mathtt{do{:}harm} \in \mathtt{I_{method}(K_i)}$*. Let* H *be an ordinary service with method interface* $\mathtt{I_{method}(H)} = \{\mathtt{ok}\}$*.*

Then it cannot be the case that for all threads P *and* Q *both with required interfaces included in* $\mathtt{J} = \mathtt{I_{provided}}(\mathtt{f.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n})$ *the following holds: the reply of the service* H *to the first method call in the computation* $(\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}) \bullet \mathtt{f.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *equals* true *if and*

only if
$\mathtt{P} \bullet \mathtt{f.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ sat $\mathtt{noCALL(g_i.do{:}harm)}$.

Proof. There are two cases: $\mathtt{reply_{ok}(H)} = \mathtt{true}$ and $\mathtt{reply_{ok}(H)} = \mathtt{false}$. In the first case, choose $\mathtt{P} \equiv \mathtt{g_i.do{:}harm} \circ \mathtt{S}$ and $\mathtt{Q} \equiv \mathtt{S}$ and in the second case choose $\mathtt{P} \equiv \mathtt{S}$ and $\mathtt{Q} \equiv \mathtt{S}$. In both cases the required equivalence fails. □

Theorem 1. *Let* $\mathtt{i} \in \{1, \ldots, n\}$ *and let* $\mathtt{K_1}, \ldots, \mathtt{K_n}$ *be ordinary (i.e., non lookahead) services with* $\mathtt{do{:}harm} \in \mathtt{I_{K_i}}$, *and let* P *and* Q *be threads both with required interfaces included in* $\mathtt{J} = \{\mathtt{f.ok}\} \cup \mathtt{I_{provided}}(\mathtt{g_1.K_1} \oplus \ldots \oplus \mathtt{g_n.K_n})$.

Then the reply of the service SHRAT *(as defined above) to the first method call in the computation* $(\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ *equals* true *if and only if*
$\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_n.K_n}$ sat $\mathtt{noCALL(g_i.do{:}harm)}$

Proof. Without loss of generality, we will assume that $\mathtt{n} = \mathtt{i} = 1$ and we will write K instead of $\mathtt{K_1}$ and g instead of $\mathtt{g_1}$. Let V be a state space with $\mathtt{r} \in \mathtt{V}$ so that $\mathtt{K} = \mathtt{K(r)}$. We will distinguish two cases in this order: (i) the case that the computation $\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r)}$ involves a method call g.do:harm, and (ii) the case that it does not.

For case (i), suppose that the computation $\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r)}$ involves a method call g.do:harm. It is shown that the reply by SHRAT to the first method call of the computation $(\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r)}$ equals false. To this end, it suffices to prove that for each infinite sequence of Boolean values $\beta = \mathtt{b_0}, \mathtt{b_1}, \ldots$, the computation $\mathtt{P} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\}, \beta) \oplus \mathtt{g.K(r)}$ involves a basic action g.do:harm. If not, choose a thread $\mathtt{P'}$ with required interface included in J and $\mathtt{r'} \in \mathtt{V} = \mathtt{STATES(K)}$ such that (i) the computation from architerm $\mathtt{P'} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r')})$ involves a method call g.do:harm, (ii) for some finite or infinite sequence of Boolean values $\beta = \mathtt{b_1}, \mathtt{b_2}, \ldots$ the computation $\mathtt{P} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\}, \beta) \oplus \mathtt{g.K(r')}$ involves no basic action g.do:harm, (iii) the number n of steps of the computation of $\mathtt{P'} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r')}$ until its first call of g.do:harm is minimal, say $\mathtt{n_{min}}$. A contradiction is derived from a case distinction on the structure of $\mathtt{P'}$.

Consider first the case that $\mathtt{P'} \equiv \mathtt{S}$, $\mathtt{S} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r')}$ terminates at once and does not involve an occurrence of the basic action g.do:harm, thereby contradicting requirement (i) on $\mathtt{P'}$ and r'. A similar argument applies if $\mathtt{P'} \equiv \mathtt{D}$. Now suppose that $\mathtt{P'} \equiv \mathtt{P'_1} \trianglelefteq \mathtt{g.do{:}harm} \trianglerighteq \mathtt{P'_2}$. In this case, there is no β such that the computation $\mathtt{P'} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\}, \beta) \oplus \mathtt{g.K(r')}$ avoids performing basic action g.do:harm, thereby contradicting requirement (ii) on the choice of $\mathtt{P'}$ and r'. Next let $\mathtt{P'} \equiv \mathtt{P'_1} \trianglelefteq \mathtt{g.m} \trianglerighteq \mathtt{P'_2}$ for $\mathtt{m} \in \mathtt{I_K}$. Now, two cases are distinguished: $\mathtt{reply_m(r')} = \mathtt{true}$ and $\mathtt{reply_m(r')} = \mathtt{false}$. Consider the first case, then in one step, the architerm $(\mathtt{P'_1} \trianglelefteq \mathtt{g.m} \trianglerighteq \mathtt{P'_2}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r')}$ evolves to the architerm $\mathtt{P'_1} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(effect_m(r')}$. Now, it must be the case that (a) the computation from the latter architerm involves an occurrence of g.do:harm (because of assumption (i) on $\mathtt{P'}$ and $\mathtt{r'}$), (b) for some β (the same as for the combination $\mathtt{P'}$ and $\mathtt{r'}$) the computation $\mathtt{P'_1} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(effect_m(r'))}$ does not involve an occurrence of the basic action g.do:harm, and (c) the number of steps in the computation of $\mathtt{P'_1} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(effect_m(r'))}$ until the first call of g.do:harm equals $\mathtt{n_{min}} - 1$ which contradicts the minimality of $\mathtt{n_{min}}$. The case $\mathtt{reply_m(r')} = \mathtt{false}$ works in the same way. Finally consider the case $\mathtt{P'} \equiv \mathtt{P'_1} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{P'_2}$. There are two subcases: in the first step of the computation of $\mathtt{P'} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r')}$ SHRAT returns true (subcase 1) or it returns false (subcase 2).

(Subcase 1). If SHRAT returns true, then it is known that (a) the computation of $\mathtt{P'_1} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K(r')}$ involves an occurrence of g.do:harm (because of requirement (i) on $\mathtt{P'}$ and $\mathtt{r'}$), (b) there is a sequence α such that the computation $\mathtt{P'_1} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\}, \alpha) \oplus \mathtt{g.K(r')}$ does not contain an occurrence of g.do:harm (the existence of α is obtained from the definition of SHRAT and the fact that its most recent reply was positive), and (c) the number of steps in this computation until the first occurrence of g.do:harm equals $\mathtt{n_{min}} - 1$. Together, (a), (b), and (c) contradict the minimality of $\mathtt{n_{min}}$ for $\mathtt{P'}$ and $\mathtt{r'}$.

(Subcase 2).The case that SHRAT returns false begins with the observation (a) that because the computation $(\mathtt{P}'_1 \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{P}'_2) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}(\mathtt{r}')$ involves a call g.do:harm so does the computation from the next step (because of the assumption made on the value of the call f.ok) $\mathtt{P}'_2 \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}(\mathtt{r}')$; observation (b) reads that $\mathtt{b}_1 = \mathtt{false}$. To see this, notice that due to the definition of SHRAT, the fact that the most recent reply to a call f.ok was negative must have been caused by the fact that, for each α, each computation starting from the architerm $\mathtt{P}'_1 \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\alpha) \oplus \mathtt{g.K}(\mathtt{r}')$ involves a call g.do:harm. It follows from the latter that $\mathtt{b}_1$ (the reply to the recent call f.ok as encoded in β) must be false.

The computation from architerm $(\mathtt{P}'_1 \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{P}'_2)' \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus g.K(r')$ involves no call g.do:harm, and together with $\mathtt{b}_1 = \mathtt{false}$, it follows that the computation from architerm $\mathtt{P}'_2 \bullet \mathtt{f.RFS}(\beta') \oplus \mathtt{g.K}(\mathtt{r}')$ (with $\beta' = \mathtt{b}_2, \mathtt{b}_3, \ldots$) contains no call g.do:harm either. Finally notice (c) that the length of the latter computation is $\mathtt{n}_{\mathtt{min}} - 1$. Together, (a), (b), and (c) contradict the minimality of $\mathtt{n}_{\mathtt{min}}$, which concludes subcase 2.

In the second part of the proof, regarding case (ii) ($\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}(\mathtt{r})$ does not involve a method call g.do:harm) one may assume that the computation of $\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}(\mathtt{r})$ does not contain a method call g.do:harm.

We show that the first reply of SHRAT to a call f.ok (in the computation of $(\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}(\mathtt{r})$) equals true. Consider the finite or infinite sequence of Boolean values $\alpha = a_0, a_1, \ldots$ such that the successive calls (if any) of f.ok in the computation of $\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}(\mathtt{r})$ are replied to by SHRAT as encoded in α. Making use of the fact that K is not a lookahead service, and that, consequently, its replies will only depend on the state of the service at the moment of a call being made and, of course, on the particular method which is being called, while its replies cannot not depend on future behaviour of the system, the architerm $\mathtt{P} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\alpha) \oplus \mathtt{g.K}(\mathtt{r})$ produces the same computation, and for that reason, it does not contain an occurrence of the method call g.do:harm. Now it follows from clause (b) in the definition of the behaviour of SHRAT that the reply given to the first call of f.ok in the computation of $(\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}(\mathtt{r})$ equals true. □

Definition 8. $\mathtt{SHRAT}^{\mathtt{fst/fss}}$ *is the restriction of* SHRAT *to finite state threads (fst) and finite state services (fss).*

Proposition 6. $\mathtt{SHRAT}^{\mathtt{fst/fss}}$ *is a computable service taking as additional inputs for the determination of a reply on a method call* ok *a combination of two finite PGLB instruction sequences for the representation of the threads* P *and* Q *and finite state machines for the representation of the services* $\mathtt{K}_1, \ldots, \mathtt{K}_\mathtt{m}$.

Proof. A decision method for Φ (in the definition of SHRAT) is as follows: Given arguments $\mathtt{P}, \mathtt{K}_1, \ldots, \mathtt{K}_\mathtt{n}$ let $\mathtt{u}_\mathtt{p} = \#_{\mathtt{STATES}}(\mathtt{P}), \mathtt{u}_1 = \#_{\mathtt{STATES}}(\mathtt{K}_1), \ldots, \#_{\mathtt{STATES}}(\mathtt{K}_\mathtt{n})$ be the respective sizes (number of elements) of the respective state spaces of these components. We find that a computation path without a call $\mathtt{g}_\mathtt{i}$.do:harm $\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g.K}_1 \oplus \cdots \oplus \mathtt{g.K}_\mathtt{n}$ can have at most $\mathtt{u} = \mathtt{u}_\mathtt{p} \cdot \mathtt{u}_1 \cdot \ldots \cdot \mathtt{u}_\mathtt{n}$ steps unless it gets into a cycle. So by checking all paths of a length of u steps it can be found if there is a (possibly cyclic) path without a call to $\mathtt{g}_\mathtt{i}$.do:harm, in which case the sequence of successive replies to calls f.ok provides a cyclic sequence β as required, while otherwise, no such β exists. □

Definition 9. $\mathtt{SHRAT}^{\mathtt{fst/fss/tts}}$ *is the restriction of* SHRAT *to finite state threads (fst) and a combination of finite state services (fss) with* $\mathtt{K}_1$ *a service for a Turing tape (tts).*

Proposition 7. $\mathtt{SHRAT}^{\mathtt{fst/fss/tts}}$ *is a non-computable service taking as additional inputs for the determination of a reply on a method call* ok *a combination of two finite PGLB instruction sequences for the representation of the threads* P *and* Q*, a bit sequence for the tape architerm for* $\mathtt{K}_1$*, and finite state machines for the representation of the services* $\mathtt{K}_2, \ldots, \mathtt{K}_\mathtt{m}$.

Proof. We consider the collection U of PGLB instruction sequences with required interface included in $\mathtt{g}_1.\mathtt{I}_\mathtt{r}(\mathtt{TTS})$ and such that termination can only occur at the last instruction

(which must be !). Termination of computations of the form $|\mathtt{P};!| \bullet \mathtt{g}_1.\mathtt{TSS}$ is well known to be not computationally decidable. We find that for $\mathtt{P};! \in \mathtt{U}$: the computation from $|\mathtt{P};!| \bullet \mathtt{g}_1.\mathtt{TSS}$ terminates if and only if the computation from $|\mathtt{P};\mathtt{g}_2.\mathtt{do{:}harm};!| \bullet \mathtt{g}_1.\mathtt{TSS} \oplus \mathtt{g}_2.\mathtt{K}_2$ involves a call $\mathtt{g}_2.\mathtt{do{:}harm}$, if and only if the initial call $\mathtt{f.ok}$ in the computation $(|\mathtt{P};\mathtt{g}_2.\mathtt{do{:}harm};!| \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{S}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g}_1.\mathtt{TSS} \oplus \mathtt{g}_2.\mathtt{K}_2$ receives reply $\mathtt{false}$. If $\mathtt{SHRAT}$ is a computable service, it follows that the halting problem for PGLB instruction sequences over a Turing tape service is decidable as well, which is not the case. □

The notion of a lookahead service leaves open the possibility of the interaction of multiple lookahead services, and suggests possible generalisations of Theorem 1, with one or more of the $\mathtt{K}_\mathtt{i}$ featuring lookahead capability. This suggestion leads to the following result.

Theorem 2. *The condition for Theorem 1 that* $\mathtt{K}_1, \ldots, \mathtt{K}_\mathtt{n}$ *are ordinary services is a necessary condition for Theorem 1.*

Proof. Let $\mathtt{n} = \mathtt{i} = 2$ in the notation of Theorem 1. We assume that $\mathtt{I}_{\mathsf{method}}(\mathtt{K}_1) = \{\mathtt{m1}, \mathtt{m2}, \mathtt{m3}\}$ and $\mathtt{I}_{\mathsf{method}}(\mathtt{K}_2) = \{\mathtt{do{:}harm}, \mathtt{u1}, \mathtt{u2}\}$. We consider the following example: $\mathtt{P} \equiv (\mathtt{g}_2.\mathtt{u1} \circ (\mathtt{g}_2.\mathtt{do{:}harm} \circ \mathtt{S} \trianglelefteq \mathtt{g}_1.\mathtt{m2} \trianglerighteq \mathtt{S})) \trianglelefteq \mathtt{f.ok} \trianglerighteq (\mathtt{g}_2.\mathtt{u2} \circ (\mathtt{g}_2.\mathtt{do{:}harm} \circ \mathtt{S} \trianglelefteq \mathtt{g}_1.\mathtt{m3} \trianglerighteq \mathtt{S}))) \trianglelefteq \mathtt{g}_1.\mathtt{m1} \trianglerighteq \mathtt{S}$ and $\mathtt{Q} \equiv \mathtt{S}$. Moreover, $\mathtt{K}_2$ is an ordinary service such that $\mathtt{do{:}harm} \in \mathtt{I}_{\mathsf{method}}(\mathtt{K}_2)$. $\mathtt{K}_1$, however, is a non-ordinary service, which is defined as follows:

(i) $\mathtt{K}_1$ has a single state only, so none of its actions results in a state change.

(ii) On $\mathtt{m2}$ and $\mathtt{m3}$, the reply is always $\mathtt{true}$.

(iii) on $\mathtt{m1}$, the evaluation of $\theta(\mathtt{s}, \mathtt{m1})$ for $\mathtt{K}_1$ in a context $(\mathtt{R}_1 \trianglelefteq \mathtt{g}_1.\mathtt{m1} \trianglerighteq \mathtt{R}_2) \bullet \mathtt{f.H} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ depends on prospection as follows: $\theta(\mathtt{s}, \mathtt{m1})$ is $\mathtt{false}$ (and with it the reply of $\mathtt{f.K}_1$ to the call $\mathtt{f.m}$), if and only if there are two different ordinary services $\mathtt{K}_\mathtt{a}$ and $\mathtt{K}_\mathtt{b}$ so that $\mathtt{R}_1 \bullet \mathtt{f.H} \oplus \mathtt{g}_1.\mathtt{K}_\mathtt{a} \oplus \mathtt{g}_2.\mathtt{K}_2$ $\underline{\mathtt{sat}}$ $\mathtt{CALL}(\mathtt{g}_2.\mathtt{u1})$ and $\mathtt{R}_1 \bullet \mathtt{f.H} \oplus \mathtt{g}_1.\mathtt{K}_\mathtt{b} \oplus \mathtt{g}_2.\mathtt{K}_2$ $\underline{\mathtt{sat}}$ $\mathtt{CALL}(\mathtt{g}_2.\mathtt{u2})$.

(iv) On $\mathtt{m1}$ there is no state change.

Now adopting the definition of $\mathtt{SHRAT}$ as given above, consider the computation starting from $(\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$. This computation, however, does not comply with the requirement on $\mathtt{f.SHRAT}$ as given in Theorem 1. In particular, it is the case that:

(a) $\mathtt{P} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ $\underline{\mathtt{sat}}$ $\mathtt{noCALL}(\mathtt{g}_2.\mathtt{do{:}harm})$ while

(b) The first call of $\mathtt{f.ok}$ in the computation $(\mathtt{P} \trianglelefteq \mathtt{f.ok} \trianglerighteq \mathtt{Q}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ receives reply $\mathtt{false}$ from $\mathtt{SHRAT}$.

To prove (b), first notice that the call $\mathtt{f.ok}$ requests $\mathtt{SHRAT}$ to evaluate the reply condition $\theta_{\mathtt{SHRAT}}$. It must be shown that for each reply function $\beta\colon \{\mathtt{ok}\}^+ \to \{\mathtt{true}, \mathtt{false}\}$ it is the case that $\mathtt{P} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ $\underline{\mathtt{sat}}$ $\mathtt{CALL}(\mathtt{g}_2.\mathtt{do{:}harm})$. The latter computation starts with the call $\mathtt{g}_1.\mathtt{m1}$ which will return $\mathtt{false}$ if and only if two different ordinary services $\mathtt{K}_\mathtt{a}$ and $\mathtt{K}_\mathtt{b}$ exist so that

$\mathtt{R}_1 \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_\mathtt{a} \oplus \mathtt{g}_2.\mathtt{K}_2$ $\underline{\mathtt{sat}}$ $\mathtt{CALL}(\mathtt{g}_2.\mathtt{u1})$, and

$\mathtt{R}_1 \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_\mathtt{b} \oplus \mathtt{g}_2.\mathtt{K}_2$ $\underline{\mathtt{sat}}$ $\mathtt{CALL}(\mathtt{g}_2.\mathtt{u2})$, with

$\mathtt{R}_1 = (\mathtt{g}_2.\mathtt{u1} \circ (\mathtt{g}_2.\mathtt{do{:}harm} \circ \mathtt{S} \trianglelefteq \mathtt{g}_1.\mathtt{m2} \trianglerighteq \mathtt{S})) \trianglelefteq \mathtt{f.ok} \trianglerighteq (\mathtt{g}_2.\mathtt{u2} \circ (\mathtt{g}_2.\mathtt{do{:}harm} \circ \mathtt{S} \trianglelefteq \mathtt{g}_1.\mathtt{m3} \trianglerighteq \mathtt{S})))$.

There are two cases $\beta(\mathtt{ok}) = \mathtt{true}$ and $\beta(\mathtt{ok}) = \mathtt{false}$. If $\beta(\mathtt{ok}) = \mathtt{true}$, then both computations (for $\mathtt{K}_\mathtt{a}$ and for $\mathtt{K}_\mathtt{b}$) proceed with a reply $\mathtt{true}$ in the next step, i.e., the call $\mathtt{f.ok}$, and both computations will subsequently perform a call of $\mathtt{g}_2.\mathtt{u1}$ and will therefore not perform a call of $\mathtt{g}_2.\mathtt{u2}$. It follows, in this case, that the criterion for returning $\mathtt{false}$ on the call $\mathtt{g}_1.\mathtt{m1}$ is not satisfied, and $\mathtt{true}$ is returned as a reply. For the case $\beta(\mathtt{ok}) = \mathtt{false}$, a similar argument leads to the same conclusion.

We find that the reply of $\mathtt{g}_1.\mathtt{K}_1$ on the method call $\mathtt{g}_1.\mathtt{m1}$ in the computation starting from $\mathtt{P} \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ must be $\mathtt{true}$ and therefore the next state of the computation is $\mathtt{R}_1 \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$. Now, two cases can be distinguished: $\beta(\mathtt{ok}) = \mathtt{true}$ and $\beta(\mathtt{ok}) = \mathtt{false}$.

In the first case, the computation of $\mathtt{R}_1 \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ proceeds to $(\mathtt{g}_2.\mathtt{u1} \circ (\mathtt{g}_2.\mathtt{do{:}harm} \circ \mathtt{S} \trianglelefteq \mathtt{g}_1.\mathtt{m2} \trianglerighteq \mathtt{S})) \bullet \mathtt{f.RFS}(\{\mathtt{ok}\},\beta) \oplus g_1.K_b \oplus g_2.K_2$. After a step for the

call $\mathtt{g_2.u1}$, the call $\mathtt{g_1.m2}$ will produce reply `true` (on the basis of the specification of $\mathtt{K_1}$), so that the subsequent step is a call $\mathtt{g_2.do{:}harm}$. A similar argument applies if $\beta(ok) = false$. This proves that the call $\mathtt{g_2.do{:}harm}$ is unavoidable, which proves (b) above.

For (a), it must be shown that $(\mathtt{R_1} \trianglelefteq \mathtt{g_1.m1} \trianglerighteq \mathtt{S}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \mathtt{g_2.K_2}$ $\underline{\mathtt{sat}}$ $\mathtt{noCALL(g_2.do{:}harm)}$. In the second step of the computation, $\mathtt{K_1}$ produces a reply to the call $\mathtt{g_1.m1}$. The latter reply of $\mathtt{K_1}$ equals `false`. To see this, one notices that when choosing $\mathtt{K_a}$ (as a possible future for itself, such that the next reply on `m2` is `false` and the next reply on `m3` is `true`), the reply of `SHRAT` to the next call of `f.ok` would be `false`, because otherwise $\mathtt{g_2.do{:}harm}$ would become unavoidable. Moreover, the action $\mathtt{g_2.m2}$ will take place.

Alternatively, upon contemplating $\mathtt{K_b}$ as a possible future behaviour of $\mathtt{K_1}$ so that the next reply to a call `m2` is `true` and the next reply on `m3` is `false`, the reply of `SHRAT` to the next call of `f.ok` would be `true`, because otherwise $\mathtt{g_2.do{:}harm}$ would become unavoidable. Moreover the action $\mathtt{g_2.u2}$ will take place.

So, given the fact that these two different computations for $\mathtt{K_a}$ and $\mathtt{K_b}$ are possible, the reply of $\mathtt{K_1}$ to the initial method call $\mathtt{g_1.m1}$ (in the computation $(\mathtt{R_1} \trianglelefteq \mathtt{g_1.m1} \trianglerighteq \mathtt{S}) \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \mathtt{g_2.K_2}$) equals `false` (due to the particular non-ordinary definition of $\mathtt{K_1}$). So the computation proceeds to $\mathtt{S} \bullet \mathtt{f.SHRAT} \oplus \mathtt{g_1.K_1} \oplus \mathtt{g_2.K_2}$ which will not perform any basic action any longer such that $\mathtt{noCALL(g_2{:}do{:}harm)}$ is satisfied, which proves (a). □

4. Promoting a Method Call

Rather than for avoiding a certain method call, prospection may, for whatever reason, be used to promote it taking place, i.e., to guarantee that it will take place if that guarantee can be given. However, the simplest foresight pattern to that extent fails. Instead of an action `do:harm`, which is preferably avoided, the presence of an action `do:no_harm` which is preferably performed may be assumed.

Theorem 3. *Let* $\mathtt{m}, \mathtt{i} \in \mathbb{N}$, $1 \leq \mathtt{i} \leq \mathtt{m}$ *and let* $\mathtt{K_1}, \ldots, \mathtt{K_m}$ *be ordinary services with* $\mathtt{do{:}no\_harm} \in \mathtt{I_{method}(K_i)}$. *There is no single state service* `H` *with* $\mathtt{I_{method}(H)} = \{\mathtt{find}\}$ *such that the following holds:*

for all threads `P` *and* `Q` *with required interfaces included in* $\mathtt{J} = \mathtt{I_{provided}(h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_m.K_m)}$, *the reply of the service* `H` *to the first method call in the computation* $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_m.K_m}$ *equals* `true` *if and only if it is the case that* $\mathtt{P} \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_m.K_m}$ $\underline{\mathtt{sat}}$ $\mathtt{CALL(g_i.do{:}no\_harm)}$.

Proof. Suppose otherwise, and let `H` meet the requirements of the Theorem. Using the notation of Theorem 3, let $\mathtt{P} = \mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq (\mathtt{g_i.do{:}no\_harm} \circ \mathtt{S})$ and $\mathtt{Q} = \mathtt{g_i.do{:}no\_harm} \circ \mathtt{S}$.

First assume that `H` replies `true` on the first call `h.find`. Then after processing the positive reply, the computation returns to precisely the same state as it was in before said method call. Now, because `H` is a single-state service, the computation enters in a loop from which it will not escape and during which no method call $\mathtt{g_i.do{:}no\_harm}$ will be performed. It follows that on a positive reply to the first call to `H`, it is not the case that $\mathtt{P} \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_m.K_m}$ will perform the required method call, so that the requirement on `H` is violated in that case.

Hence, it may be assumed that the first method call `h.find` receives reply `false`. In that case, given the assumptions made on `H` in the theorem, such will also be the case of the computation $\mathtt{P} \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_m.K_m}$, which is precisely the same and which is processed with `H` from the same unique state. In that case, because of the negative reply, the method call $\mathtt{g_i.do{:}no\_harm}$ will be performed in the computation of $\mathtt{P} \bullet \mathtt{h.H} \oplus \mathtt{g_1.K_1} \oplus \cdots \oplus \mathtt{g_m.K_m}$ so that a mismatch with the requirement on `H` arises. □

Theorem 3 generalises to stateful prospecting services. First notice that the definition of a single state prospecting service can be easily adapted to involve several methods:

Definition 10 (Single state multi-method prospecting service)**.** $\mathtt{L}^{\mathtt{m}_1,\ldots,\mathtt{m}_t}{}_{\theta_1,\ldots,\theta_t}$ *uses* θ_i *to determine which reply to produce on a call of method* $\mathtt{m}_i$*. Definition 5 is adapted by using a quantifier of type* $\{\mathtt{m}_1,\ldots,\mathtt{m}_t\}^+ \to \{\mathtt{true},\mathtt{false}\}$ *instead.*

Definition 11 (Stateful multi-method prospecting service)**.** *A prospecting service* L *with method interface* J*, a state space* S *and an effect function* η: $\mathtt{J} \times \mathtt{S} \times \{\mathtt{true},\mathtt{false}\} \to \mathtt{S}$ *can be defined if for each* $\mathtt{s} \in \mathtt{S}$*, and* $\mathtt{z} \in \mathtt{J}$*, a lookahead condition* $\theta(\mathtt{s},\mathtt{m}) \in \mathtt{CLA}_{\mathtt{I}}^{\mathtt{I}_{dm}(\mathtt{I})}$ *is given which determines the reply on method* $\mathtt{m} \in \mathtt{J}$ *in state* s *of* L*.*

A workable notation for a service of this form with method interface J is $\mathtt{L}(\mathtt{Z},\mathtt{S},\mathtt{s}_0,\eta,\theta)$ with η: $Z \times S \times \{true, false\} \to S$ and θ: $\mathtt{Z} \times \mathtt{S}$ a lookahead condition. We will refer to services of this form as stateful with an ordinary effect function.

Services with multiple states will use the result of the evaluation of a reply condition for determining the next state transition (the result of the effect function). Then, the effect function has type $\mathtt{effect}$: $\mathtt{Z} \times \mathtt{S} \times \mathtt{BOOL} \to \mathtt{S}$ where the third argument is taken to be the reply value which is given in the same step.

Proposition 8. *Let* $\mathtt{n},\mathtt{i} \in \mathbb{N}$, $1 \leq \mathtt{i} \leq \mathtt{n}$ *and let* $\mathtt{J}_1,\ldots,\mathtt{J}_n$ *be method interfaces with* $\mathtt{do{:}no\_harm} \in \mathtt{J}_i$*. Let* $\mathtt{J} = \{\mathtt{h.find}\} \cup \mathtt{g}_1.\mathtt{J}_1 \cup \ldots \cup \mathtt{g}_n.\mathtt{J}_n$*. There is no stateful prospecting service* $\mathtt{L} \equiv \mathtt{L}(\{\mathtt{find}\},\mathtt{S},\mathtt{s}_0,\eta,\theta)$ *for which the following holds:*

for all n*-tuples of ordinary services* $\mathtt{K}_1,\ldots,\mathtt{K}_n$*, with* $\mathtt{I}_{\mathsf{method}}(\mathtt{K}_1) \subseteq \mathtt{J}_1,\ldots,\mathtt{I}_{\mathsf{method}}(\mathtt{K}_n) \subseteq \mathtt{J}_n$*, for all states* $\mathtt{s} \in \mathtt{S}$*, and for all threads* P *and* Q *with required interfaces included in* J*, the reply of the service* $\mathtt{L}(\{\mathtt{find}\},\mathtt{S},\mathtt{s},\eta,\theta)$ *to the first method call in the computation* $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.L}(\{\mathtt{find}\},\mathtt{S},\mathtt{s},\eta,\theta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n$ *equals* true *if and only if* $\mathtt{P} \bullet \mathtt{h.L}(\{\mathtt{find}\},\mathtt{S},\eta(\mathtt{find},\mathtt{s},\mathtt{true}),\eta,\theta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n \underline{\mathtt{sat}}\ \mathtt{CALL}(\mathtt{g}_i.\mathtt{do{:}no\_harm})$.

Proof. Suppose otherwise, and let $\mathtt{L}(\{\mathtt{find}\},\mathtt{S},\mathtt{s}_0,\eta,\theta)$ meet the stated requirements. Choose threads P and Q as follows: $\mathtt{P} = \mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq (\mathtt{g}_i.\mathtt{do{:}no\_harm} \circ \mathtt{S})$ and $\mathtt{Q} = \mathtt{g}_i.\mathtt{do{:}no\_harm} \circ \mathtt{S}$ and let $\mathtt{s} \in \mathtt{S}$: $\mathtt{A}_\mathtt{s} \equiv \mathtt{P} \bullet \mathtt{h.L}(\mathtt{find},\mathtt{S},\mathtt{s},\eta,\theta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_n.\mathtt{K}_n$. Suppose $\mathtt{A}_\mathtt{s}\ \underline{\mathtt{sat}}\ \neg\theta(\mathtt{s},\mathtt{find})$. For $\mathtt{s} \in \mathtt{S}$ let $\mathtt{U}_\mathtt{s}$ be the smallest subset of S which contains $\eta(\mathtt{find},\mathtt{s},\mathtt{true})$ and which contains $\eta(\mathtt{find},\mathtt{t},\mathtt{true})$ for each $\mathtt{t} \in \mathtt{S}$. It must be the case that for all $\mathtt{t} \in \mathtt{U}_\mathtt{s}$, $\mathtt{A}_\mathtt{t}\ \underline{\mathtt{sat}}\ \theta(\mathtt{t},\mathtt{find})$, because otherwise the computation from architerm $\mathtt{A}_{\eta(\mathtt{find},\mathtt{s},\mathtt{true})}$ will perform a method call $\mathtt{g}_i.\mathtt{do{:}no\_harm}$ after as many steps as needed to arrive at a state $\mathtt{t} \in \mathtt{U}_\mathtt{s}$ with $\mathtt{A}_\mathtt{t}\ \underline{\mathtt{sat}}\ \neg\theta(\mathtt{t},\mathtt{find})$ so that, given the requirements on L, $\mathtt{A}_\mathtt{s}\ \underline{\mathtt{sat}}\ \theta(\mathtt{s},\mathtt{find})$ must be expected instead, thereby contradicting the above assumption that $\mathtt{A}_\mathtt{s}\ \underline{\mathtt{sat}}\ \neg\theta(\mathtt{s},\mathtt{find})$. It follows that for all states s, of $\mathtt{L}(\{\mathtt{find}\},\mathtt{S},\mathtt{s},\eta,\theta)$ it is the case that $\mathtt{A}_\mathtt{s}\ \underline{\mathtt{sat}}\ \theta(\mathtt{s},\mathtt{find})$ whence $\mathtt{A}_{\mathtt{s}_0}\ \underline{\mathtt{sat}}\ \mathtt{PERP}$ without ever performing $\mathtt{g}_i.\mathtt{do{:}no\_harm}$, from which it follows, with the requirements on L in the theorem, that $\mathtt{A}_{\mathtt{s}_0}\ \underline{\mathtt{sat}}\ \neg\theta(\mathtt{s}_0,\mathtt{find})$ should hold, which yields a contradiction with the facts just established about L, thus concluding the proof. □

Method Call Promotion with a Single State Prospecting Service: MCP

The single state service MCP has state s and method interface $\{\mathtt{find}\}$, and it is made accessible via focus h, which differs from $\mathtt{g}_1,\ldots,\mathtt{g}_m$. In a context $(\mathtt{P} \trianglelefteq \mathtt{f.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_m.\mathtt{K}_m$, the intended foresight pattern comprises that MCP produces replies in such a way that (i) a shortest path to a call of $\mathtt{g}_i.\mathtt{do{:}no\_harm}$ is found, and (ii) a positive reply is preferred in case (i) with both choices shortest paths to a call of $\mathtt{g}_i.\mathtt{do{:}no\_harm}$ of equal lengths can be found or (ii) no such paths can be found at all.

A reply r to the call h.find (i.e. $\theta(\mathtt{s},\mathtt{find})$, in the context $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_m.\mathtt{K}_m$ as mentioned above) is computed as follows: $\mathtt{r} = \theta(\mathtt{s},\mathtt{find}) = \psi \vee \neg\chi$ with

$\psi \equiv \forall\alpha$: $\{\mathtt{find}\}^+ \to \{\mathtt{true},\mathtt{false}\}[$
$\quad \mathtt{Q} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\alpha) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_m.\mathtt{K}_m\ \underline{\mathtt{sat}}\ \mathtt{noCALL}(\mathtt{g}_i.\mathtt{do{:}no\_harm})]$

$\chi \equiv \forall\alpha$: $\{\mathtt{find}\}^+ \to \{\mathtt{true},\mathtt{false}\}, \forall\mathtt{k} \in \mathbb{N}^+[$
$\quad \mathtt{P} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\alpha) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_m.\mathtt{K}_m\ \underline{\mathtt{sat}}\ \mathtt{CALLat}(\mathtt{k},\mathtt{g}_i.\mathtt{do{:}no\_harm})$

$$\begin{array}{l}\Longrightarrow\ \exists\gamma\colon \{\mathtt{find}\}^+ \to \{\mathtt{true},\mathtt{false}\}, \exists \mathtt{l} \in \mathbb{N}^+ [\, \mathtt{l} < \mathtt{k}\ \&\\ \qquad \mathtt{Q} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\gamma) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}\ \underline{\mathtt{sat}}\ \mathtt{CALLat}(\mathtt{l}, \mathtt{g}_\mathtt{i}.\mathtt{do{:}no\_harm})]\\ ]\end{array}$$

In other words: the reply is `true` if upon processing that reply (i.e., proceeding with $\mathtt{P}$), under the assumption that $\mathtt{H}$ is non-deterministic, a shortest path (of a computation of $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.H} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$) to a call of $\mathtt{g}_\mathtt{i}$.do:no_harm can be found, or if such a path does not exist.

Or, phrased alternatively, the reply is `false` if only upon proceeding with $\mathtt{Q}$ a path in the computation of $\mathtt{Q} \bullet \mathtt{h.H} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ to a call of $\mathtt{g}_\mathtt{i}$.do:no_harm (with non-deterministic $\mathtt{H}$) can be found, which is shorter than any such path in the computation of $\mathtt{P} \bullet \mathtt{h.H} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$.

Proposition 9. *Let* $\mathtt{m}, \mathtt{i} \in \mathbb{N}$, $1 \leq \mathtt{i} \leq \mathtt{m}$ *and let* $\mathtt{K}_1, \ldots, \mathtt{K}_\mathtt{m}$ *be ordinary (i.e., non-lookahead) services with* $\mathtt{do{:}no\_harm} \in \mathtt{I}_{\mathsf{method}}(\mathtt{K}_\mathtt{i})$.

Then, for all threads $\mathtt{P}$ *with* $\mathtt{I}_{\mathsf{required}}(\mathtt{P}) \subseteq \mathtt{J} = \{\mathtt{h.find}\} \cup \mathtt{g}_1.\mathtt{J}_1 \cup \ldots \cup \mathtt{g}_\mathtt{m}.\mathtt{J}_\mathtt{m}$, *the following implication holds: if there is a function* $\beta \in \{\mathtt{find}\}^+ \to \{\mathtt{true},\mathtt{false}\}$ *such that*
$\mathtt{P} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}\ \underline{\mathtt{sat}}\ \mathtt{CALL}(\mathtt{g}_\mathtt{i}.\mathtt{do{:}no\_harm})$ *then also*
$\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}\ \underline{\mathtt{sat}}\ \mathtt{CALL}(\mathtt{g}_\mathtt{i}.\mathtt{do{:}no\_harm})$.

Proof. Without loss of generality and for the ease of notation, We will assume that $\mathtt{m} = 2$ and $\mathtt{i} = 2$. Let for natural number $\mathtt{k} > 0$, $\chi_\mathtt{k}$ be the following assertion:

> Let $\mathtt{K}_1, \mathtt{K}_2$ be ordinary (i.e., non-lookahead) services with $\mathtt{do{:}no\_harm} \in \mathtt{I}_{\mathtt{method}}(\mathtt{K}_2)$, and let $\mathtt{P}$ be a thread with required interface included in $\mathtt{J}$, then if there is a Boolean function β such that
> $\mathtt{P} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2\ \underline{\mathtt{sat}}\ \mathtt{CALLat}(\mathtt{k}, \mathtt{g}_2.\mathtt{do{:}no\_harm})$,
> then there is a natural number $\mathtt{l} \leq \mathtt{k}$ such that
> $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2\ \underline{\mathtt{sat}}\ \mathtt{CALLat}(\mathtt{l}, \mathtt{g}_2.\mathtt{do{:}no\_harm})$.

It is immediate that $\forall \mathtt{k}.\chi_\mathtt{k}$ implies Proposition 10. To prove that for all $\mathtt{k} > 0$, $\chi_\mathtt{k}$ holds, use induction on $\mathtt{k}$ with $\mathtt{k} = 1$ as the basis. If $\mathtt{k} = 1$ and $\mathtt{P} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ performs a basic instruction $\mathtt{g}_2$.do:no_harm as the first step then $\mathtt{P} \equiv \mathtt{P}_1 \trianglelefteq \mathtt{g}_2.\mathtt{do{:}no\_harm} \trianglerighteq \mathtt{P}_2$, for some threads $\mathtt{P}_1$ and $\mathtt{P}_2$ from which it follows that $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ performs $\mathtt{g}_2$.do:no_harm as the first step.

Now let $\mathtt{k} = \mathtt{l} + 1, \mathtt{l} > 0$. In case $\mathtt{P} = \mathtt{S}$ or $\mathtt{P} = \mathtt{D}$, the required implication is immediate as an instance of material implication. In case $\mathtt{P} \equiv \mathtt{P}_1 \trianglelefteq \mathtt{g}_2.\mathtt{do{:}no\_harm} \trianglerighteq \mathtt{P}_2$, the implication is immediate too. The remaining cases to be checked are (a) $\mathtt{P} \equiv \mathtt{P}_1 \trianglelefteq \mathtt{g}_\mathtt{j}.\mathtt{m}_\mathtt{j} \trianglerighteq \mathtt{P}_2$ with $\mathtt{j} \in \{1, 2\}$ and $\mathtt{m}_\mathtt{j} \in \mathtt{J}_\mathtt{j}$, except the case that $\mathtt{j} = 2$ and $\mathtt{m}_\mathtt{j} \equiv \mathtt{do{:}no\_harm}$, and (b) $\mathtt{P} \equiv \mathtt{P}_1 \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{P}_2$.

In case (a) there are two subcases, $\mathtt{j} = 1, \mathtt{j} = 2$ both of which have two subcases: $\mathtt{K}_\mathtt{j}$ returns `true` on the call $\mathtt{g}_\mathtt{j}.\mathtt{m}_1$ and $\mathtt{K}_\mathtt{j}$ returns `false` of the call $\mathtt{g}_\mathtt{j}.\mathtt{m}_\mathtt{j}$. Each of these cases works in the same way. We consider the case that $\mathtt{j} = 1$ and that `true` is returned. Now, $\mathtt{P} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ performs a single-step transition to $\mathtt{P}_1 \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\frac{\partial}{\partial \mathtt{m}_1}(\mathtt{K}_1) \oplus \mathtt{g}_2.\mathtt{K}_2$. Here, $\frac{\partial}{\partial \mathtt{f}}(\mathtt{K})$ denotes the state of a service $\mathtt{K}$ just after having replied to a call of method $\mathtt{f} \in \mathtt{I}_\mathtt{K}$. It follows that $\mathtt{P}_1 \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\frac{\partial}{\partial \mathtt{m}_1}(\mathtt{K}_1) \oplus \mathtt{g}_2.\mathtt{K}_2$ performs $\mathtt{g}_1$.do:no_harm in at most $\mathtt{l}$ steps. Now the induction hypothesis can be applied to the latter architerm, which yields that the computation of $\mathtt{P}_1 \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\frac{\partial}{\partial \mathtt{m}_1}(\mathtt{K}_1) \oplus \mathtt{g}_2.\mathtt{K}_2$ will perform a basic instruction $\mathtt{g}_2$.do:no_harm within the first $\mathtt{l}$ steps from which it follows that computing $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ involves a call of $\mathtt{g}_2$.do:no_harm within the first $\mathtt{k} = \mathtt{l} + 1$ steps.

In case (b), there are again two subcases: (b/t) $\beta_1 = \mathtt{true}$, and (b/f) $\beta_1 = \mathtt{false}$. In case (b/t), upon performing a first step, the computation proceeds from $\mathtt{P}_1 \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ arriving at a method call $\mathtt{g}_\mathtt{i}$.do:no_harm at the $\mathtt{l}$-th step. Using the induction hypothesis, the computation of $\mathtt{P}_1 \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ performs a method call $\mathtt{g}_2$.do:no_harm within the first $\mathtt{l}$ steps. Now, once more, there are two cases: (b/t/t) the reply on the first method call in the computation of $(\mathtt{P}_1 \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{P}_2) \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ is `true`, and (b/t/f) the mentioned reply equals `false`. In the first case, it is immediate that the computation of $(\mathtt{P}_1 \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{P}_2) \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ contains a call of $\mathtt{g}_2$.do:no_harm within $\mathtt{l} + 1$ steps. In the second case (b/t/f) in view of the definition of MCP, with $\beta' = \lambda \mathtt{x} \in \{\mathtt{find}\}^+.\beta(\mathtt{find}^\frown \mathtt{x})$, it must be so that the computation of $\mathtt{P}_2 \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta') \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ performs a call of $\mathtt{g}_\mathtt{i}$.do:no_harm, and in fact within less than $\mathtt{l}$ steps. Now, with the induction hypothesis, it follows that the computation of $\mathtt{P}_2 \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ involves a call of $\mathtt{g}_2$.do:no_harm within $\mathtt{l}$ steps, from which it follows (with the assumption for case (b/t/f), that the computation of $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \mathtt{g}_2.\mathtt{K}_2$ involves a call of $\mathtt{g}_2$.do:no_harm within $\mathtt{l} + 1$ steps. The argument in case (b/f) is very similar to the case for (b/t), and is deleted for that reason. □

Proposition 10. *Let* $\mathtt{m}, \mathtt{i} \in \mathbb{N}$, $1 \leq \mathtt{i} \leq \mathtt{m}$ *and let* $\mathtt{K}_1, \ldots, \mathtt{K}_\mathtt{m}$ *and* $\mathtt{P}, \mathtt{Q}$ *be as in Proposition 9*

Then the reply of the service MCP *(as defined above) to the first method call in the computation* $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ *equals* `false` *if and only if the following condition is satisfied:*

for some positive $\mathtt{k} \in \mathbb{N}$*, the computation* $\mathtt{Q} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ *performs the method call* $\mathtt{g}_\mathtt{i}$.do:no_harm *in the* k*'th step while* $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ *does not reach an occurrence of the method call* $\mathtt{g}_\mathtt{i}$.do:no_harm *in* l *steps for some* $\mathtt{l} \leq \mathtt{k}$.

Proof. This proof is a straightforward though somewhat tedious consequence of Proposition 9. Suppose that the reply of MCP to the first method call in the computation $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ equals `false`. Then in view of the definition of MCP, there exists a function β such that the computation $\mathtt{Q} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\beta) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ performs $\mathtt{g}_\mathtt{i}$.do:no_harm. With Proposition 9, it follows that the computation $\mathtt{Q} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ performs $\mathtt{g}_\mathtt{i}$.do:no_harm, say, at the k-th step. Following this computation, one finds that for some function α, it must be the case that $\mathtt{Q} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\alpha) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ performs $\mathtt{g}_\mathtt{i}$.do:no_harm in the k-th step, moreover for function α', the computation $\mathtt{Q} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\alpha') \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ performs $\mathtt{g}_\mathtt{i}$.do:no_harm within fewer than k steps. Now contemplating the definition of MCP, one notices that its condition ψ is not satisfied, whence its condition χ must be satisfied, from which it follows that for no function γ, it is the case that $\mathtt{P} \bullet \mathtt{h.RFS}(\{\mathtt{find}\},\gamma) \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ performs $\mathtt{g}_\mathtt{i}$.do:no_harm within k or fewer steps, from which it follows by Proposition 9 that $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ does not perform a method call within k steps or less. This proves one direction of the Theorem; the argument in the other direction is quite similar, and is deleted for that reason. □

Theorem 4. *With the same assumptions as in Propositions 9 and 10: for all threads* $\mathtt{P}, \mathtt{Q}$ *with required interface included in* $\mathtt{J} = \{\mathtt{h.find}\} \cup \mathtt{g}_1.\mathtt{I}_{\mathtt{K}_1} \cup \cdots \cup \mathtt{g}_\mathtt{m}.\mathtt{I}_{\mathtt{K}_\mathtt{m}}$*, the following implication holds: if* $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ *performs a basic instruction* $\mathtt{g}_\mathtt{i}$.do:no_harm *or* $\mathtt{Q} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ *performs a basic instruction* $\mathtt{g}_\mathtt{i}$.do:no_harm *then also* $(\mathtt{P} \trianglelefteq \mathtt{h.find} \trianglerighteq \mathtt{Q}) \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ *performs a basic instruction* $\mathtt{g}_\mathtt{i}$.do:no_harm.

Proof. This result follows from Proposition 10 plus the observation that if $\mathtt{P} \bullet \mathtt{h.MCP} \oplus \mathtt{g}_1.\mathtt{K}_1 \oplus \cdots \oplus \mathtt{g}_\mathtt{m}.\mathtt{K}_\mathtt{m}$ performs a basic instruction $\mathtt{g}_\mathtt{i}$.do:no_harm, then for some Boolean function β also $\mathtt{P} \bullet \mathtt{h.MCP}_\beta \oplus g_1.K_1 \oplus \cdots \oplus g_m.K_m$ performs a basic instruction $\mathtt{g}_\mathtt{i}$.do:no_harm. □

Problem 3. *Consider architerms containing both a* SHRAT *service and an* MCP *service. Can Theorem 4 be generalised to this case?*

5. Lookahead Instructions

Threads provide abstractions from sequential programs. The impossibility result of Cohen on virus detection was formulated in terms of programs rather than in terms of threads and services. In this section, we will consider how the results on lookahead services may be used for obtaining information at the lower abstraction level of instruction sequences.

Special instructions may be designed, which incorporate the support of lookahead services into instruction (sequence) processing without making explicit calls to a lookahead service. Leaving apart the task of designing a syntax for such instructions in general, it is already informative to consider some simple special cases.

With a§f.m, for a a positive natural number, we will denote the following condition which may be evaluated during a run at some position (instruction) in an instruction sequence. Let $\mathtt{X};\mathtt{Y}$ be an instruction sequence such that $\mathtt{LLOC}(\mathtt{X}) = \mathtt{b} - 1$. Now a§f.m, when evaluated at position b asserts that a computation of $\mathtt{X};\mathtt{Y}$ when stated at the beginning of Y will proceed in such a manner that the a'th step is a basic action involving a method call f.m, i.e., either the void method call or a test $+$f.m or a test $-$f.m. If the computation has terminated, either properly or improperly, at or before the a'th step, the value of a a§f.m is considered `false`.

Making use of a lookahead condition is not an obvious matter. The simplest idea is to use the condition as a test which is evaluated at the position where it occurs. However, in $\mathtt{X}_1 = \#1;\mathtt{g.m1};-(2\S\mathtt{f.m});\mathtt{f.m};!$, a difficulty arises. Assuming that at position 3, the condition 2§f.m takes value `true`, then upon starting the run in position 3, the result of the test is positive and because of its embedding in a negative test instruction, the instruction is performed as if it were #2 so that as a second step in the run, termination (!) occurs, which implies that the evaluation of 2§f.m at position 3 should have taken value `false` in hindsight. Alternatively one may consider the possibility that evaluation of the condition 2§f.m at position 3 yields value `false`, and then the third instruction is

performed as if it were #1 so that the second step is f.m so that, again in hindsight, 2§f.m (evaluated at position 3) should have yielded value true. The instruction $-$(2§f.m) is, in this context, paradoxical because no evaluation of its embedded condition is possible. In that case, a run of X_1 involving the putting into effect of the third instruction is said to terminate improperly (just like #0).

By first jumping to another position, either forward or backward, the mentioned paradox may be avoided, though not fully. With #b§a§f.m, the condition is denoted which, when evaluated at position p amounts to evaluating a§f.m at position p + b. If that position lies outside the range of instructions, the result is false. Similarly the condition \#b§a§f.m is evaluated by evaluating a§f.m at position p $-$ b, again returning false if that position is 0 or lower. Consider X_2 = #1; g.m1; $-$(#1§2§f.m); +g.m2; f.m; #0; ! When evaluating #1§2§f.m at position 3, the hypothetical computation has to start at position 4, then as the first step, the test +g.m2 is performed, and then depending on the result either the call f.m is made (so that the evaluation of #1§2§f.m yields true) or improper termination results (so that the evaluation of #1§2§f.m yields false).

A paradoxical situation may result if the computation started at position p + b for an evaluation of #b§a§f.m at position p visits the test instruction (say +(#b§a§f.m)) within k steps. In that case, it is assumed that no definite value can be found for the condition #b§a§f.m and that the execution of the instruction +(#b§a§f.m) at position p runs into improper termination. With the understanding that, say $-$(2§f.m) is an abbreviation of $-$(#0§2§f.m), it is plausible to adopt the convention that $-$(2§f.m) leads to improper termination immediately, because the test is called before an evaluation of the embedded condition is known.

For the (hypothetical) processing of a condition #b§a§f.m, say, occurring as an embedded instruction in a test +(#b§a§f.m) at position p in an instruction sequence X, we notice the following:

(i) It does not change the state of any service (because it is a hypothetical form of processing);

(ii) It returns true if, when (hypothetically) the computation is performed, that computation will perform a basic action f.m in the a-th step (without any visit to the p'th instruction in between);

(iii) It returns false otherwise (including the cases that proper or improper termination arises before the *a*'th step of the computation).

Implementing a Lookahead Instruction via a Prospective Service

A useful modification of the lookahead instructions +(#b§a§f.m) and $-$(#b§a§f.m) is as follows: +(#b§F§f.m) and $-$(#b§F§f.m) include tests which require that at least once during a run, a basic action or test based on the method call f.m is performed.

The positive result in Theorem 1 can be used for clarifying the feasibility of a specific lookahead instruction. Here, the use of infinite instruction sequences, obtained via repetition of a finite instruction sequence, comes in handy. As was shown in [2], repetition, which underlies the program algebra PGA, has the expressive power of arbitrary flow charts though without making use of backward jumps.

Proposition 11. *Let* $U \equiv -(\#1§F§g_i.do{:}harm)$. *Let* X *be an instruction sequence with zero or more occurrences of* U *as its only lookahead instructions and without backward jumps and with required interface contained in* $J = I_{provided}(g_1.K_1 \oplus \cdots \oplus g_n.K_n)$. *Let* X′ *be obtained from* X *by replacing each occurrence of* U *by the method call* +f.ok.

Then, upon defining $|X| \bullet g_1.K_1 \oplus \cdots \oplus g_n.K_n$ *as* $|X'| \bullet f.SHRAT \oplus g_1.K_1 \oplus \cdots \oplus g_n.K_n$, *the semantics of instruction* U *is given in such a way that it complies with the (informal) definition given above for the (lookahead) test instruction* $-(\#1§F§g_i.do{:}harm)$.

Proof. When using repetition as an instruction sequence constructor, an infinite, repeating, instruction sequence results in which only forward jumps are used. Using this fact, the correspondence between the semantics of instruction U and its counterpart after thread abstraction is perfect. □

6. Concluding Remarks

Several aspects of this paper suggest further work. The notion of a lookahead condition merits more scrutiny. Perhaps we have not yet been able to prove anything positive about cases where different prospecting services are simultaneously present. As intelligent agents will try to forecast one-another's behaviour, the idea of different forecasting mechanisms acting in parallel may be considered intuitively obvious, while the mathematics of that situation is not at all clear.

6.1. Multi-Agent Aspects

Suppose $\mathtt{P}_1,\ldots,\mathtt{P}_n$ are threads and $\mathtt{W}_i$ are service families such that (i) the different focus interfaces ($\mathtt{I}_{\mathtt{focus}}(\mathtt{W}_i)$) are pairwise disjoint and (ii) $\mathtt{I}_{\mathtt{required}}(\mathtt{P}_i) \subseteq \mathtt{I}_{\mathtt{provided}}(\mathtt{W}_i)$. We will assume that the threads constitute a so-called thread vector, so that the ordering matters as well. Following [1], we will write this vector as follows: $\langle \mathtt{P}_1\rangle^\frown \ldots^\frown \langle \mathtt{P}_n\rangle$. Then strategic interleaving of the thread vector is denoted with

$$||_c(\langle \mathtt{P}_1\rangle^\frown \ldots^\frown \langle \mathtt{P}_n\rangle)$$

In [1], a range of different strategic interleaving operations are provided. The idea behind strategic interleaving operators is to model multi-threading in the way it is implemented in programming systems which arrange for deterministic scheduling rather than for some form of the arbitrary interleaving of concurrent threads. The advantage of modelling multi-threading with strategic interleaving is that arguably, that is more realistic. A disadvantage, however, is that the very well understood parallel composition operator based on arbitrary interleaving, which stands at the basis of most process algebras, is replaced by a range of different strategic interleaving operators of increasing complexity.

For cyclic interleaving and variations thereof (as discussed in detail in [1]), the required interface of a thread vector is found by taking the union for the individual threads: $\mathtt{I}_{\mathtt{required}}(||_c(\langle \mathtt{P}_1\rangle^\frown \ldots^\frown \langle \mathtt{P}_n\rangle)) = \cup_{i=1}^n \mathtt{I}_{\mathtt{required}}(\mathtt{P}_i)$. Because $\mathtt{I}_{\mathtt{provided}}(\mathtt{W}_1 \oplus \ldots \oplus \mathtt{W}_n) = \cup_{i=1}^n \mathtt{I}_{\mathtt{provided}}(\mathtt{W}_i)$ the architerm

$$||_c(\langle \mathtt{P}_1\rangle^\frown \ldots^\frown \langle \mathtt{P}_n\rangle) \bullet \mathtt{W}_1 \oplus \ldots \oplus \mathtt{W}_n$$

makes perfect sense. It follows from Proposition 5 that if, say, $\mathtt{W}_1$ contains $\mathtt{f.SHRAT}$ and this instance of $\mathtt{SHRAT}$ attempts prospection on a basic action, say, $\mathtt{g.m}$ with $\mathtt{g} \in \mathtt{I}_{\mathtt{focus}}(\mathtt{W}_2)$ that this will allow $\mathtt{P}_1$ to use foresight to the extent that it may only perform certain basic actions in the case that $\mathtt{P}_1$ "knows" that $\mathtt{P}_2$ will not perform $\mathtt{g.m}$ in the future. It follows from Theorem 2 that the situation becomes much more complicated if different agents try to make use of prospecting services at the same time. On the other hand, expecting that $\mathtt{P}_1$ may be equally certain about guarantees that $\mathtt{P}_2$ will actually perform $\mathtt{g.m}$ in the future may be asking too much in view of Proposition 8.

Collective action and underlying collective intelligence rests upon knowledge that agents have or pretend to have about other agents in the same flock. The results of the paper shed some light on the options and limitations of such knowledge.

6.2. Why Thread Algebra, in the Context of Prospection and Foresight?

Viewed from the perspective of process algebra working with a domain-specific process algebra is an overhead which one might preferably avoid.

The simplest idea on foresight in process algebra is to allow a condition $\mathtt{noACT}(\mathtt{a}_\mathtt{g}(\mathtt{d}))$ which is true if and only if an atomic action $\mathtt{a}_\mathtt{g}(\mathtt{d})$ (i.e., atomic action of type $\mathtt{a}$ at location/port $\mathtt{g}$, with parameter/data, $\mathtt{d}$) will not be performed in the future of the system (process at hand). One may look for a counterpart of Proposition 5 in this case, but there is a problem with that idea.

Indeed consider $\mathtt{P} = \mathtt{a}_\mathtt{g}(\mathtt{d}) \lhd \mathtt{noACT}(\mathtt{b}_\mathtt{g}(\mathtt{d})) \rhd \mathtt{a}_\mathtt{g}(\mathtt{d})$ with $\mathtt{a} \neq \mathtt{b}$. $\mathtt{P}$ is paradoxical in the sense that the suggested foresight cannot exist. This very issue of foresight preventing self-reference is, at least to some degree, avoided by working with thread algebra.

6.3. Potential Connections with Classical AI

While prospecting, services which allow the realisation of attractive foresight patterns may be impossible or hypothetical. Reworking the concepts at hand in such a manner that prospecting and foresight can be replaced by computable approximations thereof may be a workable approach. Practical prospecting, especially when based on empirical data may be termed lookahead; practical foresight, again when performed on a scientifically defensible basis, may be termed forecasting. An option for further work along the lines of this paper is to adapt software engineering life-cycle models in two stages: first to make use of the admittedly hypothetical advantage of prospecting services and foresight patterns realised with the help of such services, and thereafter to work with approximations of the various instances of prospecting and foresight, thereby obtaining computable (non-hypothetical) lookahead services and forecasting patterns.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Bergstra, J.A.; Middelburg, C.A. Thread algebra for strategic interleaving. *Form. Asp. Comput.* **2007**, *19*, 445–474. [CrossRef]
2. Bergstra, J.A.; Loots, M.E. Program algebra for sequential code. *J. Log. Algebr. Program.* **2002**, *51*, 125–156. [CrossRef]
3. Bergstra, J.A.; Middelburg, C.A. A short introduction to program algebra with instructions for Boolean registers. *Comput. Sci. J. Mold.* **2019**, *26*, 199–232. Available online: http://www.math.md/files/csjm/v26-n3/v26-n3-(pp199-232).pdf (accessed on 12 May 2022).
4. Bergstra, J.A.; Middelburg, C.A. Instruction sequence processing operators. *Acta Inform.* **2012**, *49*, 139–172. [CrossRef]
5. Bergstra, J.A.; Middelburg, C.A. Distributed strategic interleaving with load balancing. *Future Gener. Comput. Syst.* **2008**, *120*, 530–548. [CrossRef]
6. Bergstra, J.A.; Middelburg, C.A. Thread algebra for poly-threading. *Form. Asp. Comput.* **2011**, *23*, 567–583. [CrossRef]
7. Middelburg, C.A. Progam algebra for random access machines. *arXiv* **2020**, arXiv:2007.09946.
8. Bergstra, J.A.; Klop, J.W. Process algebra for synchronous communication. *Inf. Control* **1984**, *60*, 109–137. [CrossRef]
9. Bergstra, J.A.; Middelburg, C.A. On the behaviours produced by instruction sequences under execution. *Fundam. Informaticae* **2012**, *24*, 111–144. [CrossRef]
10. Bergstra, J.A.; Klop, J.W. Verification of an alternating bit protocol by means of process algebra. In *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence*; Springer: Berlin/Heidelberg, Germany, 1986; pp. 9–23. [CrossRef]
11. Baeten, J.C.M.; Middelburg, C.A. *Process Algebra with Timing*; Monographs in Theoretical Computer Science; Springer: Berlin/Heidelberg, Germany, 2002. [CrossRef]
12. Bergstra, J.A.; Ponse, A. Execution architectures for program algebra. *J. Appl. Log.* **2004**, *5*, 170–192. [CrossRef]
13. Middelburg, C.A. Probabilistic process algebra and strategic interleaving. *Sci. Ann. Comput. Sci.* **2020**, *30*, 205–243. [CrossRef]
14. Groote, J.F.; Ponse, A. The syntax and semantics of μCRL. In *Algebra of Communicating Processes*; Springer: London, UK, 1994; pp. 26–62.
15. Groote, J.F.; Mathijssen, A.; Reniers, M.A.; Usenko, Y.S.; van Weerdenburg, M.J. Analysis of distributed systems with mCRL2. In *Process Algebra for Parallel and Distributed Processing*; Chapman & Hall: Boca Raton, FL, USA, 2009; pp. 99–128.
16. Baeten, J.C.M.; Bergstra, J.A. Global renaming operators in concrete process algebra. *Inf. Comput.* **1988**, *78*, 205–245. [CrossRef]
17. Baeten, J.C.M.; Bergstra, J.A.; Klop, J.W. Conditional Axioms and α/β- Calculus in Process Algebra; In *Formal Description of Programming Concepts III, Proceedings of IFIP TC 2/WG 2.2 Working Conference, Ebberup, Denmark, 25–28 August 1986*; North Holland Publishing Company: Amsterdam, The Netherlands, 2017; pp. 53–74.
18. Cohen, F. Computer viruses—Theory and experiments. *Comput. Secur.* **1984**, *6*, 22–35. [CrossRef]
19. Evans, D. *On the Impossibility of Virus Detection*; Department of Computer Science, University of Virginia: Charlottesville, VA, USA, 2017. Available online: https://www.cs.virginia.edu/~evans/virus/ (accessed on 12 May 2022).
20. Selçuk, A.A.; Orhan, F.; Batur, B. Undecidable problems in malware analysis. In Proceedings of the 12th International Conference for Internet Technology and Secured Transactions (ICITST), Cambridge, UK, 11–14 December 2017. [CrossRef]
21. Laski, J. Programming faults and errors: Towards a theory of software incorrectness. *Ann. Softw. Eng.* **1997**, *4*, 79–114. [CrossRef]
22. Mili, A.; Frias, M.F.; Jaoua, A. *On Faults and Faulty Programs*; Höfner, P., Jipsen, P., Kahl, W., Müller, M.E., Eds.; RAMiCS 2014, LNCS 8428 Series; Springer: Cham, Switzerland, 2014; pp. 191–207. [CrossRef]
23. Bergstra, J.A. Instruction sequence faults with formal change justification. *Sci. Ann. Comput. Sci.* **2020**, *30*, 105–166. [CrossRef]
24. Bergstra, J.A. Promises in the context of humanoid robot morality. *Int. J. Robot. Eng.* **2020**, *5*, 20.
25. Bergstra, J.A.; Middelburg, C.A. Data linkage dynamics with shedding. *Fundam. Informaticae* **2010**, *103*, 31–52. [CrossRef]
26. Bergstra, J.A. Quantitative expressiveness of instruction sequence classes for computation on bit registers. *Comput. Sci. J. Mold.* **2019**, *27*, 131–161. Available online: http://www.math.md/publications/csjm/issues/v27-n2/12969/ (accessed on 12 May 2022).
27. Raclet, J.-B.; Badoel, E.; Benveniste, A.; Callaud, B.; Passerone, R. *Why Are Modalities Good for Interface Theories?* Report No. 6899; INRIA: Rocquencourt, France, 2009; ISSN 0249-6399. Available online: https://hal.inria.fr/inria-00375098 (accessed on 12 May 2022).
28. de Alfaro, L.; Henzinger, T.A. Interface Automata. ACM SIGSOFT Software Engineering Notes. 2001. Available online: https://luca.dealfaro.com/papers/01/FSE01.pdf (accessed on 12 May 2022).
29. Bergstra, J.A.; Ponse, A. A bypass of Cohen's impossibility result. In *Advances in Grid Computing 2005, Proceedings of the European Grid Conference, Amsterdam, The Netherlands, 14–16 February 2005*; LNCS 3470 Series; Springer: Berlin, Germany, 2005; pp. 1097–1106. Available online: https://staff.fnwi.uva.nl/a.ponse/publist/EGCreport.pdf (accessed on 12 May 2022).
30. Christodorescu, M.; Jha, S.; Seshia, S.A.; Song, D.; Bryant, R.E. Semantics-Aware Malware Detection. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05), Oakland, CA, USA, 8–11 May 2005. [CrossRef]
31. Bergstra, J.A.; Bethke, I.; Ponse, A. Thread algebra and risk assessment services. *Camb. Lect. Notes Log.* **2007**, *28*, 1–17. Available online: https://staff.fnwi.uva.nl/a.ponse/publist/Bergstra.pdf (accessed on 12 May 2022).