AD-A043 492 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
AN INSTRUCTABLE PRODUCTION SYSTEM: BASIC DESIGN ISSUES.(U)
MAY 77 M D RYCHENER, A NEWELL F44620-73-C-0074

UNCLASSIFIED AFOSR-TR-77-1144 NL

1 OF 1
AD A043492

END
DATE FILMED
9-77
DDC

AFOSR-TR- 77-1144

AD A043492

An Instructable Production System:

Basic Design Issues

Michael D. Rychener and Allen Newell
May 1977

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Approved for public release;
distribution unlimited.

This paper has been submitted for presentation at a Workshop on Pattern-Directed Inference Systems, chaired by D. A. Waterman and F. Hayes-Roth, Honolulu, May 23-27, 1977.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

AD No. ____
DDC FILE COPY

DDC
RECEIVED
AUG 29 1977
B

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (18) AFOSR-TR-77-1144 (19)	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) (6) AN INSTRUCTABLE PRODUCTION SYSTEM: BASIC DESIGN ISSUES.	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Michael D. Rychener [redacted] Allen Newell	8. CONTRACT OR GRANT NUMBER(s) (15) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (16) 61102F 2304 A2 (17) A2
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE (11) May 1977
	13. NUMBER OF PAGES ~~31~~ (12) 34 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution.

(9) Technical rept.

see inside cover

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The full advantages of the incremental properties of production systems have yet to be exploited on a large scale. A promising vehicle for this is the task of instructing a system to solve problems in a complex domain. For this, it is important to express the instruction in a language similar to natural language and without detailed knowledge of the inner structure of the system. Instruction and close interaction with the system as it behaves are preferred over a longer feedback loop with more independent learning by the system. The domain is initially an abstract job shop. The beginning system has capabilities for solving problems,

(continued)

next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403081

20. abstract (continued)

processing language building productions, and interacting with the task environment. All parts of the system are subject to instruction. The main problem-solving strategy, which permeates all four system components, is based on means-ends analysis and goal-subgoal search. This is coupled with an explicit representation of control knowledge. The system's behavior so far is restricted to simple environmental manipulations, a number of which must be taught before more complex tasks can be done.

Table of Contents

SECTION			PAGE
1	Instruction Tasks and Large Production Systems		1
	1.1	Introduction and overview	2
	1.2	Building a large production system	3
	1.3	The abstract job shop task	4
	1.4	The instruction mode	6
	1.5	The production system architecture and task environment	8
2	The Initial Instructable System		12
	2.1	The problem-solving component of the Kernel	13
	2.2	External language capabilities of the Kernel	16
	2.3	Building productions and the interface to the TE	18
	2.4	Discussion of the Kernel design	19
3	Sample System Behavior		19
4	Conclusions		23
5	References		27
	5.1	Footnotes	28

ACCESSION for
NTIS White Section X
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

Hawaii

An Instructable Production System: Basic Design Issues #1.

Michael D. Rychener and Allen Newell

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract. The full advantages of the incremental properties of production systems have yet to be exploited on a large scale. A promising vehicle for this is the task of instructing a system to solve problems in a complex domain. For this, it is important to express the instruction in a language similar to natural language and without detailed knowledge of the inner structure of the system. Instruction and close interaction with the system as it behaves are preferred over a longer feedback loop with more independent learning by the system. The domain is initially an abstract job shop. The beginning system has capabilities for solving problems, processing language building productions, and interacting with the task environment. All parts of the system are subject to instruction. The main problem-solving strategy, which permeates all four system components, is based on means-ends analysis and goal-subgoal search. This is coupled with an explicit representation of control knowledge. The system's behavior so far is restricted to simple environmental manipulations, a number of which must be taught before more complex tasks can be done.

1. Instruction Tasks and Large Production Systems

1.1. Introduction and overview

This paper reports the beginnings of a system-building project. The aim is to build a large, generally intelligent system by gradual instruction starting from a small initial system. At present, the large system is still in the future. This description is limited to describing a promising initial system, along with the rationale for believing it has significant potential for further work. Likewise, the task domain of the eventual system is not yet determined, though there is an initial domain.

Production systems are the system architecture most consonant with the project's aims. Their basic condition-action form, along with the global and open nature of their action, indicate their usefulness for a task involving incremental growth, recognition-based problem solving, responsiveness to unexpected conditions, and other attributes discussed below. The initial task domain is based on the problem of scheduling a job shop. This has unusual features that allow tests of basic instruction issues, particularly a wide range of tasks with simply-produced variants. Thus the potential exists for instructing the system on one variant, and then introducing perturbations to which it must dynamically adjust, evoking the need for further instruction. There are several constraints on what instruction is and on what the instructors can know about the internal content of the system.

The remainder of this section discusses in more detail the basic task and system issues, and introduces a suitable production system architecture. Section 2 discusses the rationale for our approach to building the initial system, called the Kernel. The Kernel embodies a set of assumptions about problem solving, language use, the particular task, and augmentation. Section 3 presents an instruction protocol that the system has performed. Section 4 summarizes our current status.

1.2. Building a large production system

Production systems (abbreviated "PSs") have a brief but illustrious history within artificial intelligence (AI) and cognitive psychology. For general background, the reader is referred to [5], [10] and [12]. There are four architectural components of the kind of PS used here: production memory, working memory, recognize-act cycle, and conflict resolution principles. Action arises from the system as a result of conditions (left-hand sides) of productions being recognized true of the current working memory state. The recognition leads to the performance of associated actions (right-hand sides). This is the basic recognize-act step, except that in general the conflict resolution principles must be applied to distinguish between productions whose conditions are simultaneously true, making a selection before actions are actually performed. The performance of actions results in a new working memory state, and the recognize-act step is repeated.

We have chosen PSs for our instructable system for a number of reasons. All productions are sensitive to a single working memory, with no control organization imposed on them, and with all necessary control achieved by goals and other data conventions within working memory. In practice, productions tend to be small (only a few conditions and actions) and relatively independent of each other. Thus they are attractive where structure is to be added gradually and incrementally. Their feasibility, power, transparency, flexibility, and conciseness has been shown empirically by implementing well-understood AI systems [13]. The importance of having actions conditioned on the recognition of aspects of a global state is central. Actions are not evoked directly by other actions, but are performed whenever the appropriate conditions emerge. Thus intelligence is distributed rather than concentrated in a complex control executive or other orchestrating mechanism. Since intelligence requires the ability to respond to important

aspects of complex states, the high degree of conditionality of action in PSs appears to have merit.

In building the system, the PS architecture is used according to specific conventions. All long-term knowledge is kept as productions, and working memory is used exclusively for short-term, dynamic state. This is in contrast to a possible view of working memory as a long-term database, with "facts" stored in it, to be manipulated by "procedures" coded as productions. Though working memory may become large, our convention is to store as productions such database-like things as facts about objects in the world, relational structures (semantic networks), etc.

A large intelligent system of the sort envisioned places new demands on PSs and on system-building capabilities in general. Building such systems is interesting in its own right, raising issues of representation accommodation, and a whole range of activities associated with intelligence [9, see also 1]. To study many of these issues fruitfully, as many have noted, an uncommonly large-scale knowledge base is necessary. From a pure PS research standpoint, building a large system (on the order of several thousand productions), especially including a rich diversity of knowledge, allows us to test hypotheses about PS efficiency openness, modularity, automatic augmentation, representational flexibility, and feasibility. The system is to develop, eventually, past the current state of the art in AI.

1.3. The abstract job shop task

Several criteria are essential to our choice of an initial task for an instructable system. The task domain should be rich in problems of sufficient challenge to require instruction it should be amenable to the instructional mode (see Section 1.4); and it should not be amenable to a general solution mechanism, which, once constructed, would

make further instruction unnecessary. Among the general task areas that might be appropriate are: a tutor in some domain, an intelligence-test taker, an automatic programming system, and the higher, cognitive levels of an image understanding system. We have chosen for the time being a toy task, the abstract job shop (AJS). The job shop has as its objective to produce objects with specified desired properties from raw materials according to some schedule. The shop contains stacks of materials and partial products, machines that must be started with explicit commands, and means for transporting objects from one place to another within the shop. The details of the particular implementation of this idea are given in Section 1.5. This toy task has a number of close analogs that are potential applications of any useful techniques developed: real-world production scheduling, the general problem of functional design, scheduling in computer operating systems, and coding computer programs (to name just a few). It also contains within it the possibility of exploring the full range of AI tasks known as the "toy blocks world."

If all goes well, the AJS task has attributes that are the extra bonus for immediate purposes. AJS has an unusual number of variants, including the basic task of producing desired objects, the allocation of scarce resources, advanced kinds of planning, and production under time constraints. After the system has been instructed in a number of basic variants, perturbations to the tasks and to the environment (the job shop) can pose major difficulties for the system. Among these perturbations are: changes in the profit-objective function for various mixes of outputs, spoilage of materials, error in machines, accidents in moving objects, additional time constraints and last-minute changes in orders. The difficulties of the basic task should preclude any advance planning on the part of the instructor to have the system respond gracefully to such basic task changes. Thus, the

system's behavior will be interesting, whether it can adjust easily or not. The effectiveness of the entire approach, including the use of PSs, will be measured by the adequacy of the system's behavior over a set of such perturbations.

1.4. The instruction mode

Posing the task as one of growing a large system through instruction introduces additional issues. Some of these might seem irrelevant to the main aims, but others are directed towards important questions with respect to the study of the representation and use of knowledge. The following attempts to justify this third major concern, instruction, which is in addition to the concerns with building a generally intelligent system and using PSs as a basis.

The instruction mode used here forces the automatic encoding of knowledge as productions. This allows the verification of essential properties of PSs, particularly those dealing with the independence or modularity of the knowledge in the PS. If the PS were augmented by simply composing and adding Ps, there would still be a possibility for the system to be very intricately contrived, with implicit global coordination of production action sequences. A language of instruction is used that states each new item of knowledge in a human-readable, plausibly independent form, with no reference to internal structure.

Instruction takes place under the following constraints:

1. The instructor (Ins) can see what the system (IPS, Instructable PS) is doing in the environment, and can communicate with IPS, but cannot examine the internal structure of the system directly.
2. Interaction between Ins and IPS is in an external language, analogous to natural language, rather than in internal representations, either of working memory or production memory forms.

3. The initiative for interaction is mixed. IPS's behavior can be interrupted by Ins at any time, for corrective instruction or interrogation. Likewise, IPS may communicate to Ins and interrupt him.
4. Instruction may be about any topic within the total environment: the structure of the environment, how to perform a task, the language of communication, the detection and correction of errors, how to learn about the environment, etc. Also, the instruction may be at whatever level Ins wishes or can achieve: specific behavior sequences, general methods, abstract principles, models, theories, etc.
5. Knowledge gained through instruction accumulates over the life of the system.

Having the system be instructable adds to its capabilities as a total man-computer interactive system, so that in ultimate real applications the performance of the combination system can be expected to be higher than either participant alone. As a practical measure, making the system instructable also reduces the possibility of internal coding conventions that would prohibit multiple instructors from understanding the existing system. That is, all communication is forced to be in a language of instruction, which may be more easily shared than program conventions. If instructability can be achieved, it should be worth the extra initial effort.

The instruction mode can be contrasted with a learning mode in which the system is set tasks and then required to learn on its own from the environment. Here Ins gives incomplete or approximate instructions and watches very closely for opportunities to interrupt IPS and refine them. It is "incomplete or approximate instructions" because too much pre-planning by Ins is bound to be futile, given Ins's imperfect knowledge of IPS's internal structure, and given a task sufficiently complex to make anticipation difficult and

ineffective. With Ins watching IPS so closely, the need for learning by the system on its own is minimized. But such independent learning is not excluded. It can eventually arise in the way IPS interacts with and gathers knowledge from the task environment, in the way IPS uses the external language, and in other knowledge acquisition mechanisms. Presumably the best strategies of instruction and performance require that IPS be able to learn for itself about a changing environment. The sequence of novel but related tasks is intended in part to arouse this. Nor does the futility of pre-planning rule out giving IPS general methods, anticipating certain types of difficulties. Such general methods, however, are bound to have incompleteness similar to that of specific ones.

1.5. The production system architecture and task environment

Before detailing the PS architecture used for IPS, a few distinguishing features of our overall PS approach are pointed out. The way that action develops from the PS differs from some others in being a forward recognition-driven cycle, rather than a backward-chaining, goal-driven cycle, as in the MYCIN system [4]. The system is controlled by signals and symbol structures in the global working memory, called goals, which are included explicitly in production conditions when appropriate. This is in contrast to MYCIN and to DENDRAL [3]. The PS architecture is used as the total system, rather than having it be one of a number of procedural components. Other systems have employed additional, non-PS procedures for such activities as modifying and analyzing the PS. Working memory is arbitrary list structures in an extensive database-like structure, with a vast majority of items explicitly stored rather than represented as computable predicates. Production conditions make use of general pattern-matching capabilities, as is common in other recent AI languages [2]. Though the general architecture derives from concern for human cognition [10, 12], little consideration is given to psychological constraints.

The particular architecture and language used for IPS is called OPS (Official, at least locally, PS) [7] and is an iteration on earlier designs [11, 13]. Production memory in OPS is an unstructured, unordered set of productions. Working memory is likewise an unordered set of list structures, without duplications. It is bounded in size, by deleting elements whose last assertion occurred more than some arbitrary number of system actions in the past (currently 300). The recognize-act cycle is: (1) form a conflict set of productions whose conditions are currently satisfied; (2) apply the conflict resolution principles to select a unique element from the conflict set; and (3) execute the actions of the selected production.

For conflict resolution (the most distinctive component of OPS) the following rules apply, in order. These rules are experimental in nature, and are expected to change as understanding of instructability increases [8].

1. Refraction: a production is not fired twice on the same data (instantiation of a pattern) unless some part of that data has been re-inserted into working memory since the previous firing. This prevents most infinite loops and other useless repetitions.
2. Lexicographic recency: the production using the most recently inserted elements of working memory is preferred. "Most recent" is determined lexicographically, i.e., if there is a tie on the most recent element used, the next-most recent elements are compared, and so on; use of any element is considered more recent than using none, e.g., (A X) is ordered before (A). Recency order discriminates at the level of individual actions within productions, rather than taking all the actions performed by a production to be of equal recency. This rule serves to focus the attention of the system very strongly on more recent events, allowing current goals to go to completion before losing control.

3. Special case: a production is preferred that has more conditions, including negative conditions which do not match to specific memory elements. Most of the meaning of having one production be a special case of another is captured by rule 2, since a special case that uses more data than a general one is lexicographically more recent. Preferring special cases to general ones follows the expectation that a specific method is more appropriate to a situation than a more general one. Also, this is consonant with a strategy of augmentation by providing more discriminative rules.
4. Production recency: the more recently created production is preferred. This allows identically conditioned rules (with perhaps contradictory actions) to be distinguished and assumes that a more recent instruction is more correct.
5. Arbitrary: a selection is made among multiple matches to the same production using the same data.

As a matter of practice, conflict resolution rarely requires more than the first two rules.

OPS has several other distinguishing features. The pattern matching allows a limited form of segment variables namely, a variable may match an indefinite-sized tail of a list. The Pattern-And (Pand) feature allows an expression to be matched to several patterns, and then bound to a variable. OPS allows complex negative conditions to be specified, for instance, including the negation of an entire production condition within the condition of another production. Productions in OPS are compiled into an efficient network form, rather than interpreted [6]. OPS has an operator for adding productions to production memory which have been formed (in terms of an appropriate data structure) in working memory; such additions are done directly into the compiled network during the runtime cycle without excessive cost.

A subsystem of OPS provides the task environment (TE) for instruction. The TE is represented as an array of discrete locations, within which objects can be placed, plus a set of "perceptual" and "motor" operators. Each location and object is represented as a list of pairs in attribute-value form, with certain attributes given special interpretations. For example, the external display of a location in the TE (L15), would be

```
L15     |-------|
        |W6     |
        |W98    |
        |W72    |
        |       |
        |       |
        |-------|
```

with the internal representation,

```
L15:    ( NAME          L15
          TYPE          STACK
          MEMBER        TE-ARRAY
          POSITION      (2 3)
          COMPOSITION   (W6 W98 W72) )
```

The object W98 might be defined as:

```
W98:    ( NAME          W98
          TYPE          WOOD
          MEMBER        L15
          POSITION      (2)
          SHAPE         TRIANGULAR
          LENGTH        5
          WIDTH         7
          COLOR         RED )
```

Objects are potentially hierarchical, with values of attributes composed of other objects. Relations between positions of the TE and objects can be determined by TE operators.

The operators on the TE are:

1. View: the attribute-value pairs for an object or location appear in working memory.

2. Scan: the TE is searched for an object satisfying a pattern, and if it is found, it is Viewed.
3. Trans: an object is transferred from one location to another within the TE.
4. Start: a machine in the TE is started, consuming a set of inputs (specified as values of INPUT attribute) and producing a set of outputs (specified as values of OUTPUT). The machine operates once, not continuously.
5. Compare: two attribute-value pairs are compared, with results depending on the values compared. For instance, if the values are pairs of numbers, as for POSITION, the result is a spatial relation, amounting to, say, "northwest."

2. The Initial Instructable System

The instructable system is initialized with a relatively small set of hand-coded productions called the Kernel. The Kernel design includes a minimal set of components that can support all of the present instruction goals and provide an interface to the TE. The components at present achieve minimal capabilities for: (1) solving problems, (2) processing language, (3) building productions, and (4) interacting with the TE.

A number of design issues influenced the Kernel. These derive from a wish to maintain easy instructability within the rules laid out above for the instructional mode.

1. Everything in the system is potentially instructable and improvable. This includes especially the components of the Kernel and the results of instruction that the Kernel produces. The Kernel itself may eventually be superseded by productions gained through instruction, and commitments to techniques and representations in the Kernel may eventually be altered.

2. The system should be instructable without detailed knowledge of internal structures. Thus the Kernel design must include some capability for mapping from external to internal forms, and vice versa.
3. Knowledge should not be globally coordinated or pre-planned, but should develop in locally plausible, concrete increments. This particularly affects the form of problem-solving methods and language processing techniques.
4. The construction of the Kernel should not embody a commitment to focusing on a particular kind of problem, e.g., language, but should be amenable to instruction in a number of problem areas.

2.1. The problem-solving component of the Kernel

The Kernel has two general forms of problem-solving unit, corresponding to two uses of the basic condition-action form of productions. The first recognizes a goal and proposes means to achieve it:

goal & conditions => possible means.

The means to achieving a goal can be one or more subgoals, direct actions on the TE, or requests to Ins.

The second form of production serves as a test or recognizer:

goal & conditions => goal success or failure or consequences.

The growth philosophy for the IPS revolves around means-ends analysis [12]. Knowledge added to the system forms a conceptual network of connections between goals, means to achieving them, and tests on the results of applications of means. Goals constitute the most meaningful portion of the dynamic state of the system (working memory), while means and tests are permanent productions added gradually through instruction and learning. It is important that this network of means and ends is defined at

the level of individual productions rather than, say, at some higher level of organization with productions used to code an interpretive mechanism for the network. For the means-ends structure must be applicable to creating, shaping, and correcting all aspects of the behavior of the system, down to the finest detail.

Augmentation of networks of means-ends structures leads to a flexible but highly inefficient computational structure. Strategies for converting or compiling these structures are a necessary component of the growth philosophy outlined here, which however will not be discussed further.

Two basic conventions built into the system help to make the basic production form adequate for general problem solving: the lexicographic event order conflict resolution rule and a taxonomy of PS control, represented in a particular way. Recall that the conflict resolution principle orders production firings based on the relative recency of data used. This gives a depth-first emphasis, focusing on recently proposed goals before older ones and allowing successes to propagate in orderly fashion. It does not preclude, however, having emerging conditions unexpectedly satisfy an older goal and lead to action quite distinct from what was the immediately preceding focus.

While the conflict resolution principle is built into the PS architecture, the Kernel's knowledge of control is by way of modifier tags that appear in most working memory elements. The current system of representation is based on an analysis of past PSs [13]. The basic representation form is:

(Primary Secondary Modifier Body)

A primary is a verb or main data structure name, while a secondary is an object of a verb, an attribute of a structure, or the name of a substructure. Some examples of primary-secondary pairs: examine object, interrogate value, object color, and phrase boundary.

The modifier is a list with positions occupied by values from predefined classes: goal values, data values, process values, truth values, and degrees of completion. By combining values from various classes, a large number of meanings can be assumed by a modifier, which in turn affects the interpretation by productions of the representational unit containing it.

Space does not permit giving the entire modifier system, but the main entities that are used in the Kernel are as follows (examples of actual representations appear in Section 3). The most important goal value indicates "Want", and marks units that are currently desired goals. Other goal values indicate "Old", "Don't-want", and "Neutral". Evocation, intermediate control, and results of processes employ goal values in combination with process and data values: "Activate", "Iterate", "Hold", "Result", and "Continue". Truth values are "True", "False", and "Unknown".

Modifier values are made coherent by certain established knowledge about control. For instance, a process is usually initiated by a "Want Activate" signal, which then becomes "Want Continue", if it has several steps to be performed. The steps are indicated by using degrees of completion, which are simply ordinals. When the process is started, the "Want Activate" becomes "Old Activate", but the content of the initiation signal itself is still available, should it become necessary later to examine it. Similarly, control for a process can go into a dormant "Hold" status until some pre-set condition arises, whereupon it reverts to its former status. When a process finishes, it may produce an item with modifier "Neutral Result".

The use of these explicit modifiers in the basic representational units makes the behavior of the system open for detailed self-examination, when combined with the basic openness of working memory. Such a simple scheme for managing control knowledge is

based on the ease of control in PSs generally, and its feasibility has been tested extensively on typical AI tasks [13], though its suitability for the present instruction task has yet to be verified in large-scale practice. As shown below, control knowledge can be easily expressed in the external language by using key phrases corresponding to modifier variants. The available knowledge and basis of control can similarly be expanded.

The Kernel itself is a problem solver (in the domains of language, building productions, etc.), and is written using the conventions just sketched. But it is also the producer of programs embodying the same conventions. Thus, initial instruction is constrained to be close to such forms. Later on, as IPS becomes more sophisticated, internal problem-solving method forms should be producible from instruction requiring more difficult mappings. Incidentally, the Kernel itself is simple enough that a straightforward instruction sequence should be able to reproduce it.

2.2. External language capabilities of the Kernel

The Kernel is built to understand a limited external language. The language capability has three aims: to make interactions with IPS readable by the instructors and by other AI researchers; to make the interactions occur in something other than a PS language; and to encode a number of representational conventions, so that instructors can refer to the same internal entity in a variety of ways - i.e., a mapping or assimilation facility, relating external to internal structures. To keep the Kernel simple, an initial language with rather rigid format has been chosen.

Language expressions are processed primarily in a bottom-up fashion, with only a few keywords having specific meanings to start with. That is, a keyword is recognized and classified, and a number of the actions associated with it (its semantics) are performed. A default action is taken for words with no known classification. Occasionally, a keyword

sets up anticipations for actions later in the input, giving the processing a partial top-down orientation. The default action for unclassified words is easily superseded, using the special-case conflict resolution principle. Along with the careful design of the Kernel to allow all of its goals and subgoals to be discussed in the external language, this use of special cases forms the basis for extensibility. A similar bottom-up approach, though not coded strictly incrementally, has been used successfully in a toy blocks domain [13].

The main form in the language is an image of a production or of a closely related set of productions. The form starts with "To", with an expression of a condition following, then a sequence of actions. "To" is taken as an abbreviation of "If you want to". For example,

To examine an object in some location , do view that location .

In this example, the keywords have been italicized. The other words are given in an ordering that corresponds to the basic primary-secondary form discussed above. Thus "examine object" is the essence of a representational unit forming the goal in a condition-side of a production to be built. Most of the keywords not shown deal with the formation of conjunctions and sequences of units, so that productions can test more complex conditions and perform more complex actions: "and", "then", and "if". "Some", "that", and a few other keywords allow the specification of match variables, as opposed to constants. Detailed examples of the use of the language in a simple instruction protocol are shown in the following section.

Another main keyword in the language is "Next". "Next" is followed by text very much like the "To" clause above. This allows a process for achieving some goal to be expressed as a set of closely related productions, related by being continuations or steps in the common process. That is, the "To" clause of an instruction signals the main or first

step in the solution of some goal (the phrase immediately following the "To"). When "Next" clauses follow the "To" clause, they give succeeding steps in the process, which presumably test the outcome of the first step and take further actions accordingly. Some form of the main goal appears in all of the productions constructed within such a set of clauses. This loose content-based association of productions is called a module, though nothing structural in the architecture distinguishes it. The productions in such a module share internal assumptions, since they arise from a contiguous instruction sequence. The module is known to other modules usually only through its main goal unit, which is its evoking condition. To connect this with the discussion on means-ends analysis above, the language allows the local (intra-module) sharing of assumptions about means to achieve a goal.

2.3. Building productions and the interface to the TE

The Kernel's third and fourth components are minimal: the system's abilities to build productions and to manipulate the TE effectively are expected to develop as system behavior develops and as considerations arise from the task that vary from our present preconceptions.

Basically only simple, direct ways exist for telling IPS what to store in production memory and what to do in the TE. Both capabilities rely on the closeness of external language expressions to internal forms. The main sentence form is an image of a production, so the operation needed to build an actual production is a simple iteration over a list of units extracted from an input string. The current strategy for adding to production memory treats the memory as an unordered set. To specify a TE operator, the instructor uses the keyword "do" and follows it with a phrase in the proper form for the operator. The expandability of both components rests on the basic openness of their

Kernel representation, rather than on specific structural design. The simple goals by which they are presently achieved are expressed using a small set of primaries and secondaries, along with a few modifier tags to indicate partial results and iterations.

2.4. Discussion of the Kernel design

To recap the basic strategy in building the Kernel, focus is placed on a primitive language capability embodying definite problem-solving and goal search methods. This is not because language issues are most important, but because language seems to provide the shortest path to easier instruction, to flexibility, and to the encoding of basic problem-solving method assumptions. That is, an instructable language system leaves a large amount of openness for further instruction without precluding desirable options. Our experience over the short history of our attempts, covering a dozen or so initial abortive Kernels, indicates that the best strategy is to rely more on spontaneous ad hoc methods arising from interactions than on initial knowledge about aspects of problem solving. This insures that important aspects of the system relating to the problem-solving task are themselves instructable, rather than "cast in concrete."

3. Sample System Behavior

The Kernel starts with essentially no behavioral capability with respect to the TE. Thus, it must be instructed in some basic TE tasks to build up a network of goals for doing more significant tasks. A sequence of progressively more complex tasks has been established to build complex abilities from the simple ones in the Kernel.

The first task attempted is to instruct IPS to look at the top of a stack of objects in

a TE location. This will involve Viewing the location (in the TE sense of View defined above), checking to see that there is in fact a stack of objects there, and then finding the top of that stack and Viewing it. Instructing it to achieve these goals gives it some material to work with on tasks that follow this first one, which include looking at objects other than the top one in a stack, determining the type of an object (which requires that it be looked at first), and comparing the types of two objects. More complex tasks include rearranging objects and determining the requirements and effects of machines in the TE.

The instruction sequence to be given now consists first of giving IPS a top-level subgoal sequence to achieve the goal. Then a concrete task is given, involving that goal, so that its behavior gives rise to subgoals that it can't solve, at which points further instruction is given.

IPS starts by asking for input, and it is given the following:

To examine an object the top in some location , want do view that location then want test the status of the value of that location composition , Next ...

In the text, "..." indicates that more is to come below; it doesn't appear in actual input. Note that the language is very primitive in expression, a consequence of our desire to include only the minimum necessary for communication of basic ideas, thus excluding familiar linguistic elaborations.

The system uses the word "next" to act as a temporary boundary for the input, and forms the following production before continuing to scan further.

```
R1      (examine object (want activate) top in =location) $ =c1
    --> (view.TE =location (want activate))
        (test status (want activate) value =location composition)
        (examine object (want continue 1) top in =location)
        (examine object (old activate) top in =location) (delete =c1) ;
```

The OPS notation for productions gives the name, R1, followed by the condition side,

followed by "-->", followed by the action side, and terminated by ";". "=" is used to mark variables as in "=location", which makes "location" a variable. "$" stands for Pattern-and, which allows two match expressions to use the same working memory expression. In R1, "$" is used to bind the single condition element and the variable c1 to a single working memory element. A production is executed as a result of matching its condition, which results in binding its variables to elements or subelements of working memory. Firing the production then results in asserting or deleting working memory elements according to the action-side forms ("assert" is implicit, "delete", explicit), in such a way that the leftmost action becomes the most recent working memory element, for purposes of conflict resolution.

The phrase between "To" and the first "," has been formed into the condition side of R1. The mapping of text to representational unit is direct, with the Kernel supplying the "want activate" modifier and the "$ =c1", which is used to delete the "want activate" form after it is converted to "old activate" at the end of the action side. "Direct mapping" means that words are added to a unit in the same order as given in the text. The remainder of the text is converted to a sequence of two representational units, the first two in the action side. This conversion is also quite direct. These serve as subgoals, the first of which, view.TE, will be achieved by Kernel mechanisms, and the second, by further instruction. The third action element is a signal that will stay in working memory until the first and second action elements have been recognized and their consequences followed up, at which time it will become most recent, and productions using it will become candidates for firing.

The first segment of the instruction has established a topic for the entire sequence up to the next ".", namely the goal of "examine object". The next segment of the instruction for this goal is:

... Next if the result of test the status is non-empty is the value of that location composition , want find the top of the value of that location composition , Else ...

This results in forming a second production, including as a condition the "want continue" unit from R1 and a unit corresponding to the text between "if" and ",":

(test status (neutral result) non-empty value =location composition)

The production tests the result of the second subgoal of R1 (to test the status of the composition attribute of the Viewed location) and proceeds with a further subgoal (to find the top of the composition list) if appropriate. To do the continuation, it includes a "want continue 2" unit with the form of the main topic.

The next phrase tells IPS what to do if another condition arises as a result of that second R1 subgoal.

... Else if the result of test the status is empty is the value of that location composition , the result of examine the object is failure , Next ...

This fragment gives one of the possible results of the topic goal, which indicates a failure (at present, in a non-informative way).

The next segment forms a production to recognize the result of the "find top" subgoal, View that result, and leave the name of the result in working memory (a "neutral result" form similar to the one above).

... Next if the result of find the top is some object , want do view that object and the result of examine the object is that object .

The system has now taken that instruction sequence and formed four productions. It then awaits further input, which is the following:

Try examine the object top in L23 .

This tells IPS to actually try to achieve an "examine object" goal for a particular TE

location. It would not get very far before stopping, in need of further instruction to allow it to achieve the "test status" goal (the second goal in the action side of R1). Ins determines what the system is in need of (if he can't remember) by asking it, "What want ?", which IPS answers by finding its most recent "want"-modified unit. Instruction for the "test status" goal can be given so that the system can achieve it directly. That is, a single production suffices to perform the test and return a result, without further subgoals. After that, it needs instruction on the "find top" goal, which can also be achieved directly.

4. Conclusions

The aim of this initial examination of the problem of an instructable PS has been to motivate the starting assumptions, to discuss some broad issues for PSs and for instructable systems, and to give some detail on where the project stands as a result of several design iterations.

A PS architecture has been developed that builds on substantial experience with past architectures. Its conflict resolution scheme appears to be compatible with incremental additive growth, and it essentially abandons the use of static orderings of the productions. A small kernel system of about 150 productions currently exists, embodying an incremental approach to solving problems, processing language, representing knowledge as productions, and interacting with a toy environment. The system's problem solving capabilities exploit a form of knowledge that is natural for PSs, means-ends analysis. With that, a taxonomy of some simple control mechanisms in PSs provides the system with the capability to do complex tasks.

The initial ingredients assembled seem ideally suited to attaining the incremental growth of a complex understanding system. Experimentation with a sequence of progessively more difficult tasks will reveal whether this is the case. However, it may not happen quickly. Experience so far, expressed in a good dozen kernels, has shown how difficult it is to get the details right, even while the general features of the scheme hold up rather well. Partly the difficulty is that our design intuitions imply a substantial "kernel" of capabilities in place, whereas it is necessary to grow (instruct) that "kernel" through a much leaner initial system, which is certainly highly atypical and counter-intuitive in many ways. Any attempt to lay down this larger "kernel" by an act of design (in the usual system-building fashion) seems sure to create a beast that is uninstructable except along limited dimensions. Thus a regimen of iteration and back-tracking at primitive levels seems necessary.

One issue that has arisen about the particular ingredients can serve to summarize and illuminate the current state of understanding of the approach. If the observable behavior of the PS is to be organized completely as a means-ends network, then what role do the properties of the PS architecture play? Wouldn't any other architecture or basic programming system do as well?

To see the force of this, consider what makes means-ends analysis attractive for the instructable-system task. It permits simple addition to the existing system, since increments are made by attaching new (alternative) methods to existing goals, from which other goals and methods may freely branch. Its network of goals and methods can be driven down to any fineness of processing detail, permitting the scheme to be used for any processing. If care is taken initially, then all aspects of the system will be formed as a means-ends structure; hence, all aspects will be open to modification by further means-

ends construction. To utilize such a scheme requires of the underlying programming system (which constitutes the operators and associated data structures through which the means-ends net works) only that it be complete, have a fine enough grain of action, and perhaps be rather simply composable.

These properties constitute a significant fraction of the claimed advantages of PSs. What do PSs provide that is not already latent in an approach that focuses purely on construction of means-ends networks?

Means-ends analysis dictates (hence provides) a structure of goals, methods, goal-tests, operators and operands (data structures). But it only provides functional roles, not the programming system. Subject to the conditions of fineness of detail and completeness, we do not see that PSs have a striking advantage over other homogeneous architectures for realizing these processes in general.

In a performance system the instances of these structures are provided by the system creator. In a learning system they must be provided by the system itself, and the properties of the architecture become relevant to how easy or hard that will be. An instructable system, though it can shade into a programmed system especially initially, is fundamentally a learning system.

Without being exhaustive, some of the important functions to be performed in learning a means-ends structure are the detection of error and/or the opportunity for learning, the construction of hypothesized means-ends structures, their installation so as to supersede selected pre-existing structure, their validation and debugging, and a secure environment within which learning experimentation can occur. Means-ends itself does help on some of these functions, notably installation and (possibly) supersession, but does not provide help with most of them.

PSs appear to be a useful architecture for a number of these aspects, though no claim can be made that it is especially perspicuous across the board. Detection, validation and debugging seem to require wide-band access to the existing knowledge in the system at all times, i.e., in a monitoring-like mode. This is distinctly a feature of PSs. Although not quite so obvious, wide-band access seems important also to the construction of new hypothesized goals and methods. Successful modifications depend, not just on some fixed procedure for correcting failed goals, but on detecting and attending to highly various features of the environment and to equally various aspects of past experience. The problem of memory search to make contact with relevant but disparately represented experience is not solved in PSs just by the recognition scheme, but the architecture seems a useful one for approaching the problem.

The extreme simplicity of the ultimate forms of both goal tests and method steps in PSs is probably also important. They can consist simply of throwing together some hopefully relevant distinctions on the environment as conditions along with some likewise hopefully relevant actions, then adding the collection to the program memory without further ado. This is a fundamentally task-oriented operation, unencumbered by syntactic ceremonies or other detailed knowledge. That the elementary form of functionally relevant additions can be so simple rests upon two other aspects that PSs seem to provide in some measure. The first is security, which is provided by all behavior in a PS being effectively monitored (by the whole production memory). It can be brought under interpretive control at any time to dampen the prospects of "sudden death". The second aspect is the related ability to program by debugging, i.e., by adding fragments and modifying the system later on the basis of self-observed behavior.

Our present assessment is that the ingredients will cooperate together in a mutually

supporting way. However, it has taken a long time to evolve reasonable conflict resolution rules, to take an important recent example [8]. It may also take a while, for example, to evolve systems that map to distant experience appropriately [9] but that can be serviced by the limited recognition abilities inherent in efficient recognize-act cycles.

5. References

[1] Bobrow, D. G. and Collins, A. (Eds.) Representation and Understanding: Studies in Cognitive Science, Academic Press, New York, NY, 1975.

[2] Bobrow, D. G. and Raphael, B. R. New programming languages for artificial intelligence research. Computing Surveys, Vol. 6: 3, 1974. Pp. 153-174.

[3] Buchanan, B. G. and Sridharan, N. S. Analysis of behavior of chemical molecules: Rule formation on non-homogeneous classes of objects. Proc. Third International Joint Conference on Artificial Intelligence, 1973. Pp. 67-76.

[4] Davis, R., Buchanan, B. and Shortliffe, E. Production rules as a representation for a knowledge-based consultation program. Report STAN-CS-75-519, Memo AIM-266. Stanford University, Computer Science Department, Stanford, CA, 1975.

[5] Davis, R. and King, J. An overview of production systems. Report STAN-CS-75-524, Memo AIM-271. Stanford University, Computer Science Department, Stanford, CA, 1975.

[6] Forgy, C. A production system monitor for parallel computers. Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1977.

[7] Forgy, C. and McDermott, J. The OPS reference manual. Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1976.

[8] McDermott, J. and Forgy, C. Production system conflict resolution strategies. Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1976. Also in Proceedings of Workshop on Pattern-Directed Inference Systems, Hawaii, 1977.

[9] Moore, J. and Newell, A. How can MERLIN understand?. In Gregg, L. (Ed.), Knowledge and Cognition, Lawrence Erlbaum Associates, Potomac, MD, 1973. Pp. 201-252.

[10] Newell, A. A theoretical exploration of mechanisms for coding the stimulus. In Melton, A. W. and Martin, E. (Eds.), Coding Processes in Human Memory, Winston and Sons, Washington, DC, 1972. Pp. 373-434.

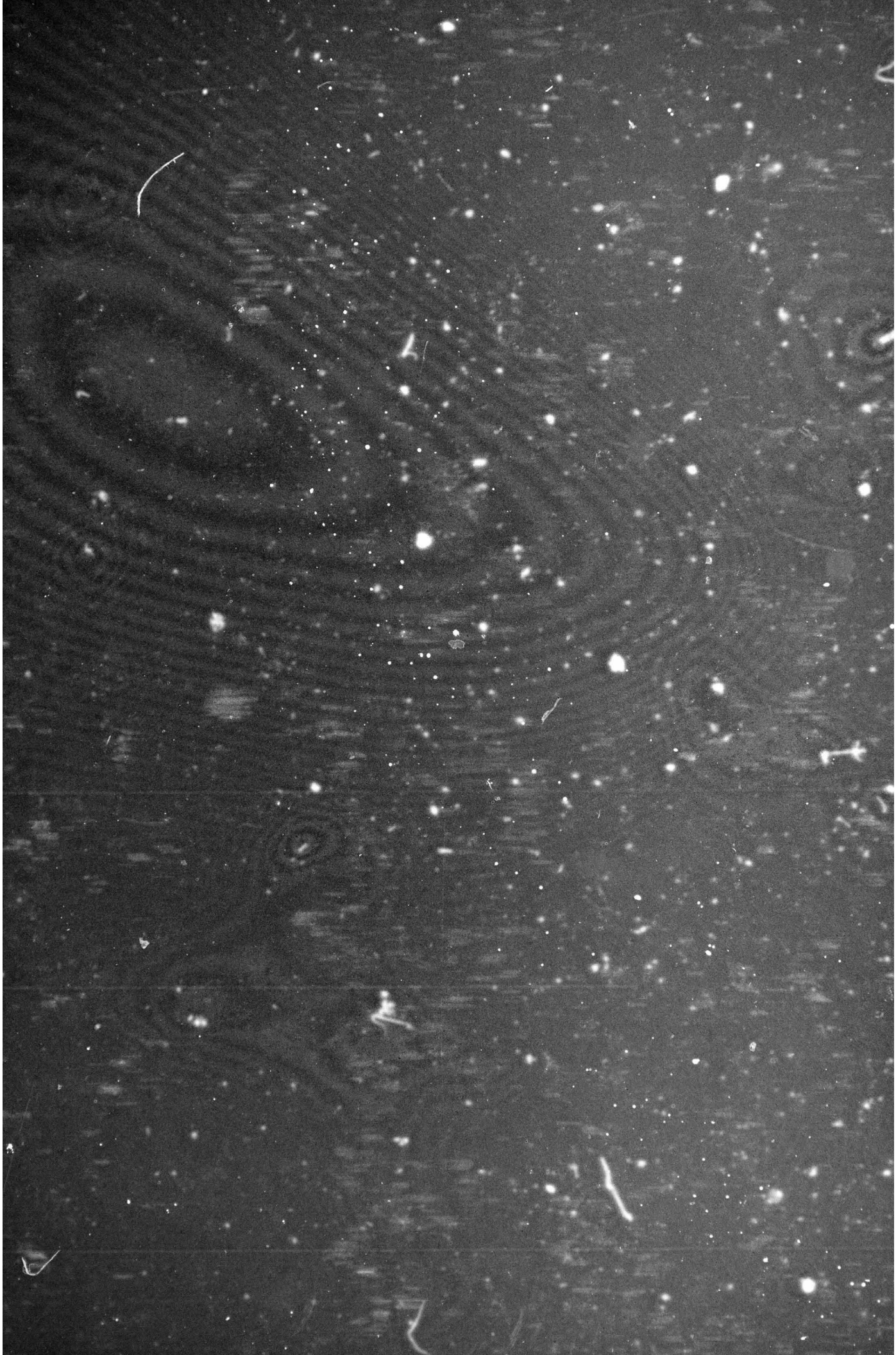
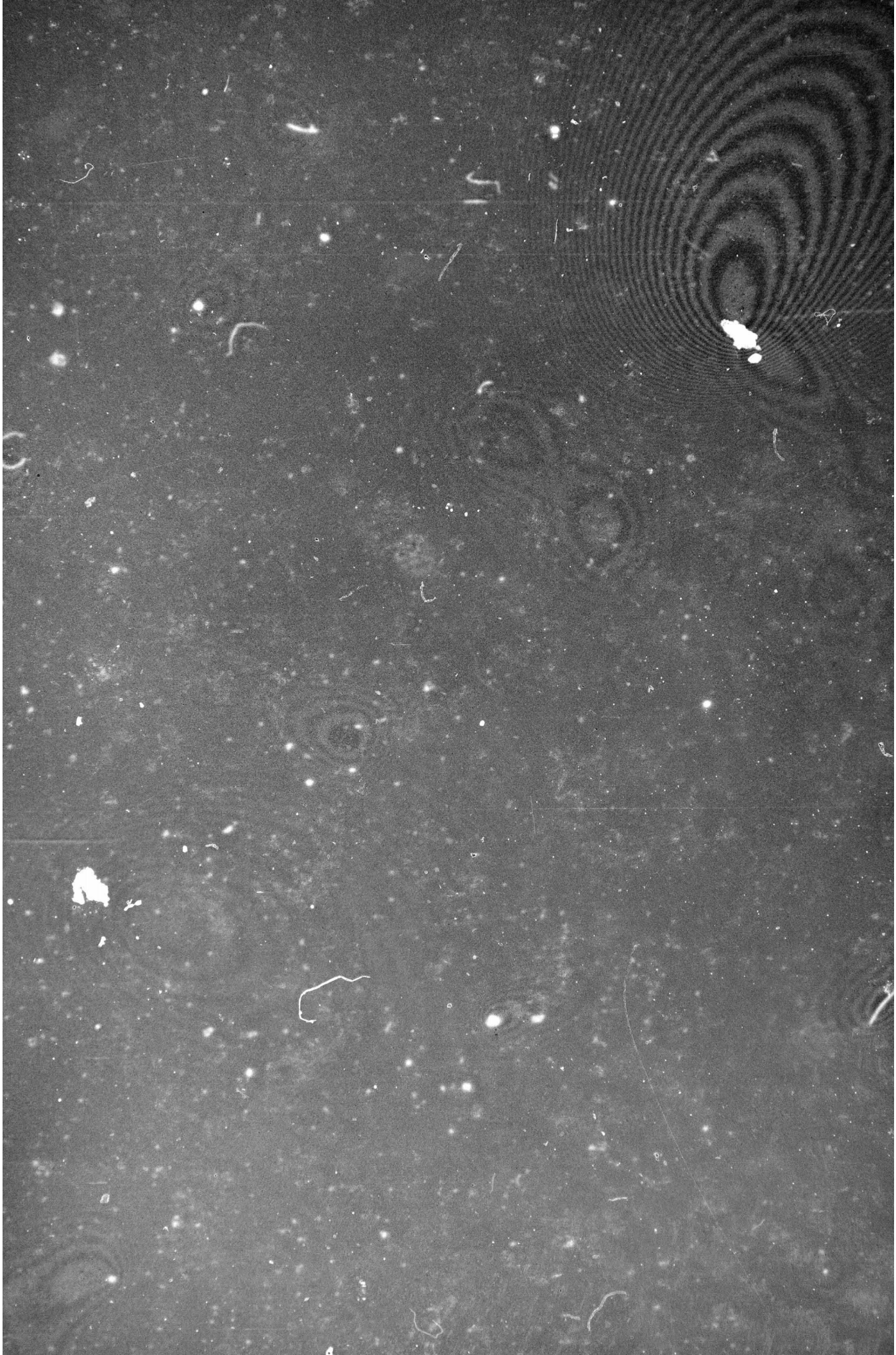
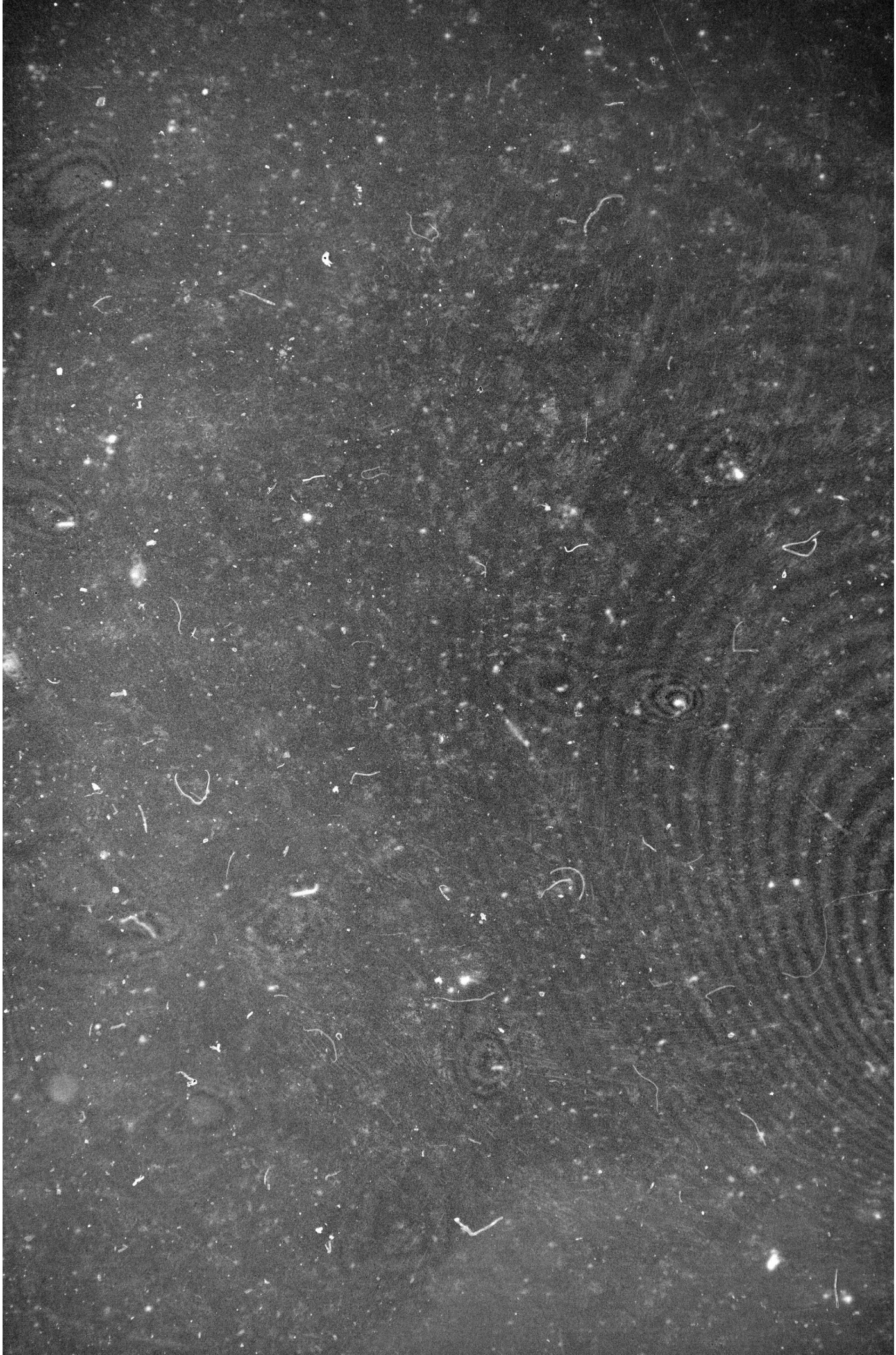
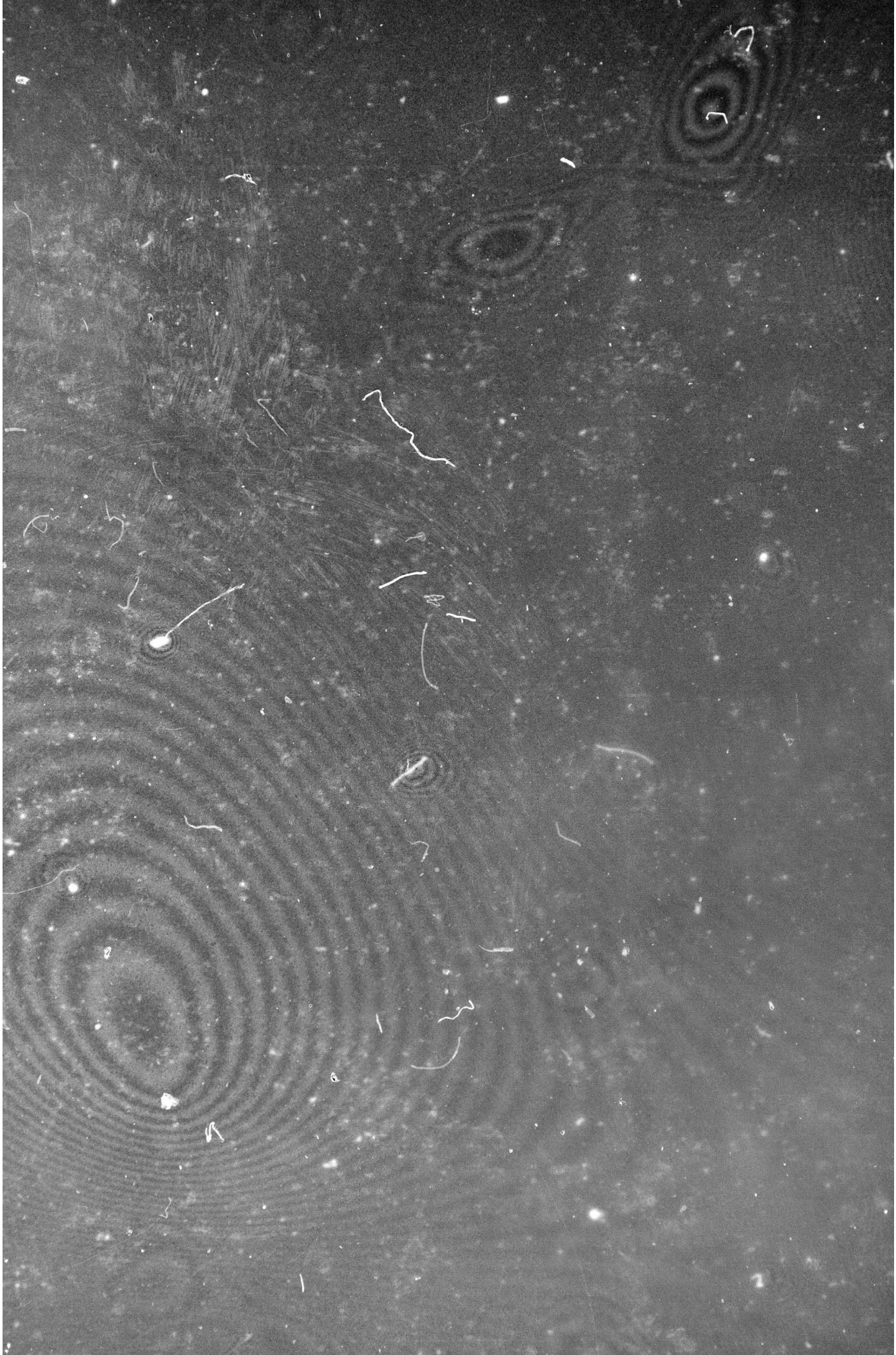
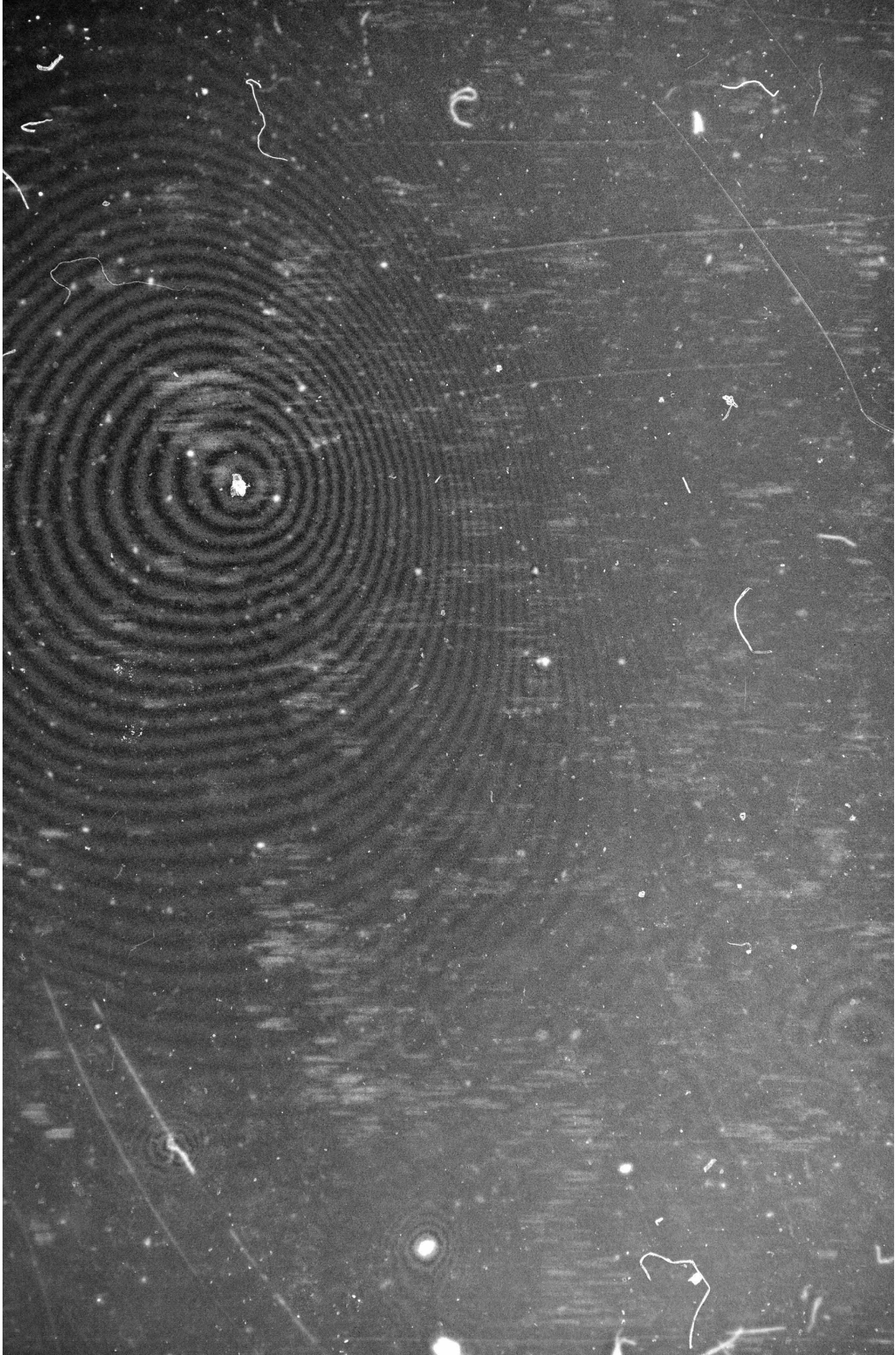
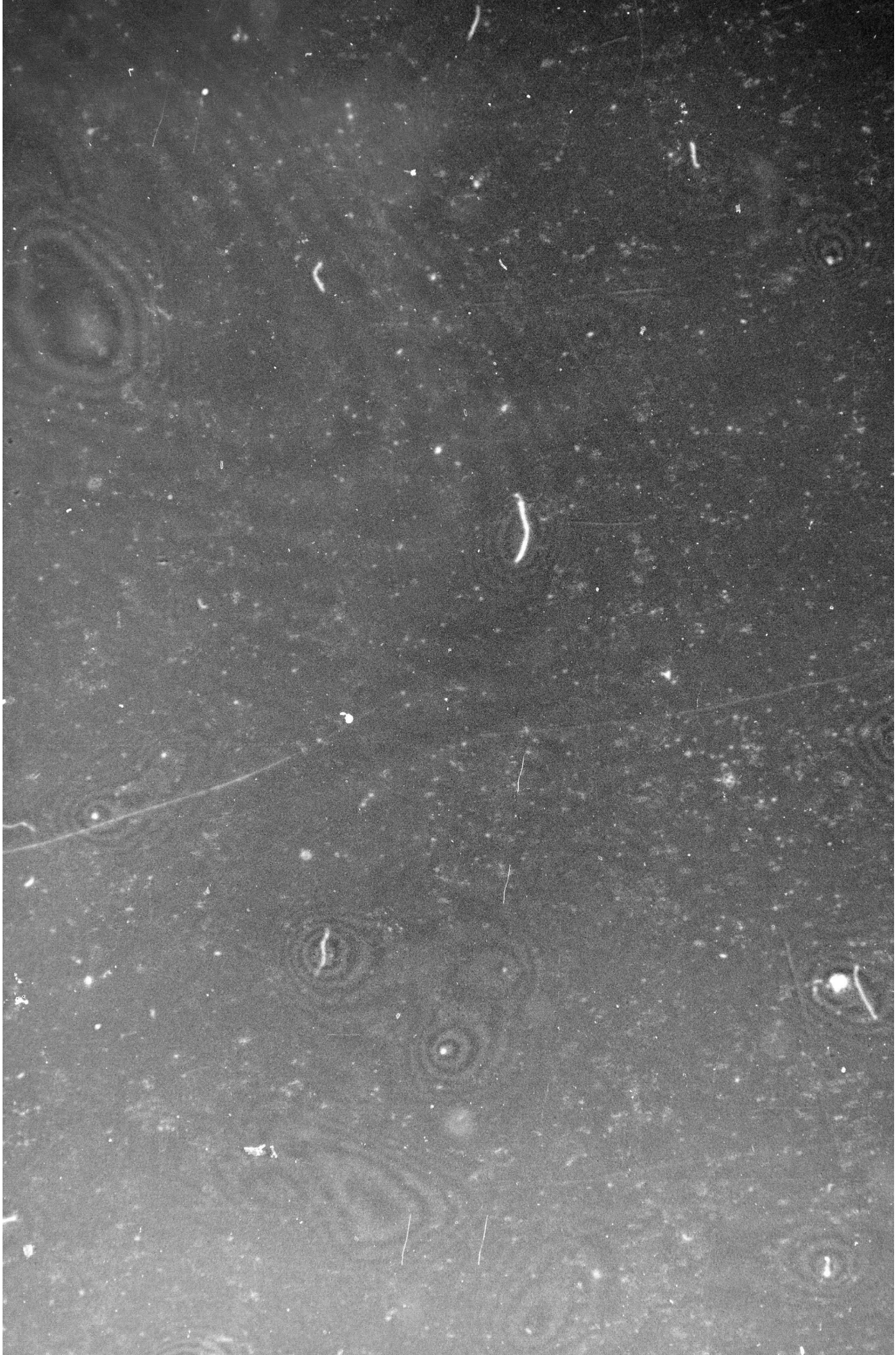
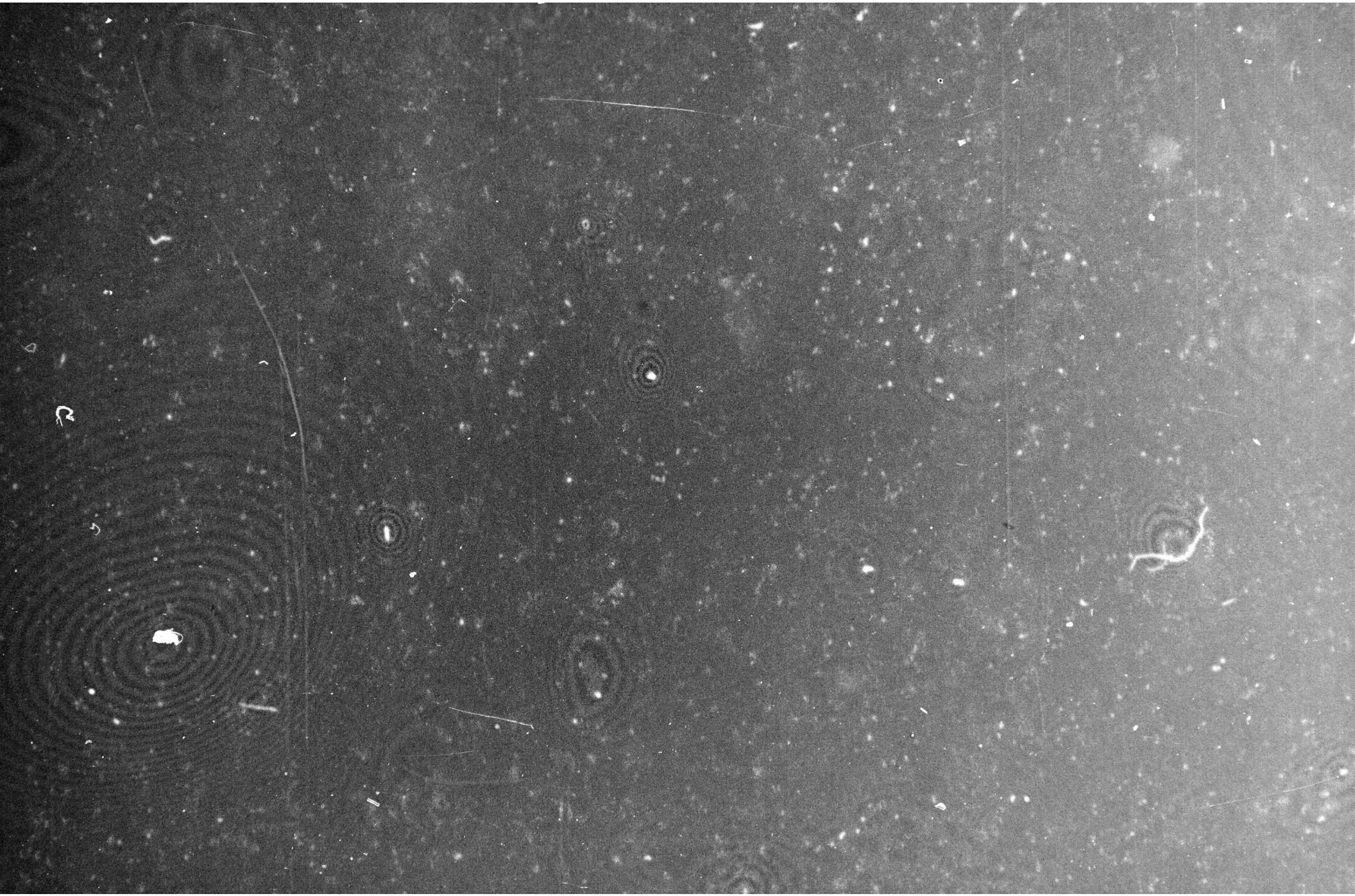
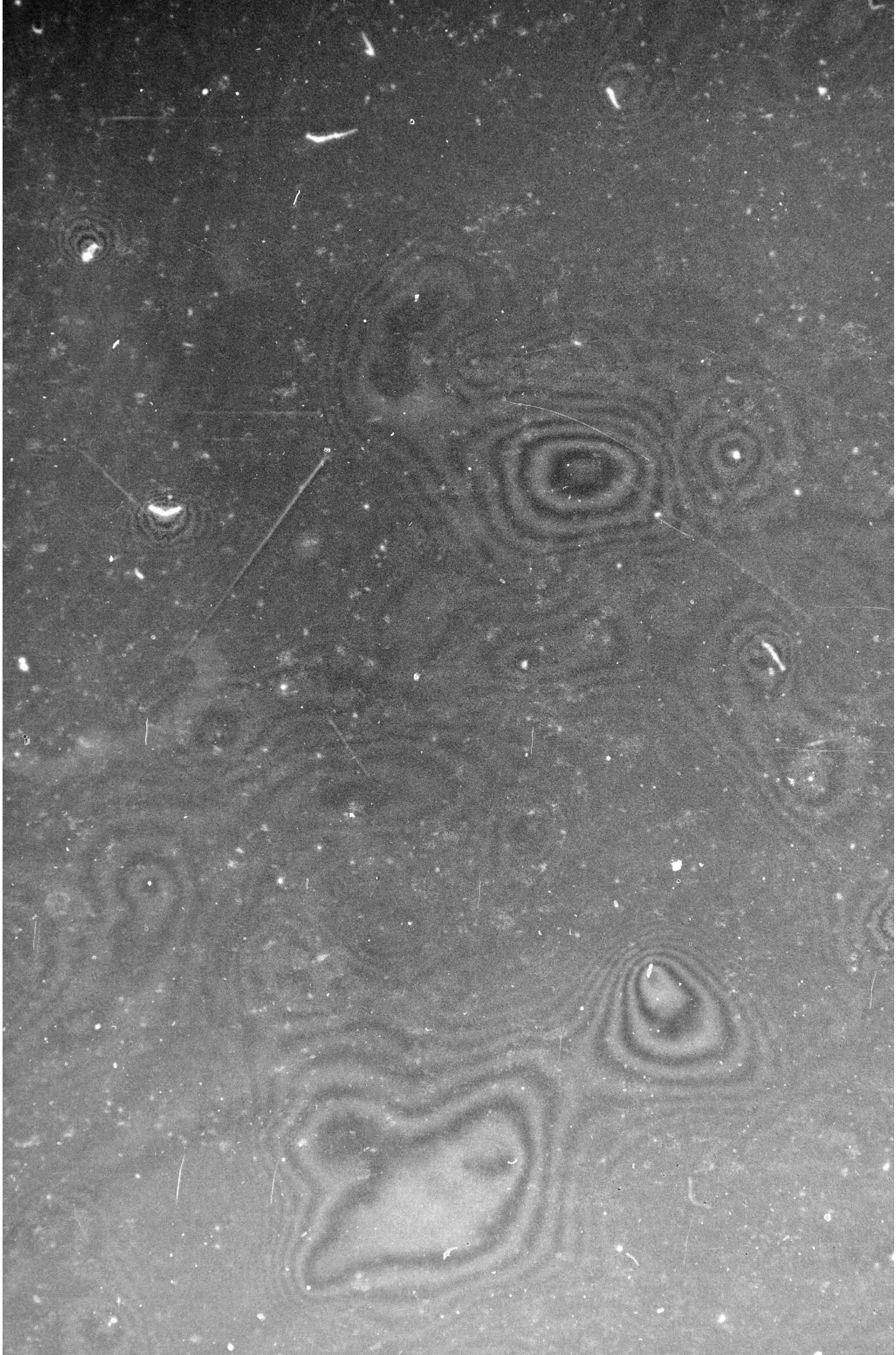
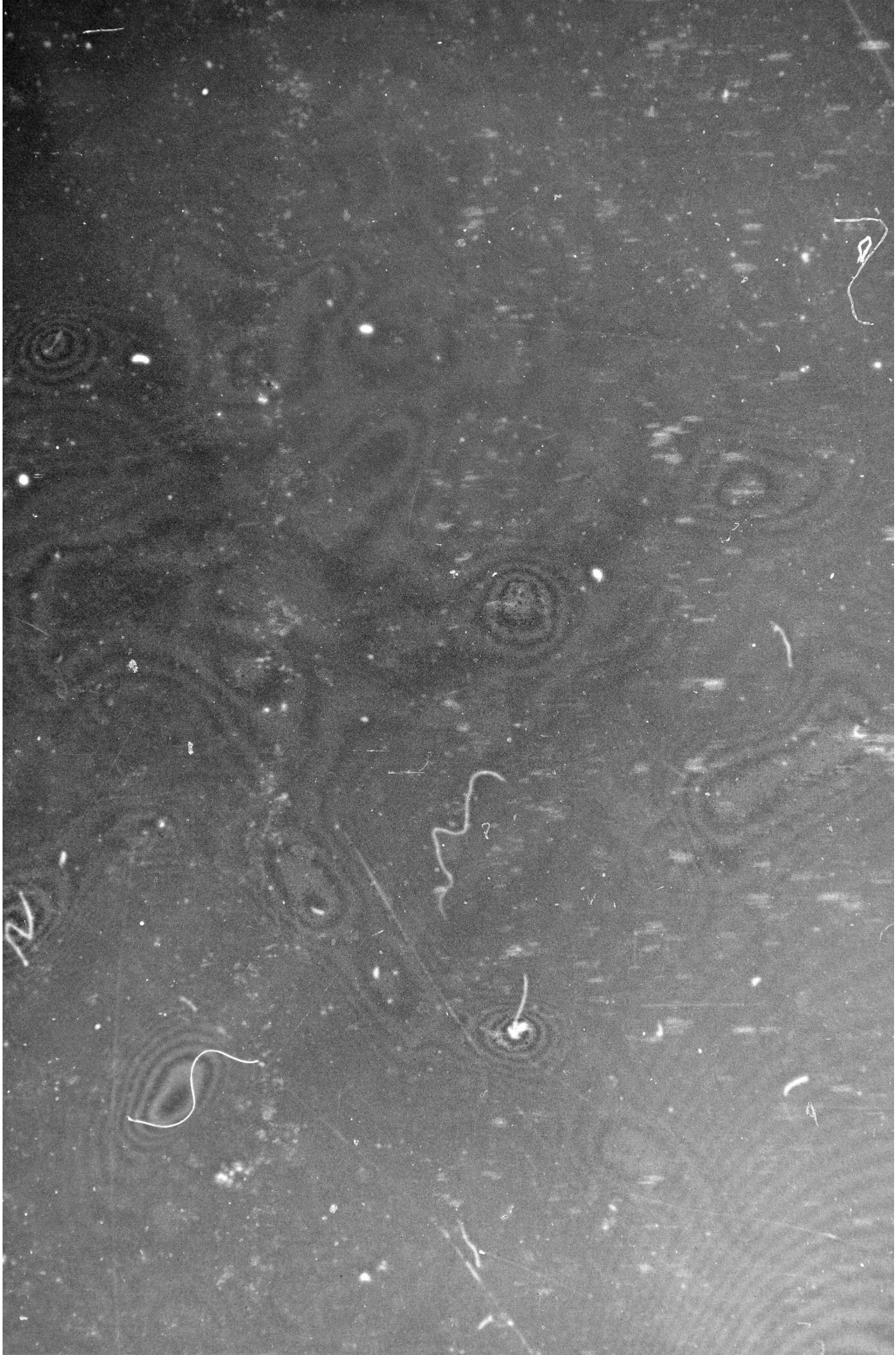
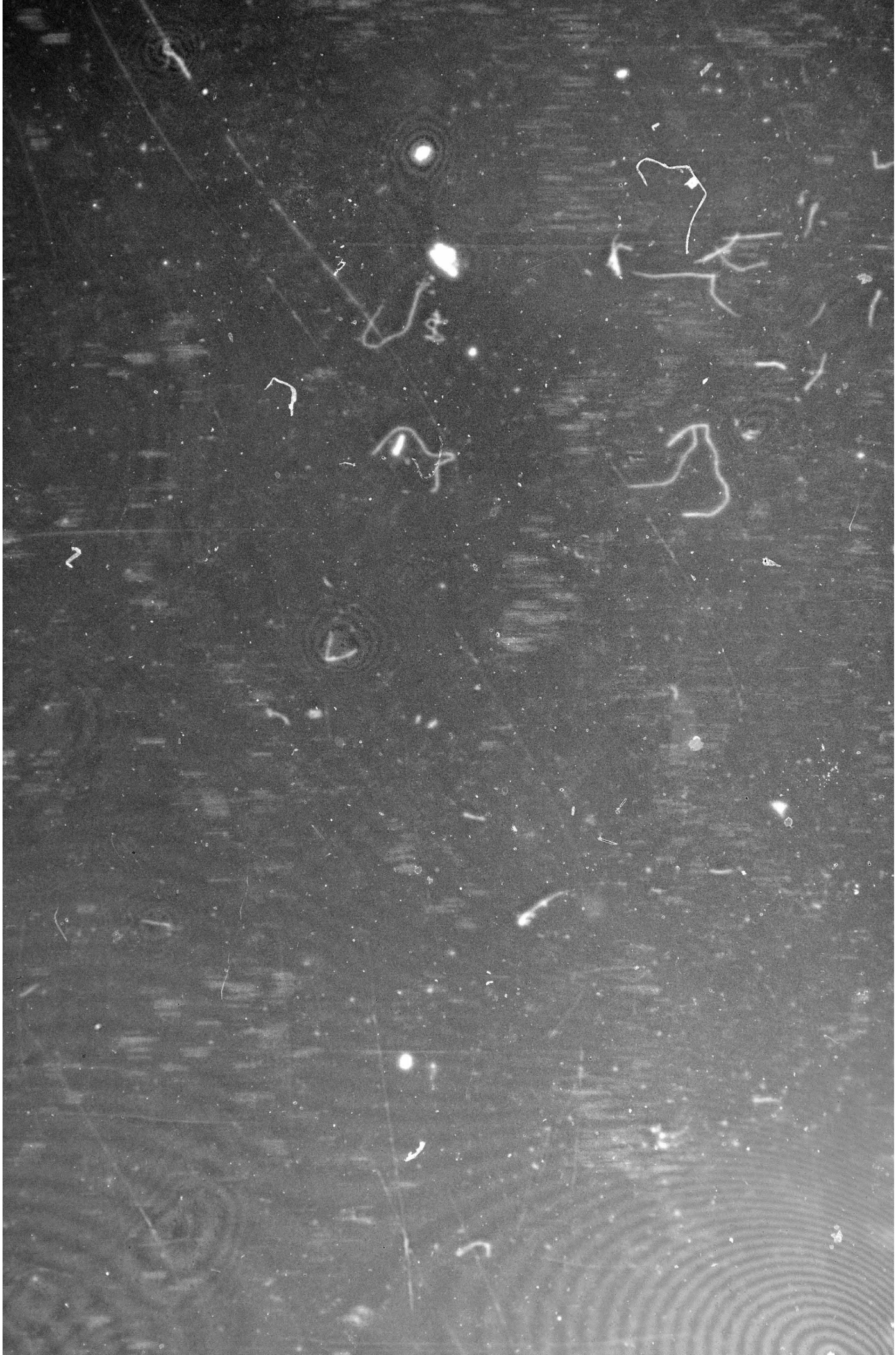
[11] Newell, A. and McDermott, J. PSG manual. Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1975.

[12] Newell, A. and Simon, H. A. Human Problem Solving, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[13] Rychener, M. D. Production systems as a programming language for artificial intelligence applications. Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1976.

5.1. Footnotes

#1. This paper draws ideas from an ongoing project involving C. Forgy, J. McDermott, Kamesh Ramakrishna, and P. Langley, in addition to the authors. Forgy and McDermott have been responsible for the implementation of the OPS production system architecture and other system facilities described here. We acknowledge the joint role these people have played in developing the ideas in this paper, but they are not responsible for our detailed expression or interpretation of them.

This research was supported in part by the Defense Advanced Research Projects Agency

under Contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.