Exploring the Enforcement of Private, Dynamic Policies on Medical Workflow Execution

Christopher A. Esterhuyse
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
c.a.esterhuyse@uva.nl

Tim Müller
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
t.muller@uva.nl

L. Thomas van Binsbergen
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Adam S. Z. Belloum
Informatics Institute, University of Amsterdam
Netherlands eScience Center
Amsterdam, The Netherlands
a.s.z.belloum@uva.nl

***Abstract*—We report on the ideas and experiences of adapting Brane, a workflow execution framework, for use cases involving medical data exchange and processing. These use cases impose new requirements on the system to enforce policies encoding safety properties, ranging from access control to legal regulations pertaining to data privacy. Our approach emphasizes users' control over the extent to which they cooperate in distributed execution, at the cost of revealing information about their policies.**

***Index Terms*—Workflow, Policy, Multi-Agent, Privacy, Safety**

I. Introduction

The Brane framework was originally developed by the EU PROCESS project[1] for use in exascale, high performance computing (HPC) [1]. Its focus is on enabling (data) scientists to execute high-level scripts, which abstract away the underlying, distributed workflow execution. Brane's central design tenet is affording the separation of users' concerns, in accordance with their specialized roles. For example, software developers provide the functional building blocks applied by scientists in scripts. Section II begins with an overview of Brane for HPC.

When automated systems compute and share data across organizational boundaries, user participation is predicated on the system's reliable enforcement of their *policies*. Intuitively, users are willing to share data, but want to retain control of its use. Ultimately, policies control the system at an *operational* level, i.e., how data is read and written (called *access control*), and processed (called *usage control*). Many languages exist to capture these notions in rule-based forms, well-suited to automated enforcement, e.g., XACML [2] and ODRL [3]. In the context of processing medical data, there is a further need for policies to capture *normative* concepts; these include various social policies, including organizational policies and legal regulations, such as the EU General Data Protection Regulation (GDPR) [4]. Generally, it is difficult to correctly translate between these representations of policy. In this work, we draw from the work on the eFLINT language, which aims to formalize many notions of policy such that they have unambiguous interpretations, both legally and operationally [5].

This paper reports on ongoing work to adapt Brane for the exchange and computation of medical data, across organizational boundaries. We describe the essential extensions to Brane to support *checkers*, a new, automated service, central to the enforcement of user policies (Section III). We then explore the design space of *cooperative checkers* (Section IV), which use an abstract, normative policy representation (e.g., eFLINT) in communicating with agents, i.e., both human users and automated services. This makes the system easier for humans to understand, and improves the efficiency of its execution. Our work emphasizes the users' control over the extent to which their policy information is exposed to other agents. This is motivated by the observation that medical policies ultimately encode private information, e.g., processing Bob's data requires Bob's written consent.

The last sections reflect on these contributions. We draw attention to related works (Section V) which inspire our own by solving sub-problems and related problems. Finally, we enumerate existing ideas and design decisions suited to exploration in future work (Section VI).

II. Brane for High-Performance Computing

Brane is a framework for distributed execution of scientific workflows. Its design maximizes the ease-of-use for (data) scientists, by providing them a high-level scripting abstraction, hiding the complexities of utilizing the underlying, distributed resources. The complexity, inherent in such a system, is made approachable by emphasizing the separation of concerns between cooperating (human) users and (automated) services.

This section primarily summarizes a previous work [1], reflecting Brane's roots in HPC. However, the relationships between services are presented here from a different perspective, for the sake of supporting the subsequent sections.

[1]https://www.process-project.eu

978-1-6654-6124-5/22/$31.00 ©2022 IEEE. Preprint of: C. A. Esterhuyse, T. Müller, L. T. Van Binsbergen and A. S. Z. Belloum, "Exploring the Enforcement of Private, Dynamic Policies on Medical Workflow Execution" *2022 IEEE 18th International Conference on e-Science (e-Science)*, 2022, pp. 481-486, doi: 10.1109/eScience55777.2022.00086

A. Three Kinds of Users; Three Kinds of Services

[1] specializes users according to their role: (1) a *(data) scientist* feeds a script to a *driver*, which drives the execution of the script, and returns the results; (2) a *software engineer* implements and registers *functions*, packaged, executable containers, usable in scripts; (3) a *system engineer* configures and maintains the automated services comprising the system.

In this paper, we present a simpler view than that presented in [1], defining only three kinds of (automated) services that constitute the system: *driver*, *planner*, and *worker*. In summary, a driver interfaces one scientist with the system at large, interpreting their script. Some work is delegated by drivers to planners, and then further delegated to workers.

In the sequel, we use *agent* to characterize the constituents of the distributed system, generalizing over users and services.

B. Each Driver Executes One Scientist's Script

Via a persistent communication session, a scientist feeds a *script* to their dedicated driver. Scripts are expressed in *BraneScript*, a purpose-built scripting language.[2] BraneScript is designed to be intuitive to data scientists through its similarity to many general-purpose programming languages, such as Python and Lua. BraneScript is imperative, defining sequences of statements for reading and writing local variables, manipulating branching and looping control flows, and evaluating expressions. Typical Boolean, integer, and string literals and operators are also included. Listing 1 gives an example of a simple script, which repeatedly updates `val`.

```
let val := 1 + 1;
while (val < 10) {
  val := g(f(val, 5)); }
```

Listing 1. An example of a BraneScript script, demonstrating its imperative control flow, integer literals and combinators, and (re)assignment of variables.

A driver parses and interprets the given script incrementally, one statement at a time. This entails modifying a persistent state. Much of this state is comprised of primitive data, such as integers, that is isolated to the driver. Other values are stored in a *virtual file system*[3], shared between services. We assume that this virtualization abstraction is preserved; two services never disagree on a shared value. We also assume that shared values are identifiable, e.g., with an absolute file path. This lets services refer to shared values in messages without accessing the values themselves. Drivers rely on sharing to delegate the work of evaluating each *function(-application) expression* to a planner. To scientists, a function is applicable to parameters to produce a result, as usual. To services, a function identifies a shared value, an executable container that emits output values when executed with given input values. On encountering a function expression, a driver encodes it in a *compute-task*, sends it to a planner in a message, and pauses its interpretation of the script until it is notified of the result. Thus, services *cooperatively schedule* [6] the evaluation of expressions.

[2]In [1], *Bakery* is defined as an alternative to BraneScript, differing only in its concrete syntax. Bakery prioritizes human-readability over brevity.

[3]Brane is currently implemented atop JuiceFS (https://juicefs.com).

C. Services Cooperate in Computing Workflows

Services delegate the computation of values to one another by exchanging messages: compute-tasks. Concretely, each compute-task (1) defines the functional relationship between an output value, a function, and a set of input values, and (2) encodes a *goal*, the obligation to compute the output value. A compute-task can be understood as a 'one-step' workflow. Fig. 1 gives an example of two inter-related compute-tasks.

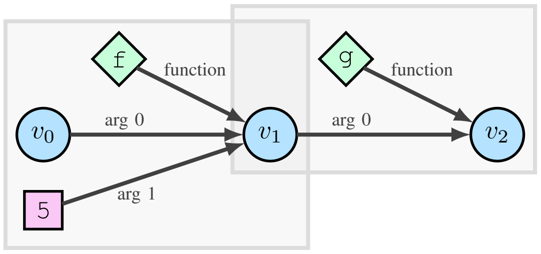

Fig. 1. The figure depicts the values (vertices) and functional relationships (edges, labelled suggestively) that constitute two compute-tasks (gray boxes), i.e., 'one-step' workflows. Each compute-task is depicted as graphically containing its logical constituents, i.e., included vertices, and edges between included vertices. Diamonds are functions. Squares are integer literals. This example corresponds to one execution of line 3 for the script in Listing 1, whose execution replaces the assignment `val` $\mapsto v_0$ with `val` $\mapsto v_2$.

Work is delegated by drivers to planners, and further by planners to *selected* workers. Each such selection presents an optimization opportunity, as not all workers can be relied upon to perform any given task at the same rate (or at all). This is motivated in practice, where workers are hosted by heterogenous hardware, and distributed over heterogenous network links. Planners are responsible for creating an abstraction of reliable task execution. Planners may make many selections per task, delegating the same work to any number of workers. In the sequel, we call a compute-task annotated with the worker selected to execute it a *compute-event*. As in [7], we call this activity *planning*, and its products *plans*.

In summary, drivers partition scripts into compute-tasks, planners plan these tasks, and workers execute these plans.

III. Brane with Policy Enforcement

This section details essential extensions, to the version of Brane presented in Section II, to afford the enforcement of users' *policies*. To this end, we introduce the *checker*, a novel service that encapsulates the policy of a *policy expert*, a novel user. This is comparable to how a driver interfaces with a scientist, and encapsulates a script. This section explains the way checkers interface with other agents via *authorization*, to facilitate enforcement. Until Section IV, we minimize the extent to which we define checkers' internals, e.g., we remain agnostic to the representation of policy used by policy experts.

A. Policies and Behavior

Scientists and policy experts have in common that they interface with the system via an abstraction appropriate to their roles. They are both ultimately concerned with influencing and observing what the system does, i.e., how the system behaves. BraneScript is the abstraction suited to scientists, who are concerned with prescribing the functional relationships between values, and prescribing goals. In contrast, policies prescribe particular safety properties to preserve, by stating what is, and is not, permitted to happen. In this context, policies are typically called *access control*, and *usage control*; intuitively, these prescribe which values may be accessed, and which computations may be performed, respectively. In either case, policies are defined in terms of modalities for 'time' and 'place', i.e., whether a value can be accessed depends on when it would be accessed, and by which agent.

For our purposes, we define the *behavior* of a system to be the set[4] of *events* that have *happened*. There are three kinds of events; a compute-event happens when it is executed, i.e., a particular worker executes a particular compute-task, producing an output value. A *read-* or *write-event* happens whenever a particular shared value is read or written, respectively, by a particular agent. We use *access* to generalize over reading and writing. At runtime, the system's behavior is extended to include events as they happen. Listing 2 shows an example behavior, with examples of all three kinds of events.

```
{      read-event(w0, f),
       read-event(w0,v0),
       read-event(w0, 5),
    compute-event(w0,v1,f,(v0,5)),
      write-event(w0,v1),
       read-event(w1, g),
       read-event(w1,v1),
    compute-event(w1,v2,g,(v1)),
      write-event(w1,v2)                 }
```

Listing 2. Example of a behavior, whose events are listed textually using suggestive, ad-hoc textual syntax, in an order that suggests realistic execution. This demonstrates what may result from the execution of the left and right compute-tasks in Fig. 1, by workers w_0 and w_1, respectively.

Let a *policy* denote a set of *compliant* behaviors. The preservation or restoration of the compliance of the system's behavior is called *enforcement*. Policies are useful while agents can rely on their enforcement; this formalizes the connection between users' expectations, and the system's implementation. In the context of some compliant behavior, we say that some events are compliant if the behavior that results from the events happening is also compliant.

In summary, agents model the system in terms of behavior, determining which events have happened, and should happen.

[4]This set-representation specifies no order on events that have happened. In practice, behavior may be extended with additional ordering information, e.g., events are annotated with timestamps. This was not essential to the narrative.

B. Private Policies and Public Authorization

Enforcement emerges from agents ensuring that only compliant events happen. Clearly, this necessitates planners and workers reasoning about policies to some extent. However, it is undesirable to give agents unchecked access to policies. This is motivated by realistic policies encoding compliance in terms of *private* information, which should remain hidden from some subset of agents. For example, a compute-event processing Bob's medical records is compliant only if Bob's written consent is stored in the hospital's server.

The role of the checker is to control the extent to which they publicize their policy information. Concretely, a checker decides which events to *authorize*, publicizing the fact that the event is compliant to their policy. The set of events authorized by checker c can be understood as an interface, written only by c, but read by all agents. Crucially, checkers hide policy information by keeping events unauthorized, as the absence of authorization does not imply non-compliance.

While checkers are free to create any authorizations at all, it is practical for them to remain passive, only creating authorizations on the *request* of planners. On the one hand, it affords a checker cooperating with the execution of scripts. On the other hand, it avoids the checker unnecessarily revealing policy information by authorizing events unnecessary for execution. Checkers must strike the balance, as best suits their policy experts' needs, between preserving the privacy of policy information (by creating few authorizations) and cooperating in execution (by creating many authorizations). Checkers are incentivized to model the extent to which attacking agents can recreate their policies via observing their authorizations. These attackers can draw from decades of research; e.g., attackers incrementally learn hidden automata (our policies) by making selected queries (our requests), and learning from the responses (our authorizations). For example, consider learning algorithms L^*, from 1987 [8], and $L^\#$, from 2022 [9].

C. Meta-Policies and Implicit Authorization

In practice, there are many reasons that a worker could infer that an authorization would be granted on request. In these cases, an agent can acquire authorization *implicitly*. Obviously, implicit authorization avoids the overhead of communication in which the worker would gain no (new) information. More interestingly, it lays the groundwork for statically-established limits on the powers of checkers *not to* authorize events.

Checkers implicitly authorize events with which they are not *involved*, where involvement is a relation defined entirely in terms of public, static information. This is expected to be sufficiently generic for many realistic use cases, as it can model a variety of concepts, including ownership, jurisdiction, and physical proximity. For example, involvement models the *(administrative) domains* of [10], a logical partition of agents and resources over their owners; a checker is involved with all events executed by workers with which it shares a domain. This touches on the big idea of *meta-policies*: policies defining which other policies may be enforced. For example, an event is implicitly authorized, because its prohibition is prohibited.

D. Checkers can Observe the System's Behavior

Our approach to enforcing the compliance of the system's behavior is inherently conservative; no event happens until all checkers have authorized it (either implicitly or explicitly). Thus, checkers and policy experts can infer that a given event *did not* happen, namely, if it remains unauthorized. A checker learns what *did* happen by observing the system's behavior. At some level, this must be communicated explicitly; for example, services gossip about each event as it happens, such that checkers are ultimately notified. Alternatively, we may rely on the virtual file system to share the behavior; e.g., it is written by workers, and read by checkers.

E. Mutable Policies and Temporary Authorization

Recall the example from Section III-B; Bob's consent is defined in terms of information outside the system. This demonstrates that policies are subject to unexpected change, such that they reflect changes in the outside world; e.g., Bob withdraws consent. This poses a problem; workers can no longer infer the compliance of an event from having previously observed its authorization. Ideally, such *staleness* is avoided through the use of instantaneous, synchronous communications. Realistically, we opt for an approximate solution, based on timestamps; checkers bound the validity of each authorization message with a timestamp, until which the authorization is assumed not to be stale.

IV. Cooperative Checkers

This section provides a design for the implementation of *cooperative checkers*. These checkers make effective use of the authorization interface defined in Section III, along with additional communications to coordinate with other, cooperative services. In summary, cooperative checkers share extra information about their policies, to improve cooperation.

A. Expressing Dynamic Policies in eFLINT

It is important that checkers are free to be unpredictable, as this makes them autonomous, i.e., able to make important decisions on their own; concretely, they may decide, arbitrarily, which authorizations to send. However, the productivity of the system as a whole is inhibited by this unpredictability; generally, planners cannot predict which events will be authorized. We characterize a *cooperative checker* as one that, on request of its policy expert, eases cooperation by sharing policy information with its peers.

Cooperative checkers formalize their policies in *eFLINT*, a language specialized for policies [11]. The language aims to bridge the gap between operational systemic rules, such as access control policies, and legal texts, such as the GDPR. This generality is afforded by eFLINT policies having sensible interpretations in both the normative world (in terms of duty-claim and power-liability relationships), and in the cyber-physical world (in terms of event-labelled transition systems).

Furthermore, eFLINT is of particular interest because of its *dynamism*; in some cases, the compliance of an event is defined as a predictable function of past events. A policy is a pair: a *specification* and a *state*. These terms are defined precisely in previous works on eFLINT [11]; here, it suffices to say that this partitioned representation is natural in practice, and facilitates a policy changing frequently, despite its specification changing only infrequently. This lays the groundwork for checkers publicizing small specification fragments that communicate much about their policies in future. Listing 3 gives an example of a fragment of an eFLINT specification, from which one can infer much about the policy as a whole.

```
Act compute Actor worker Related to
    output, function, arguments
  Conditioned by function.id != "f"
```

Listing 3. A fragment of an eFLINT policy's specification, with identifiers chosen to suggest their interpretation. From only this fragment, one can infer that any compute-event applying function `f` is non-compliant to the policy.

As a formal language, eFLINT reifies policies as concrete artifacts, enabling unambiguous communication of policy information to other agents, e.g., in the interface between a checker and its policy expert. Its formal semantics also lays the groundwork for automated tooling that assists humans in reasoning about policies, and their effects on behavior.

B. Cooperating with Planners; Publicizing Policy Hints

Within the *bounds* prescribed by its policy expert, upon request, a cooperative checker shares as much policy information as possible, as quickly as possible. This necessitates the checker continuously reasoning about the part of their policy that has been publicized so far. Within the bounds, each request is given a response; an event is authorized if compliant, and otherwise, explicitly *rejected*, expressed in a message returned to the sender. Furthermore, responses are annotated with *hints*. A hint encodes a policy, and is intended to assist the recipient in predicting the checker's policy in future.

Generally, hints are *unreliable*; formally, a hint is a policy, independent to the checker's policy. However, these hints represent a best effort; a hint is most useful when it is highly accurate for a long time. A special case of hint is *reliable*, owing to its explicitly-expressed, perfect accuracy. Such a hint is capable of encoding a set of authorizations. Observe that the usage of reliable hints obviates the usual authorization interface. This standardization in services' representation of policy is conceptually compelling, but may be undesirable in practice; it would necessitate all services reasoning about policy at a level of abstraction suited only to checkers.

C. Cooperating with Checkers; Transactional Authorization

Recall that checkers benefit from being notified of events as they happen, as this lets policies model the system. This lays the groundwork for modeling a useful computation primitive: the *(distributed) transaction*. A transaction returns the results of a computational task to a set of *observers*; crucially, each transaction is *atomic*, committing or aborting as an indivisible unit, such that all observers agree on its results (values).

Checkers cooperate to complete a transaction in three steps; (1) the results are computed, but remain inaccessible to the observers (2) checkers independently decide if some *condition* is met, as a function of the results; if so, then potentially, (3) the transaction commits, making the results readable by observers. Unless committed, a transaction is eventually aborted, and no result is observed. To realize step 3, the checkers must reach consensus on whether to commit or abort. Various consensus algorithms, suited to this task, are summarized in [12].

These distributed transactions let the system model *speculative execution*, as in [13]; checkers define the compliance of (the observable effects of) a workflow's execution as a function of its (intermediate) results; e.g., a checker authorizes a user reading v_2 of Fig. 1 only after the checker reads v_1.

V. Related Work

Our approach is inspired by a variety of existing works that share our central idea; automated agents constrain the execution behavior of a system by reasoning about some form of policy, and announcing decisions in another. [14] focuses on cooperation. All policies are always public. Our distinction between planners and checkers is thus de-emphasized; they coincide in 'site'. All sites have the same constraints on planning, modulo policy staleness. Our approach can roughly model that of [14] by checkers always sharing their entire policies as unreliable hints. Our checkers meet the requirements of 'auditor' defined in [15]. Our approaches differ in the details, as a consequence of using very different representations of script and policy. For example, our scripts omit notions of 'place', to be determined later by planners.

Other work is interesting, as it complements our own. For example, [16] pre-processes plans by injecting bridging functions, which change the plan's security properties, but do not change its outputs. Our hope is for this to provide a systematic means of executing the same scripts with new, compliant plans, not currently considered. For example, the expression `f(x)` is pre-processed to `f(decrypt(encrypt(x)))`, such that `x` crosses the network in encrypted form. In [17], planners improve the security of networked containers by isolating them in an overlay network, derived from scripts and policies.

Planning and compliance-checking can both be approached as distributed constraint satisfaction problems. As such, there is much relevant literature on the topic in general, reviewed in [18]. We are particularly interested in work that leverages properties specific to our application context to improve efficiency. For example, [19] drives the behavior of the actuators in a cyber-physical system. A complex problem is solved via its decomposition into simpler, independent problems.

Finally, our notion of behavior and our approach to generating it from behavioral specifications (our scripts and policies) is inspired by similar work on synchronous protocols. This includes the static generation of web service orchestration code from Reo [20], and Scribble [21], and the dynamic generation of network communications from Reo [22], and PDL [13]. These synchronous languages also inspired the transactional approach to authorization, presented in Section IV-C.

VI. Future Work

Work is ongoing on extending Brane's implementation to reflect Section III-B, affording automatic enforcement via authorization. Emphasis is on enabling *auditing*, holding an agent responsible for providing evidence that its actions are compliant. We imagine an approach based on cryptography; for each event that happens, an authorization signed by each checker must be available. This lays a foundation for improvements to the system's robustness and flexibility.

There is a need to formalize the trust relationships between agents, as a function of policies. The aim is to minimize users' vulnerabilities to exploitation, as a consequence of erroneously expecting their vulnerabilities to be protected. This phenomenon is called 'the dark side of trust' in [23].

Section III-C introduced the notion of meta-policies, systemic approaches to constraining which policies checkers enforce. There is a large design space of simple meta-policies that have profound effects on the system's properties. For example, if any checker in group g authorizes any e, then all in g authorize e. How are such meta-policies expressed? Is it practical to model realistic schemes, e.g., n-grid group authorization? [24] Can agents disagree on meta-policies?

Hints present another significant design space, letting checkers and planners cooperate in finding policy-compliant plans. Consider planning the evaluation of `f(x)`. First, the planner announces the task to the checker. Second, the checker encodes a 'partial plan' as a policy; the task is compliant if not executed by worker w_0. Finally, the planner assigns the task to w_1.

VII. Conclusion

This paper presents ongoing work to prepare Brane for the exchange and processing of medical data. This requires users to retain control of how their data is shared and processed. Users constrain the system's behavior via policies, modeling notions ranging from access control to legal regulations.

Our approach revolves around the explicit reification and tracking of what events comply to policies, and what events have happened. Checkers enforce user policies at an operational level, by authorizing planned events. Checkers balance competing requirements. On the one hand, checkers maximize the privacy of the policies being enforced, e.g., by authorizing few events. On the other hand, checkers maximize the publicity of policies, such that other agents may make informed decisions, increasing the efficiency of execution.

Much work remains to be done in exploring the various design decisions touched upon in this paper. In time, the hope is that future findings confirm our expectations, that Brane's current strengths can be retained, while also enforcing policies sufficiently expressive for a wide variety of use cases.

Acknowledgments

This research is part of the EPI project and is supported by NWO in the Commit2Data – Data2Person program under contract 628.011.028. The authors are funded, in part, by the AMdEX Fieldlab project supported by Kansen Voor West EFRO (KVW00309) and the province of North-Holland.

References

[1] O. Valkering, R. Cushing, and A. Belloum, "Brane: A framework for programmable orchestration of multi-site applications," in *17th IEEE International Conference on eScience, eScience 2021, Innsbruck, Austria, September 20-23, 2021*. IEEE, 2021, pp. 277–282. [Online]. Available: https://doi.org/10.1109/eScience51609.2021.00056

[2] A. Matheus, "How to declare access control policies for XML structured information objects using oasis' extensible access control markup language (XACML)," in *38th Hawaii International Conference on System Sciences (HICSS-38 2005), CD-ROM / Abstracts Proceedings, 3-6 January 2005, Big Island, HI, USA*. IEEE Computer Society, 2005. [Online]. Available: https://doi.org/10.1109/HICSS.2005.300

[3] S. Guth and M. Strembeck, "A proposal for the evolution of the ODRL information model," in *Proceedings of the First International Workshop on the Open Digital Rights Language (ODRL), Vienna, Austria, April 22-23, 2004*, R. Iannella and S. Guth, Eds., 2004, pp. 87–106. [Online]. Available: http://odrl.net/workshop2004/paper/odrl-guth-paper.pdf

[4] 2018 reform of eu data protection rules. European Commission. [Online]. Available: https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf

[5] L. T. van Binsbergen, M. G. Kebede, J. Baugh, T. M. van Engers, and D. G. van Vuurden, "Dynamic generation of access control policies from social policies," in *The 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2021) / The 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2021), Leuven, Belgium, November 1-4, 2021*, ser. Procedia Computer Science, N. Varandas, A. Yasar, H. Malik, and S. Galland, Eds., vol. 198. Elsevier, 2021, pp. 140–147. [Online]. Available: https://doi.org/10.1016/j.procs.2021.12.221

[6] S. Saewong and R. Rajkumar, "Cooperative scheduling of multiple resources," in *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*. IEEE Computer Society, 1999, pp. 90–101. [Online]. Available: https://doi.org/10.1109/REAL.1999.818831

[7] D. A. Cohen, J. Crampton, A. Gagarin, G. Z. Gutin, and M. Jones, "Iterative plan construction for the workflow satisfiability problem," *J. Artif. Intell. Res.*, vol. 51, pp. 555–577, 2014. [Online]. Available: https://doi.org/10.1613/jair.4435

[8] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[9] F. W. Vaandrager, B. Garhewal, J. Rot, and T. Wißmann, "A new approach for active automata learning based on apartness," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 223–243. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_12

[10] F. Dijkstra, B. van Oudenaarde, B. Andree, L. Gommans, P. Grosso, J. van der Ham, K. Koymans, and C. T. A. M. de Laat, "A terminology for control models at optical exchanges," in *Inter-Domain Management, First International Conference on Autonomous Infrastructure, Management and Security, AIMS 2007, Oslo, Norway, June 21-22, 2007, Proceedings*, ser. Lecture Notes in Computer Science, A. K. Bandara and M. Burgess, Eds., vol. 4543. Springer, 2007, pp. 49–60. [Online]. Available: https://doi.org/10.1007/978-3-540-72986-0_5

[11] L. T. van Binsbergen, L. Liu, R. van Doesburg, and T. M. van Engers, "eflint: a domain-specific language for executable norm specifications," in *GPCE '20: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Virtual Event, USA, November 16-17, 2020*, M. Erwig and J. Gray, Eds. ACM, 2020, pp. 124–136. [Online]. Available: https://doi.org/10.1145/3425898.3426958

[12] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds. USENIX Association, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[13] C. Esterhuyse and H. Hiep, "Reowolf 1.0: Project documentation," *Computer Security*, no. CS-2001, 2020.

[14] L. Veen, S. Shakeri, and P. Grosso, "Secure data sharing and distributed processing with Mahiru," Apr. 2022, Funded by the Netherlands eScience Center and NWO under the SecConNet project (27017G18). [Online]. Available: https://doi.org/10.5281/zenodo.6497704

[15] L. Zhang, R. Cushing, L. Gommans, C. T. A. M. de Laat, and P. Grosso, "Modeling of collaboration archetypes in digital market places," *IEEE Access*, vol. 7, pp. 102 689–102 700, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2931762

[16] J. A. Kassem, C. de Laat, A. Taal, and P. Grosso, "The EPI framework: A dynamic data sharing framework for healthcare use cases," *IEEE Access*, vol. 8, pp. 179 909–179 920, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3028051

[17] S. Shakeri, L. Veen, and P. Grosso, "Evaluation of container overlays for secure data sharing," in *45th IEEE LCN Symposium on Emerging Topics in Networking, LCN Symposium 2020, Sydney, Australia, November 16-19, 2020*, H. Tan, L. Khoukhi, and S. Oteafy, Eds. IEEE, 2020, pp. 99–108. [Online]. Available: https://doi.org/10.1109/LCNSymposium50271.2020.9363266

[18] D. L. Sallach, "Distributed constraint satisfaction: Foundations of cooperation in multi-agent systems *by Makoto Yokoo*," *J. Artif. Soc. Soc. Simul.*, vol. 8, no. 2, 2005. [Online]. Available: http://jasss.soc.surrey.ac.uk/8/2/reviews/sallach.html

[19] V. Degeler and A. Lazovik, "Dynamic constraint satisfaction with space reduction in smart environments," *Int. J. Artif. Intell. Tools*, vol. 23, no. 6, 2014. [Online]. Available: https://doi.org/10.1142/S0218213014600276

[20] S.-S. T. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, and H. Afsarmanesh, "Automatic code generation for the orchestration of web services with reo," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2012, pp. 1–16.

[21] J. King, N. Ng, and N. Yoshida, "Multiparty session type-safe web development with static linearity," in *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, ser. EPTCS, F. Martins and D. Orchard, Eds., vol. 291, 2019, pp. 35–46. [Online]. Available: https://doi.org/10.4204/EPTCS.291.4

[22] J. Proença, D. Clarke, E. P. de Vink, and F. Arbab, "Dreams: a framework for distributed synchronous coordination," in *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, S. Ossowski and P. Lecca, Eds. ACM, 2012, pp. 1510–1515. [Online]. Available: https://doi.org/10.1145/2245276.2232017

[23] D. A. Pienta, S. Tams, and J. Thatcher, "Can trust be trusted in cybersecurity?" in *53rd Hawaii International Conference on System Sciences, HICSS 2020, Maui, Hawaii, USA, January 7-10, 2020*. ScholarSpace, 2020, pp. 1–10. [Online]. Available: https://hdl.handle.net/10125/64264

[24] W. Shieh, B. P. Weems, and K. M. Kavi, "An n-grid model for group authorization," in *Sixth Annual Computer Security Applications Conference, ACSAC 1990, 3-7 December, 1990, Tucson, Arizona, USA*. IEEE, 1990, pp. 384–392. [Online]. Available: https://doi.org/10.1109/CSAC.1990.143813