



- (51) **International Patent Classification:**
G06Q 10/06 (2012.01) *G06Q 10/10* (2012.01)
- (21) **International Application Number:**
PCT/US2016/014937
- (22) **International Filing Date:**
26 January 2016 (26.01.2016)
- (25) **Filing Language:** English
- (26) **Publication Language:** English
- (71) **Applicant:** HEWLETT PACKARD ENTERPRISE DEVELOPMENT LP [US/US]; 11445 Compaq Center Drive West, Houston, Texas 77070 (US).
- (72) **Inventors:** KALIBJIAN, Jeff; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US). SCHMIDT, Peter; 585 South Boulevard, Pontiac, Michigan 48341 (US). FLOYD, Christian; 116 Gillin Road, Ambler, Pennsylvania 19002 (US).
- (74) **Agents:** KUPETS, Mark J. et al.; Hewlett Packard Enterprise, 3404 E. Harmony Road, Mail Stop 79, Fort Collins, Colorado 80528 (US).

- (81) **Designated States** (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.
- (84) **Designated States** (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— as to the identity of the inventor (Rule 4.17(i))

[Continued on next page]

- (54) **Title:** PROPAGATION FROM ACTUAL STATE TO DESIRED STATE

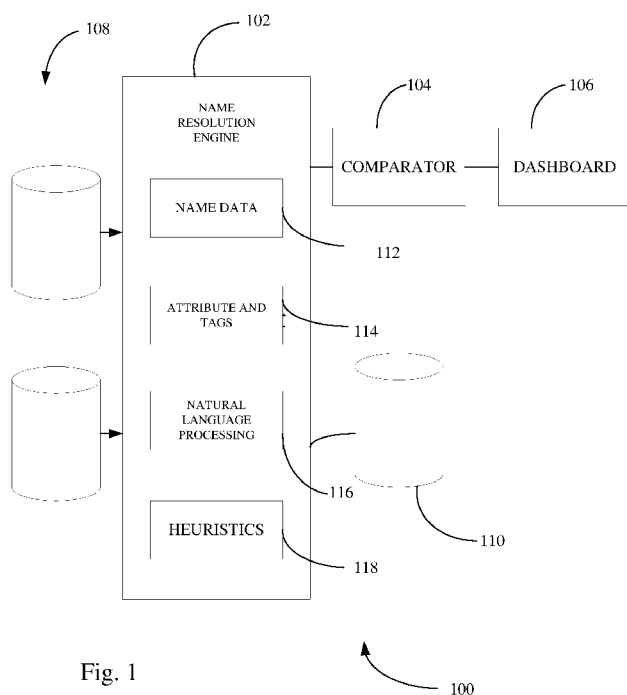


Fig. 1

(57) **Abstract:** Systems and methods propagate from actual state to desired state are disclosed. A first list is generated identifying at least one element in an actual state and a second list is generated identifying at least one element in a desired state. Discrepancies in names of the element in the actual state and desired state are resolved to determine whether the element in the actual state is the same as the element in the desired state. The first list is compared to the second list to determine whether to remove the element in the actual state or whether to add the element in the desired state.



— *as to applicant's entitlement to apply for and be granted
a patent (Rule 4.17(ii))*

Published:

— *with international search report (Art. 21(3))*

PROPAGATION FROM ACTUAL STATE TO DESIRED STATE

Background

[0001] IT (information technology) asset management, sometimes referred to as IT inventory management, is the set of practices that join financial, contractual and inventory functions to support life cycle management and strategic decision making for organization assets in the IT environment. IT assets include elements of software and hardware that are found in the IT environment. One aspect of IT asset management includes software asset management, which involves the purchase, deployment, maintenance, utilization, and disposal of software applications within an organization.

Brief Description of the Drawings

[0002] Figure 1 is a schematic diagram illustrating an example system for propagation from an actual state to a desired state.

[0003] Figure 2 is a block diagram illustrating an example method for propagation from an actual state to a desired state that can be implemented on the system of Figure 1.

[0004] Figure 3 is a block diagram illustrating an example method for processing names in one of the actual state and the desired state of the method Figure 2.

[0005] Figure 4 is a flow diagram illustrating an example method for processing names in the other of the actual state and the desired state of the method Figure 2.

[0006] Figure 5 is a flow diagram illustrating an example method of the method of Figure 3.

[0007] Figure 6 is a flow diagram illustrating an example method for processing a match failure from the method of Figure 4.

[0008] Figure 7 is a block diagram illustrating an example method for comparing names in the actual state and the desired state of the method of Figure 2.

[0009] Figure 8 is a schematic diagram illustrating an example computing device that can be used to implement the system of Figure 1 and the methods of Figure 2-7.

Detailed Description

[0010] In the following detailed description, reference is made to the accompanying drawings that form a part hereof, and in which is shown by way of illustration specific examples in which the disclosure may be practiced. It is to be understood that other examples may be utilized and structural or logical changes may be made without departing from the scope of the present disclosure. The following detailed description, therefore, is not to be taken in a limiting sense, and the scope of the present disclosure is defined by the appended claims. It is to be understood that features of the various examples described herein may be combined, in part or whole, with each other, unless specifically noted otherwise.

[0011] The disclosure relates to propagating systems from an actual state or configuration to a desired state or configuration. The disclosed examples relate to methods and systems for automated propagation from an actual software configuration to a desired software configuration in IT systems. But the systems and methods described are not intended to be so limited to software asset management. One skilled in the art can readily apply the techniques disclosed here to IT asset management in general, to inventory control of IT assets as well as non-IT assets, and to other examples. Software asset management is but one example of application of the disclosed methods and systems.

[0012] As part of an attempt to manage and secure assets, such as IT assets, organizations seek to define a desired hardware and software configuration of systems. Often, actual configurations of systems are different than this desired configuration. Discrepancies between actual and desired configurations can be due to a variety of reasons including time lags, personnel changes, and

piecemeal purchases of hardware and software. In order to address discrepancies, organizations can employ tools that provide an abstract model for each IT asset, such as a computing device, that describes its actual software configuration and its desired software configuration. The goal of such models is to make the actual software configuration match the desired software configuration.

[0013] Asset management applications attempt to reconcile discrepancies between the actual state and the desired state using discovery tools that catalog assets and software elements typically by name or other common identifier. Although many proposals have been submitted to uniquely identify software, no one policy has been universally accepted. For example, ISO/IEC (International Organization for Standardization and the International Electrotechnical Commission) 19770 family of standards for IT asset management address the processes and technology for managing software assets and related IT assets, and part two includes specifications for tagging software to optimize its identification and management. Unfortunately, part two of the ISO/IEC 19770 standards is not universally observed, making software asset management challenging because organizations often attempt to resolve the generally accepted character based names of software utilized. Rather, software vendors employ many approaches for managing the issue, and some software vendors use inconsistent approaches for managing the issue within their products. A common theme between the many approaches includes a universally accepted alphanumeric name with zero or more additional identifiers for noting differing versions or other attributes of import. Software vendors often have their own idiosyncrasies and naming conventions, and variances in naming approaches pose a limitation for discovery tools.

[0014] Also, a characteristic of modeling tools is the possibility that accepted variances in software naming text identifiers of a given software product yields failed comparisons between actual and desired states. For example, a system modeling actual states for an asset could use a different, but valid, name for a selected software product than the system modeling the desired state. Thus, even though the selected software product is in both the actual and desired

states, software management tools may mark the configurations as a discrepancy to be addressed. In another example, the actual configuration may include a different software product than the desired configuration, but the names are so broad that the systems believe the products are identical. In still another example, the difference of one character or number out of place will yield a failed comparison between actual and desired configuration.

[0015] In one aspect, the disclosure is directed to systems and methods that facilitate propagation from actual state to desired state. The systems and methods maintain actual and desired state integrating operators that can compare the states and yield instructions to propagate from actual state to desired state when the two states are not identical. Actual state and desired state elements are mapped to tokens that are used as indices, and processed to address inconsistencies and discrepancies between the names of elements. In one example, a first list is generated identifying at least one element in an actual state and a second list is generated identifying at least one element in a desired state. Discrepancies in names of the element in the actual state and desired state are resolved to determine whether the element in the actual state is the same as the element in the desired state, such as whether multiple names describe the same product. The first list is compared to the second list to determine whether to remove the element in the actual state or whether to add the element in the desired state, or take no action. The systems can work as stand-alone asset management products or service or with other management or discovery tools.

[0016] Figure 1 illustrates an example system 100 that can be employed to propagate an actual state to a desired state. System 100 includes a name resolution engine 102, which can resolve discrepancies in name variations of the managed elements on the assets and a comparator 104 to determine whether to take action to propagate an actual state to a desired state for each asset. In some examples, system 100 can include a graphics based dashboard 106 to receive the results in a machine-readable form from the comparator 104 and allow a user to review, modify, and implement any corrective actions deemed to comply with IT policies. Additionally, the name resolution engine 102

receives information about the elements in actual and desired states from one or more asset repositories 108, which can be operably coupled to system 100, and can be operably coupled to a name resolution database 110.

[0017] An example of a repository 108 is a configuration management database. A configuration management database (CMDB) is a database that acts as a data warehouse for IT organizations. Contents of the CMDB include a collection of IT assets and configuration items as well as descriptive relationships between such assets.

[0018] System 100 can extract the configuration data from the asset repository, which can include receiving an input of the configuration data. In one example, the CMDB can include an application program interface (API) for state data access, and a direct connection can be made from the system 100 to the CMDB using a protocol, such as secure protocol, e.g., Transport Layer Security (TLS), to extract element information such as software state data of interest. In one example, element information is in textual and BASE64 encoded form and can include software name, version identifiers, update identifiers, edition identifiers, language identifiers, software edition identifiers, target platform identifiers, license version identifiers, checksum identifiers, hash identifiers, digitally signed checksum identifiers, digitally signed hash identifier, and software descriptions. If a direct API is not available, the CMDB can provide the software state data into a flat file for the system 100 to process. Configuration data can include information as to the actual and desired state of the elements on an asset, which may be obtained from the CMDB or other configuration management applications.

[0019] The name resolution engine 102 can be used to address the name variations of the software elements. In one example, the name resolution engine 102 receives data from the CMDB. The name resolution engine 102 integrates and leverages resources to identify the elements referenced in the actual and desired states. The name resolution engine 102 references name data 112, supplemental attributes and tags 114, natural language processing 116, and heuristics 118. For example, name data 112 can include the data used to describe the actual and desired state software names and data provided from

vendors of the software to indicate the recognized names of their manufactured software. Supplemental attributes and tags 114 can include data such as version identifiers, update identifiers, edition identifiers, language identifiers, software edition identifiers, target platform identifiers, license version identifiers, checksum identifiers, hash identifiers, digitally signed checksum identifiers, digitally signed hash identifiers, and the like. Natural language processing 116 may be able to parse software descriptions associated with actual or desired state software to seek equivalence or similarities between the associated descriptions. Heuristics 118 can modify features in the name resolution engine 102 used to process name resolution. For example, heuristics 118 can modify standard or initial processes applied with the name resolution engine 102 to improve performance (e.g. expanding near miss criteria, reducing the number of spaces used in actual and or desired state software names). Using one or more of the resources 112-118, the name resolution engine 102 replaces the software name with a token that can be mapped to a name in the name resolution database 110.

[0020] The comparator 104 evaluates the differences between the actual state and the desired state. In one example, the comparator 104 evaluates the differences of the software elements per each asset of interest and provides a comparison result that indicates differences or similarities between the actual state and the desired state. The comparison result can also define the actions to be taken to propagate the actual state to the desired state in cases where the actual state does not match the desired state. Additional processing, such as applying the comparison result to the asset operating system, can further implement the recommended actions.

[0021] The comparator 104 can provide the comparison result to a dashboard 106, which can include an application to present the results to users such as IT personnel. In one example, dashboard 106 can quantify risks associated with deviations noted from the actual states and the desired states and depict information in a graphical manner. Dashboard 106 can be implemented as part of the system or as a separate asset management tool. In one example, the dashboard 106 can include APIs to receive and post-process the comparison

results. In another example, comparator 104 can output a flat file for the dashboard 106 to open and process.

[0022] Figure 2 illustrates an example method 200 of the system 100. In one example, the method 200 can be implemented as a combination of hardware and programming for controlling the system 100, such as in the examples illustrated and described in Figure 8 below, to perform method 200 to propagate IT assets from actual software configurations, or actual software products installed on the IT asset, to desired software configurations. In the illustrated example, a first list is generated identifying (including having or possessing) at least one software element in an actual state and a second list is generated identifying (including having or possessing) at least one software element in a desired state at 202. In one example, the first list and second list can be generated in the name resolution engine 102, and can be provided as outputs to the comparator 104. Discrepancies in the names of the element in the actual state and desired state are evaluated to determine whether the element in the actual state is the same as the element in the desired state, such as if whether multiple names describe the same software product, at 204. In one example, the discrepancies are evaluated with the name resolution engine 102 in generating the lists for output to the comparator 104. The first list is compared to the second list, such as with the comparator 104, to determine whether to remove the element in the actual state or whether to add the element in the desired state at 206, or take no action.

[0023] Figures 3 and Figures 4 illustrate example methods of the name resolution engine 102 in resolving names and generating lists for output at 202, 204. In one example, the methods can be implemented as a combination of hardware and programming (machine instructions) for controlling the system 100, such as in the examples illustrated and described in Figure 8 below, to resolve names and generate lists for output. Figure 3 illustrates an example method 300 for processing a set of software names of elements in a first state, such as one of an actual state and a desired state. Figure 4 illustrates an example method 400 for processing a set of software names of elements in a second state, such as the other of the actual state and the desired state. In the

particular examples disclosed below, Figure 3 illustrates an example method 300 for processing a set of names of elements in the desired state and Figure 4 illustrates an example method 400 for processing a set of names of elements in the actual state. In the examples below, data structures such as arrays (or linked lists) may be indicated as written in compound words or phrases that begins with a lowercase letter and subsequent words or abbreviations begin with an uppercase letter, such as camelCase (first letter of first word in compound word lower case, subsequent first letter of other words making up the compound word upper case). Also, variables may be indicated as written in compound words or phrases such that each word or abbreviation begins with an upper case letter, such as in PascalCase (first letter of first word in compound word upper case, subsequent first letter of other words making up the compound word upper case).

[0024] Figure 3 illustrates an example method 300 to process one or more software names of software elements intended to be configured on the IT assets, or the desired state software names. A token value is created from the character string for a desired state software name at 302. In one example, the token is an integer labeled FullIndex. The variable FullIndex becomes an index into a character based array, whose value is the desired state name; such as desiredStateName[FullIndex] at 304. FullIndex is also used as an index into an integer array desiredState[FullIndex] whose values can be the integer 0 or 1, at 306. This integer array (desiredState[]) indicates whether a desired state software name has been placed into the desiredStateName array for a particular index value.

[0025] Figure 5 illustrates an example method 500 of creating a token value from the character string at 302. In one example, the method 500 can be implemented as a combination of hardware and programming for controlling the system 100, such as in the examples illustrated and described in Figure 8 below, to create a token value from the character string at 302. In the example, an integer value is generated from the character string at 502. In one example, the integer value is the sum of the integer ASCII (American Standard Code for Information Interchange) character values in the character string of the software

name. For example, the letters and spaces in the character string "ACME Spread Sheet" yields an integer sum of $(65+67+77+69)+32+(83+112+114+101+97+100)+32+(83+104+101+101+116) = 1454$. This may not yield a unique integer value from all other software names of software in the desired state, so a check is made to determine whether the integer value is unique (that is, whether that integer has been encountered before since the tokenization of the desired state names began) at 504. If the integer value is unique to the other software names of the software in the desired state tokenized so far at 506 (this would be indicated by the desiredStateName array value for the current index, FullIndex to be the NULL string), the integer value is assigned as FullIndex at 508 and FullIndex is used as an index into the desiredStateName and desiredState arrays.

[0026] Thus, using the example above applied to character array, and assuming the integer 1454 had not been encountered before, at 304 and integer array at 306, desiredStateName[1454] = "ACME Spread Sheet" and desiredState[1454] = 1.

[0027] If, however, the integer value is not unique to the other software names of the software in the desired state tokenized so far at 506 (i.e. the derived token has been observed before), the character string of the currently processed software name is compared against the character string of the previously assigned index at 510. If the character strings are the same at 512, the names are the same and desired state software name has already been processed. The method 500 is complete for the particular desired state software name at 514. This would nominally not be expected to occur as the CMDB managing the desired state would not duplicate software names.

[0028] If the character strings are different at 512, a new desired state software name is encountered whose ASCII character element sum happens to be the same as a previously processed software desired state name. The index value is modified, such as incremented by a fixed value, or clash avoidance integer, at 516. Another check is made to determine whether the integer value (now incremented by the clash avoidance integer) is unique at 504. This process can

be repeated until a string match is found at 512 (again not expected) or a unique integer value not encountered before is generated at 506.

[0029] Selection of the clash avoidance integer amount can vary by organization. In one example, the amount for the clash avoidance integer value can be a value that would be expected to generally exceed the maximum integer that could be derived by summing the integer ASCII character values making up any software name used in the organization IT environment. One method to derive this amount would be to consider the longest character string of the desired state software names and multiply that number of characters by the maximum ASCII value, or 255.

[0030] Returning to Figure 3, each uniquely encountered desired state software name, such as a desired state software name assigned a token value of FullIndex at 508, will also be assigned an additional token value, such as an integer PartialIndex, derived from a substring of the desired state software name at 308. Software names are often truncated, particularly around versioning information, so a full name of a software element such as "Software X, Rev. 2.5b" can become known as "Software X." Detecting such naming artifacts can be difficult, but an approach for doing so includes creation of additional arrays using PartialIndex as an index. For example, PartialIndex becomes a token value corresponding with the partial name or substring of the desired state software name under consideration in a character array desiredStateNamePartial[] at 310. Also, PartialIndex can be made to correspond to FullIndex in an integer array partialToFullIndex[], at 312, in which partialToFullIndex[PartialIndex] = FullIndex. This enables the system to quickly get access to the full desired state software name the partial name was derived from.

[0031] In one example, desiredStateNamePartial[] array element is indexed with PartialIndex, the ASCII sum of the first n characters of the current desired state software name under consideration and set to the first n characters of the desired state software name. The particular selected value of n, or SubStringCheckLength, can vary by organization. In one example, SubStringCheckLength can be set to one half the amount of the average

character string length of the set of desired state software names. For example, if the set of desired state software names have character string lengths of 38, 21, 27, and 11 characters, the average length of the character string is 24 characters and one half of 24 yields a SubStringCheckLength of 12 in this example.

[0032] PartialIndex can be generated in a manner similar to FullIndex in Figure 5. For example, an integer value is generated from summing the corresponding ASCII values of the first SubStringCheckLength characters in the desired state full name under consideration. A check can be made to determine whether that integer value has been used before as a previously assigned partial index value by determining whether desiredStateNamePartial[PartialIndex] has been assigned a value. If the PartialIndex integer value is unique and not previously assigned another desired state software name (this would be indicated by desiredStateNamePartial[PartialIndex] having a value of a NULL string), the integer value is assigned to be PartialIndex. If the integer value is not unique, for example, a new integer value is assigned as the integer value modified, or added, by the clash avoidance integer. For convenience or simplicity, the same clash avoidance integer used at 516 can be used to increment PartialIndex and arrive at a new value. This process can be repeated until a unique integer value not encountered before is generated.

[0033] The method 300 can be performed for each software name in the set of desired state software names. In one example, software name resolution engine 102 can process the actual state software names after method 300 has been performed for each of the desired state software names, such as for each of the desired state software names per IT asset.

[0034] Figure 4 illustrates an example method 400 to process one or more software names of software elements currently configured on the IT assets, or the actual state software names. A token value is created from the character string for an actual state software name at 402. In one example, the token is an integer labeled ActualStateIndex that can be generated from the character string of the actual state software name in manner similar to 302, such as the sum of the integer ASCII character values in the character string of the actual

state software name. ActualStateIndex is applied as an index to access the array desiredStateName[] at 404.

[0035] A character string (or null value) is returned at 406. The character string returned is compared to the character string of the actual state software name at 410. If the returned character string is identical to the character string of the actual state software name at 412, an identical match of names of the software element in the desired state and actual state has been found. If there is an identical match at 412, ActualStateIndex is the same as the FullIndex and the character strings are identical; the ActualStateIndex (corresponding FullIndex) is used as an index into an integer array actualState and its value is set to the integer 1 at 414. Accordingly, actualState[ActualStateIndex] = 1.

[0036] If, as a result of the comparison at 410, the returned character string is not identical to the character string of the actual state software name and desiredStateName[ActualStateIndex] is not initially null (in which case a match failure is immediately noted), the ActualStateIndex is incremented by the clash avoidance integer at 416, and the new ActualStateIndex is used to access desiredStateName[] at 404. This process repeats as illustrated until an identical match is found at 412 or a null string in the desiredStateName[] array is finally identified at 408 indicating a match failure, at 408.

[0037] If a match failure is identified at 408, it is uncertain whether a genuinely new software product has been encountered or if the products being compared are the same but the software name on the actual state software list is somewhat different than the desired state software list.

[0038] Figure 6 illustrates an example method 600 to process a match failure, such as match failure 408. In one example, the method 600 can be implemented as a combination of hardware and programming for controlling the system 100, such as in the examples illustrated and described in Figure 8 below, to respond to a match failure at 408. A partial character string of the actual state software name under consideration is compared to the set of partial character strings of the desired state software names in desiredStateNamePartial[] at 602. For example, the sum of the first SubStringLength ASCII values of the actual state software name

character string is calculated as a token PartialActualNameIndex, and the integer value PartialActualNameIndex is used as an index to access the desiredStateNamePartial array. If PartialActualNameIndex is able to access a character string (i.e. a non NULL value is returned), the returned character string is compared to the first SubStringCheckLength characters in the actual state software name. If these partial character strings do not match, the PartialActualNameIndex is incremented by the clash avoidance integer, and the new PartialActualNameIndex token is used to access desiredStateNamePartial[], in a manner similar to that described above in Figure 4. This process repeats until an identical match of the partial character strings is found or the token used to access desiredStateNamePartial[] returns a null value, i.e., no value is returned per the index, at 604.

[0039] A match of the partial character strings at 604 indicates that a potential name match of the actual state name and a desired state name candidate has been identified. The resources of the name resolution engine 102, including supplemental attributes and tags 114, natural language processing 116, and heuristics 118, are applied to the actual state name and the corresponding desired state name candidate identified to determine if indeed the names are the same at 606.

[0040] The supplemental attributes and tags resource 114 applies additional attribute information regarding the software that may have been obtained previously from the vendor and been associated with the software names with both the actual state and desired state name data. Some of those attributes could include version identifier, update identifier, edition identifier, language identifier, software edition identifier, target platform identifier, license version identifier, checksum identifier, hash identifier, digitally signed checksum identifier, and digitally signed hash identifier. If a direct match can be found between a license version identifier, a checksum identifier, a hash identifier, a digitally signed checksum identifier, or a digitally signed hash identifier within the candidates then a match between the actual state software name and the desired state software name has been found.

[0041] If a match between one or more of the version identifier, update identifier, edition identifier, language identifier, software edition identifier, or target platform identifier of the candidates is found, then it is likely a match between the actual state software name and the desired state software name has been found, but not certain. In those cases additional verification steps can be performed, such as the near miss test described below.

[0042] The natural language processing resource 116 can be used to process descriptions available along with the naming and attribute/tag data. For example, the descriptions associated with desired state software name candidate and the actual state software are processed to determine if the descriptions are similar. In one example, the actual software state name "ACME Super Spread Sheet" under consideration has an associated software description string stating "This is Revision 4.5b of ACME Super Spread Sheet with the latest enhancements for graphing and visualization of cell data," is compared with a desired state name of "ACME Super Spread Sheet, Revision 4.5b" that has an associated software description string stating "This is Revision 4.5b of ACME Super Spread Sheet with the latest enhancements for graphing and visualization of cell data." A partial string match will be detected at 604 because the software description strings identically match up to the word "Sheet", the natural language processing of the software descriptions with resource 116 will detect equivalence at 606, and the actual software state name under consideration will be matched the desired state name "ACME Super Spread Sheet, Revision 4.5b." In this example, the software descriptions are identical, but organizations can establish an equivalence threshold for likely encounters with non-identical descriptions. The equivalence amount can be quantified, such as 98% equivalence, 30% equivalence, or the like, and that value compared to a threshold. For example, an equivalence amount of 90% or more can be chosen to indicate a potential name match between the actual state software name and the desired state software name candidate. Organizations can vary the quantified equivalence amount via the heuristics resource 118. Further, the heuristics resource 118 can be applied to remove spaces, or to limit spaces to one space between characters, in the both the

actual state and desired state software name character strings before re-executing the entire matching algorithm, if the other algorithm measures described have not yet produced a match.

[0043] If the names are determined to not be the same at 608 after the resources are applied at 606, a new software product likely has been encountered, i.e., that new software product is currently configured on an IT asset but is not included in the set of products intended to be configured on the assets. It would also be possible to run the near miss test after the match failure, but generally this will not yield better results. The new software product name is processed at 610. For example, ActualStateIndex from method 400 (the ASCII sum of the full name character string plus any multiples of one or more clash avoidance integers) is used as an index into the actual state software name character array actualStateName[] and ActualStateIndex is used as an index to set actualState[ActualStateIndex] with the integer value of 1.

[0044] If, however, the names are determined to be the same at 608 after resources are applied at 606, the software product has been encountered in processing the desired states software elements. The identified product can be processed at 612. For example, FullIndex of the desired state name candidate determined to be a match with the actual state name is set to correspond with integer 1, i.e. actualState[FullIndex] = 1. In this example, the FullIndex value is used to correspond with the actual state software name instead of the ActualStateIndex, because the software name referenced with ActualStateIndex in the actualStateName array does not correspond with a desired state software name as determined in process 400.

[0045] Returning to 604, the actual state name and the ActualStateIndex are processed with a near miss test at 614 in the example if the partial character strings do not match at 604. For example, the near miss test at 614 includes an examination of tokens near the value of the ActualStateIndex token, against a set of tokens used to populate the desiredStateName array. For instance, the actual state software name can be compared to a set of desired state software name candidates having a character string sum comparable to a fraction of the character string sum of the of the actual state software name under

consideration, such as 10% of the character string sum of the actual state software name. The fraction, or percentage, applied to determine a threshold similitude in the near miss test at 614 can vary by organization, and be later adjusted by history.

[0046] As an example, a desired state software name “Sam’s Scientific Calculator” has been processed and includes FullIndex (and character string sum) of 2566. The actual state software name “Sam Scientific Calculator” (missing the “s”) is under consideration to determine if it is a match to a desired state software name or a new product not in the set of desired state software products. The sum of its ASCII characters for “Sam Scientific Calculator” is 2412, which in the example returns a null value when used to access desiredStateName[] array and thus would return a match failure at 408. Further, because the two partial strings diverge after the third character, any SubStringCheckLength greater than three will not yield a partial string match at 604. In this example, however the desired state name “Sam’s Scientific Calculator” will be selected as a near-miss compare candidate because 10% of actual state software character string sum 2412 is 241, well within the range of the “Sam’s Scientific Calculator” index of 2566.

[0047] The near miss test at 614 develops a set of desired state software name candidates having a character string sum comparable to the selected fraction of the character string sum of the actual state software name. The resources of the name resolution engine 102, including supplemental attributes and tags 114, natural language processing 116, and heuristics 118, are applied to the actual state name and the corresponding desired state name candidates to determine if indeed the names are the same in a manner similar to 606. For the example mentioned in the prior paragraph, the actual state software name “Sam Scientific Calculator” can include a supplemental attribute entity associated with it, i.e., a digitally signed checksum <checksum_ssc>, and the desired state software name “Sam’s Scientific Calculator” can include a supplemental attribute entity associated with it, i.e., a digitally signed checksum <checksum_ssc>. The digitally signed checksums would be matched similar to 606 and the actual and desired state software names would be considered to be

product matches, despite the actual state software management system does not have the “s” on the word “Sam” in the string “Sam Scientific Calculator”.

[0048] If the candidates are determined not to be the same at 616 after the resources are applied in the near miss test 614, more tokens could be gathered for the name resolution engine 102 to examine by expanding the initial percentage used to collect initial near miss token candidates. In the example above, 10% was initially used, which could be modified to 20% or some other percentage, for example. Above a selected percentage additional meaningful tokens may not be identified above a threshold percentage. This threshold percentage will likely vary per organization depending on the software names used. If the near miss tests yield no matches, a new software product has been encountered, i.e., that new software product is currently configured on an IT asset but is not included in the set of products intended to be configured on the assets. The new software product name is processed at 610. For example, ActualStateIndex from method 400 (the ASCII sum of the full name character string plus any multiples of one or more clash avoidance integers) is used as an index into both the actual state software name in character array actualStateName[], and the actualState[] integer array. The string from actualStateName[ActualStateIndex] is set to the current actual state software name being processed and actualState[ActualStateIndex] is set to the integer 1.

[0049] If, however, a name from the desired state candidates is determined to be the same as the actual state software name at 616 after resources are applied in the near miss test at 614, the software product has been encountered in processing the desired states software elements. The identified product can be processed at 612. For example, FullIndex of the desired state name candidate determined to be a match with the actual state name is used as an index to both the actualStateName character array and the actualState integer arrays, and actualStateName[FullIndex] = desiredStateName[FullIndex] and actualState[FullIndex] is set to with a value of the integer 1.

[0050] Figure 7 illustrates a method 700 to evaluate the differences between the actual state and the desired state, such as at 206, that can be performed using the comparator 104. In one example, the method 700 can be implemented as a

combination of hardware and programming for controlling the system 100, such as in the examples illustrated and described in Figure 8 below, to evaluate the difference between the actual state and the desired state at 206. In one example, method 700 can be performed after the name resolution operations of Figures 3 and 4 have transformed the repository 108 data, such as data in a CMDB state, into a set of tokens representing the actual and desired states of software on an asset. The actual state and desired state are retrieved in a data structure at 702. For example, information as to whether a Token, such as token, which can represent a software element mapped and tracked in the name resolution database 110, can be retrieved for each IT asset, such as a computing device, or other category of asset. In one example, the set of all tokens Token includes the set of all FullIndex and ActualStateIndex tokens. For each asset, an evaluation is performed to determine whether the actual state matches the desired state, so that no action is taken to propagate the actual state to the desired state, at 704. For example, if an evaluation determines a software element is not configured on an IT asset and the software element is not intended to be configured on the IT asset, no action is taken. Similarly, if an evaluation determines a software element is configured on an IT asset and the software element is intended to be configured on the IT asset, no action is taken. For discrepancies between the actual state and the desired state, the condition of the discrepancy is evaluated to determine whether to add the element or remove the element, at 706. For example, if an evaluation determines a software element is not configured on an IT asset and the software element is intended to be configured on the IT asset, the software element is installed on the IT asset. If an evaluation determines a software element is configured on an IT asset and the software element is not intended to be configured on the IT asset, the software element is removed from the IT asset.

[0051] In one example of method 700, the data structure at 702 includes arrays. For example, the actual state data and desired state data can be stored in one dimensional arrays. The size of the arrays can be the total number of the software elements the name resolution database 110 maps and tracks. In one

example, the array holds the total number of software elements on the given IT asset, which implies the array is relatively sparse.

[0052] Two arrays are retrieved for each computer asset, e.g. `actualState[]` and `desiredState[]`. The array elements for any index, such as `Index`, may have an integer value of 0 or 1. A value of 1, as determined above in the name resolution operations, for `Index` in `actualState[]` (`actualState[Index] = 1`) indicates the software element is in the actual state, or configured on the IT asset. A value of 0 for `Index` in `actualState[]` (`actualState[Index] = 0`) indicates the software element is not in the actual state, or not configured on the IT asset. A value of 1, as determined above in the name resolution operations, for `Index` in `desiredState[]` (`desiredState[Index] = 1`) indicates the software element is in the desired state, or is intended to be configured on the IT asset. A value of 0 for `Index` in `desiredState[]` (`desiredState[Index] = 0`) indicates the software element is not in the desired state, or not intended to be configured on the IT asset.

[0053] A subtraction operation can be performed on the arrays for each IT asset, and the result of the subtraction operation for each element Index can be stored in an integer array resultState[]. For example for a value Index, $\text{resultState[Index]} = \text{actualState[Index]} - \text{desiredState[Index]}$. The possible values for these elements are depicted in the table below.

actualState[Index]	desiredState[Index]	resultState[Index]
0	0	0
0	1	-1
1	0	1
1	1	0

[0054] The resultState[] for an Index can include a -1, 0, or 1. If the resultState[] for an Index includes a 0 after subtraction, the actual state and desired state for the software element match, and no action is taken, at 704.

[0055] If the resultState[] for an Index does not include a 0 after subtraction, the actual state and desired state for the software element do not match, and action is to be taken to install or uninstall the software element on the IT asset at 706. For any element Index in resultState[] that is a -1, the software element is to be added, or installed on the IT asset. For any element Index in the resultState [] that is 1, the software element is to be removed, or uninstalled from the IT asset.

[0056] In one example, the operating system of the IT asset can be employed to accomplish the installation or removal of the software element to propagate the actual state to desired state. For an index Index, if resultState[Index] is -1 indicating the software element is to be added, the associated software name can be found by using Index as an index to the large desired state character array such as desiredStateName[]. The resultant name can be submitted to the operating system install application, where it may be used to obtain the software from an organization software store. If resultState[Index] is 1, Index can be used as an index into the large actual state character array such as actualStateName[] to obtain the actual name of the software to be removed. The resultant name can be submitted to the host operating system uninstall application. The results of the installation and removal processes can be written

to a log file and digitally signed so that a definitive record of the correction of the system to align the actual and desired states is maintained.

[0057] In another example of method 700, linked lists can be employed as the data structure. In this example, a desiredState linked list includes nodes containing the tokens that correspond to software elements in the desired state, such as FullIndex, for the IT asset. The actual state linked list includes nodes having the tokens that correspond to software elements in the actual state for the IT asset. In one example, the actualState and desiredState linked lists include the tokenized software elements arranged from smallest to largest integer values. After a comparison of the linked lists, up to two new linked lists can be created including an addList linked list and a removeList linked list. The addList linked list represents software elements to be added to the IT asset to propagate the actual state of the IT asset to the desired state. The removeList linked list represents software elements to be uninstalled from the IT asset to propagate the actual state to the desired state.

[0058] An example process for comparing the linked lists in pseudo C code includes the following, where the function HEAD() returns the first node in the linked list, and the function NEW() creates a new linked list node:

```
desiredStatePtr = HEAD(desiredState)
actualStatePtr = HEAD(actualState)
addListPtr = NULL
removeListPtr = NULL
While( desiredStatePtr != NULL AND actualStatePtr != NULL)
{
    If ( desiredStatePtr->element != actualStatePtr->element )
    {
        if (actualStatePtr->element >desiredStatePtr->element)
        { /* add desired state element to actual state
            Put on add list */
            addListElementPtr=NEW(addList)
            addListElementPtr->element = desiredStatePtr->element
            desiredStatePtr = desiredStatePtr->next
            if (addListPtr == NULL)
            { /* first element on add list */
                addListPtr = addListElementPtr
                tailAddListPtr = addListPtr
            }
        }
        else
        { /* add list already exists just add element to it*/
            tailAddListPtr->next = addListElementPtr
            tailAddListPtr = tailAddListPtr->next
        }
    }
}
```

22

```

else
{ /* identified actual state element not in desired
state, put the element on the remove list */
removeListElementPtr = NEW(removeList)
removeListElementPtr->element = actualStatePtr->element
actualStatePtr = actualStatePtr->next
if (removeListPtr == NULL)
{ /* first element on remove list */
removeListPtr = removeListElementPtr
tailRemoveListPtr = removeListPtr
}
else
{ /* remove list already exists, just add element to it */
tailRemoveListPtr->next = removeListElementPtr
tailRemoveListPtr = tailRemoveListPtr->next
}
}
}
else
{ /* desired state element matches actual state element, move
to next elements */
desiredStatePtr = desiredStatePtr->next
actualStatePtr = actualStatePtr->next
}
}
/* if actual state list empty but still remaining desired */
/* state elements on desired state list, put those remaining */
/* elements on add list */
while (desiredStatePtr != NULL)
{
addListElementPtr = NEW(addList)
addListElementPtr->element = desiredStatePtr->element
desiredStatePtr = desiredStatePtr->next
if (addListPtr == NULL)
{ /* first element on add list */
addListPtr = addListElementPtr
tailAddListPtr = addListPtr
}
else
{ /* add list already exists just add element to it */
tailAddListPtr->next = addListElementPtr
tailAddListPtr = tailAddListPtr->next
}
}
/* similarly if desired state list empty but still remaining */
/* actual state elements, place actual state elements on */
/* remove list */
while (actualStatePtr != NULL)
{
removeListElementPtr = NEW(removeList)
removeListElementPtr->element = actualStatePtr->element
actualStatePtr = actualStatePtr->next
if (removeListPtr == NULL)
{ /* first element on remove list */
removeListPtr = removeListElementPtr
tailRemoveListPtr = removeListPtr
}
else

```

23

```
    { /* remove list already exists, just add element to it */  
      tailRemoveListPtr->next = removeListElementPtr  
      tailRemoveListPtr = tailRemoveListPtr->next  
    }  
  }
```

[0059] Generally, in the example process, if the current actualState and current desiredState node match, then the actual state and the desired state for the element match, and the next node can be evaluated. If the current actualState node is greater than the current desiredState node, the IT asset is missing the software element represented by the current desiredState node. Thus, the current desiredState node is moved to the addList linked list, and the next node in the desiredState linked list is compared to the current actualState node. If the current actualState node is less than the current desiredState node, the corresponding software element in the actual state list is not intended to be configured on the IT asset. Thus, the current actualState node is moved to the removeList linked list, and the next node in the actualState linked list is compared to the current desiredState node. This is repeated until the actual state and desired state lists are traversed. If the actualState linked list is completely traversed before the end of the desiredState linked list is reached; then the remaining desiredState linked list elements are placed on the addList linked list. If the desiredState linked list is completely traversed before the end of the actualState linked list is reached, then the remaining actualState linked list elements are placed on the removeList linked list.

[0060] Process 700 implemented with linked lists as described in the examples can be relatively storage efficient, but can employ more computational effort than if process 700 was implemented with the arrays as described in the examples. The determination of the data structure used to implement method 700 can depend on features available on the computing platform being utilized. For example, the linear arrays may be preferred on a computer platform that had linear array acceleration hardware or software optimization of sparse linear arrays. Linked lists may be preferred on a computer platform that had generalized processing capabilities but limited memory.

[0061] System 100 includes name resolution engine 102, comparator 104, and dashboard 106 to implement example methods 200, 300, 400, 500, 600, 700, which may be any combination of hardware and programming to implement the functionalities of the example systems and methods. Such combinations of hardware and programming may be implemented in a number of different ways. For example, the programming for the system 100 and methods 200, 300, 400, 500, 600, 700 may be processor executable instructions stored on at least one non-transitory machine-readable storage medium, and the hardware for the engines may include at least one processing resource to execute those instructions. In some examples, the hardware may also include other electronic circuitry to at least partially implement at least one feature of system 100 and methods 200, 300, 400, 500, 600, 700. In some examples, the at least one machine-readable storage medium, such as a memory device, may store instructions that, when executed by the at least one processor, at least partially implement some or all features of system 100 and methods 200, 300, 400, 500, 600, 700. In such examples, system 100 may include the at least one machine-readable storage medium storing the instructions and the at least one processing resource to execute at least one of the methods 200, 300, 400, 500, 600, 700. In other examples, the functionalities of any engines of system 100 and methods 200, 300, 400, 500, 600, 700 may be at least partially implemented in the form of electronic circuitry.

[0062] Figure 8 illustrates an example computer system that can be employed in an operating environment and used to host or run computer programming in the form of a computer application 820 implementing an example method 200, and other methods of this disclosure, as included on one or more computer readable storage mediums storing computer executable instructions for controlling the computer system, such as a computing device, to perform a process. In one example, the computer system of Figure 8 can be used to implement modules and associated tools of application 820 corresponding with system 100, namely name resolution engine 822, comparator 824, and dashboard 826 including the functionalities of name resolution engine 102, comparator 104, and dashboard 106.

[0063] The exemplary computer system of Figure 8 includes a computing device, such as computing device 800. Computing device 800 typically includes one or more processors 802 and memory 804 for storing and executing application 820. The processors 802 may include two or more processing cores on a chip or two or more processor chips. In some examples, the computing device 800 can also have one or more additional processing or specialized processors (not shown), such as a graphics processor for general-purpose computing on graphics processor units, to perform processing functions offloaded from the processor 802. Memory 804 may be arranged in a hierarchy and may include one or more levels of cache. Memory 804 may be volatile (such as random access memory (RAM)), non-volatile (such as read only memory (ROM), flash memory, etc.), or some combination of the two. The computing device 800 can take one or more of several forms. Such forms include a tablet, a personal computer, a workstation, a server, a handheld device, a consumer electronic device (such as a video game console or a digital video recorder), or other, and can be a stand-alone device or configured as part of a computer network, computer cluster, cloud services infrastructure, or other.

[0064] Computing device 800 may also include additional storage 808. Storage 808 may be removable and/or non-removable and can include magnetic or optical disks or solid-state memory, or flash storage devices. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any suitable method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. For example, application 820 can be stored in storage 808, and at least one or more components of application 820 can be loaded and stored into memory 804 for execution on processor(s) 802. A propagating signal by itself does not qualify as storage media.

[0065] Computing device 800 often includes one or more input and/or output connections, such as USB connections, display ports, proprietary connections, and others to connect to various devices to receive and/or provide inputs and outputs. For example, computing device 800 may be coupled to an asset repository 108, such as configuration management database, and to a name

resolution database 110. Input devices 810 may include devices such as keyboard, pointing device (e.g., mouse), pen, voice input device, touch input device, or other. Output devices 812 may include devices such as a display, speakers, printer, or the like. In one example, an output device 812 such as a display can be configured to render dashboard 826, which corresponds with the dashboard 106.

[0066] Computing device 800 often includes one or more communication connections 814 that allow computing device 800 to communicate with other computers/applications 816 such as with the operating system of the IT asset or other applications to install or remove an identified software element to propagate the actual state to desired state. Example communication connections can include, but are not limited to, an Ethernet interface, a wireless interface, a bus interface, a storage area network interface, a proprietary interface. The communication connections can be used to couple the computing device 800 to a computer network 818, which is a collection of computing devices and possibly other devices interconnected by communications channels that facilitate communications and allows sharing of resources and information among interconnected devices. Examples of computer networks include a local area network, a wide area network, the Internet, or other network.

[0067] Although specific examples have been illustrated and described herein, a variety of alternate and/or equivalent implementations may be substituted for the specific examples shown and described without departing from the scope of the present disclosure. This application is intended to cover any adaptations or variations of the specific examples discussed herein. Therefore, it is intended that this disclosure be limited only by the claims and the equivalents thereof.

CLAIMS

What is claimed is:

1. A method to propagate from actual state to desired state, the method comprising:

generating a first list identifying at least one element in an actual state and a second list identifying at least one element in a desired state;

resolving a discrepancy in a name of the element in the actual state with a name of the element in the desired state to determine whether the element in the actual state is the same as the element in the desired state; and

comparing the first list to the second list to determine whether to remove the element from the actual state and whether to add the element in accordance with the desired state.

2. The method of claim 1 wherein resolving includes generating a plurality of indices that include tokens corresponding with a name of an element in a desired state, the indices including a full index and a partial index, wherein the full index is included in a desired state list.

3. The method of claim 2 wherein resolving includes adding a full tokenized value of a name of an element in an actual state to an actual state list if the full tokenized value corresponds with the full index.

4. The method of claim 3 wherein resolving includes adding the full index to the actual state list if the name of the element in the actual state is determined to be the same as the name of the element in the desired state after applying a partial tokenized value of the name of the element in the actual state to the partial index.

5. The method of claim 4 wherein resolving includes adding the full tokenized value to the actual state list if the name of the element in the desired state is not the same as the name of the element in the actual state after applying a partial tokenized value of the name of the element in the actual state to the partial index.
6. The method of claim 2 wherein resolving includes applying at least one of attribute information, natural language processing, and heuristics.
7. The method of claim 1 wherein comparing includes one of installing and uninstalling discrepancies of state.
8. A computer readable medium for storing computer executable instructions for controlling a computing device to perform a method to propagate from actual state to desired state, the method comprising:
 - generating a first data structure identifying at least one element in an actual state and a second data structure identifying at least one element in a desired state via creating first values from character strings of the at least one element in the actual state and the at least one element in the desired state;
 - resolving a discrepancy in a name of the element in the actual state with a name of the element in the desired state to determine whether the element in the actual state is the same as the element in the desired state via creating second values from partial character strings of the at least one element in the actual state and the at least one element in the desired state; and
 - comparing the first data structure to the second structure to determine whether to remove the element from the actual state and whether to add the element in accordance with the desired state.
9. The computer readable medium of claim 8 wherein the first data structure and second data structure are linear arrays.

10. The computer readable medium of claim 9 wherein values in the first data structure is subtracted from the second data structure and a difference is included in a result linear array.
11. The computer readable medium of claim 8 wherein the first data structure and second data structure are linked lists ordered from smallest to largest integer tokens in each of the first data structure and the second data structure.
12. The computer readable medium of claim 11 wherein nodes of the first data structure and second data structure are sequentially traversed to determine which is greater.
13. A system to propagate from actual state to desired state, the system comprising:
a processor; and
memory comprising instructions executable by the processor to:
generate a first list identifying at least one element in an actual state and a second list identifying at least one element in a desired state in a name resolution engine referencing name data;
resolve a discrepancy in a name of the element in the actual state with a name of the element in the desired state to determine whether the element in the actual state is the same as the element in the desired state in the name resolution engine referencing supplemental attributes and tags; and
compare the first list to the second list to determine whether to remove the element from the actual state and whether to add the element in accordance with the desired state.
14. The system of claim 13 wherein the data regarding the at least one element in the actual state is received from a configuration management database.

15. The system of claim 13 wherein data regarding the first list and the second list is received from a name resolution database.

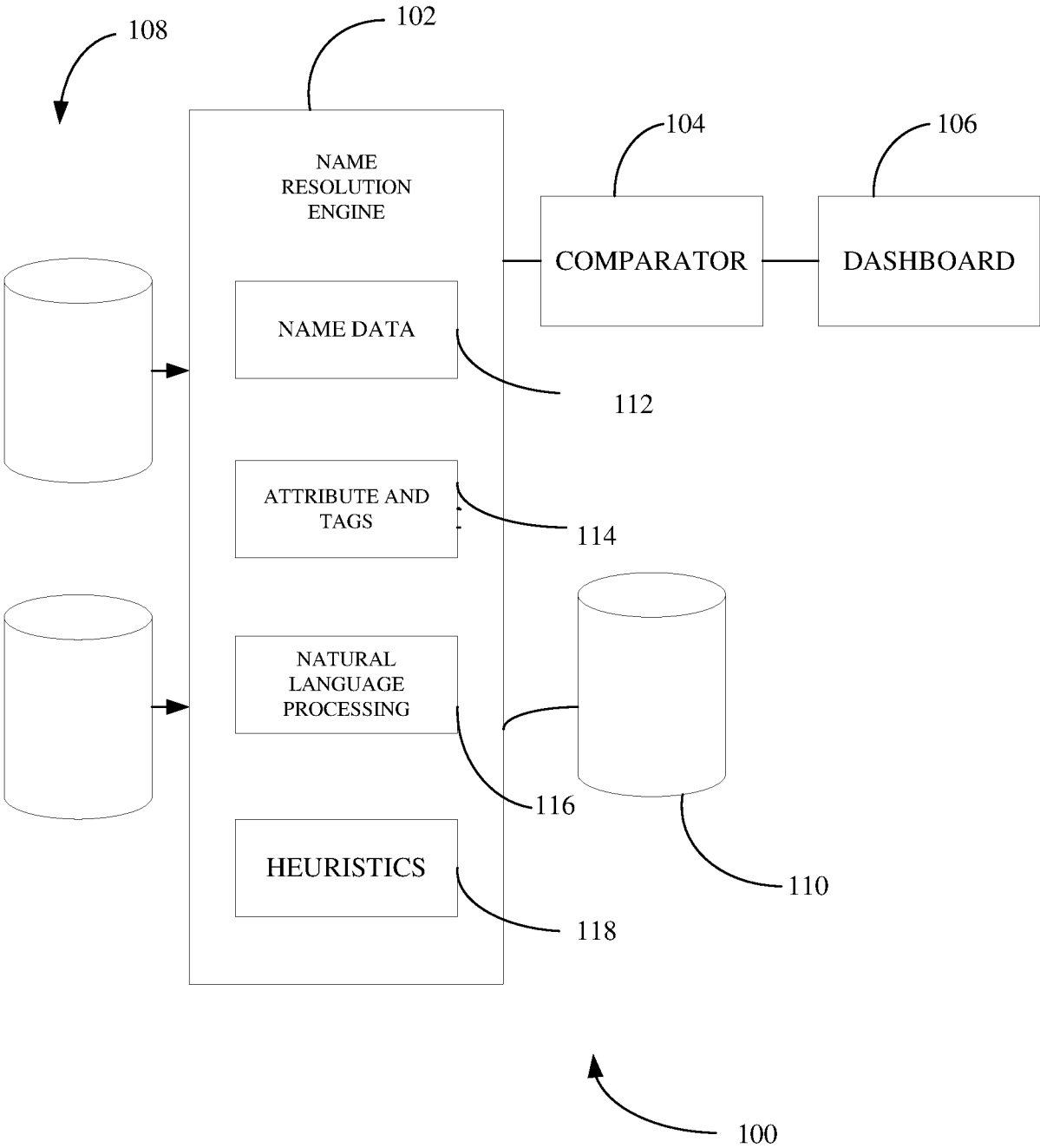


Fig. 1

2/7

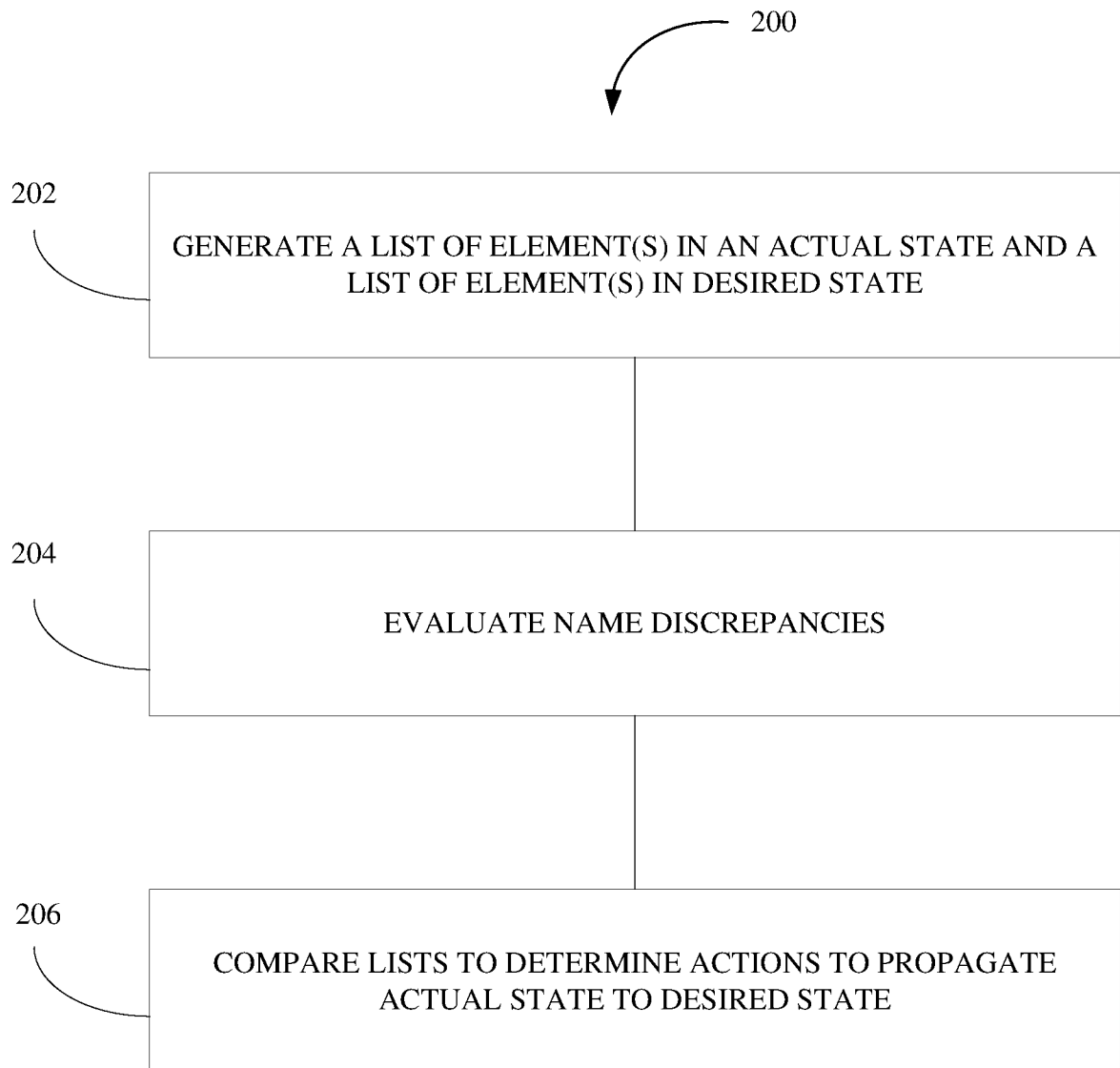


Fig. 2

3/7

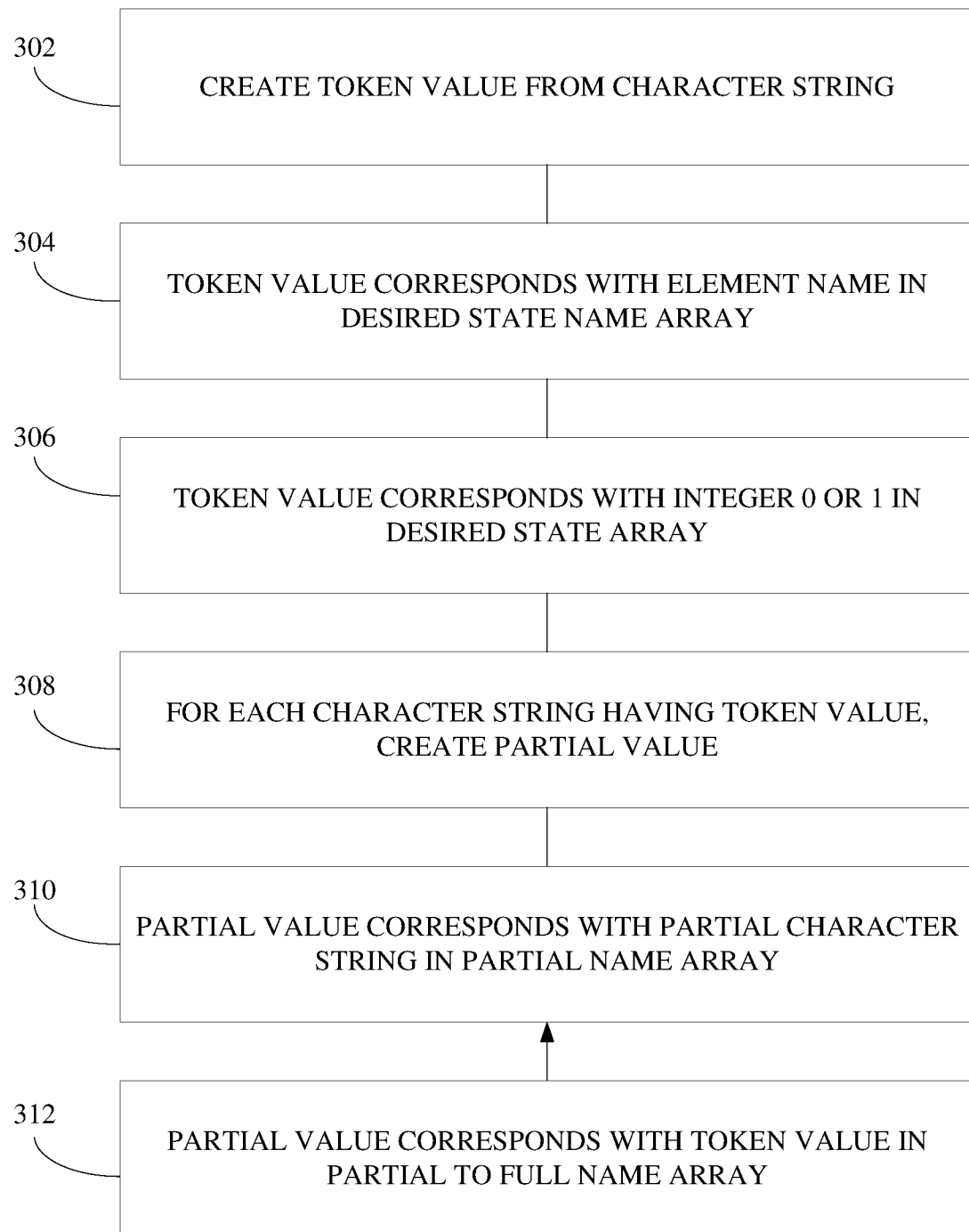


Fig. 3

300

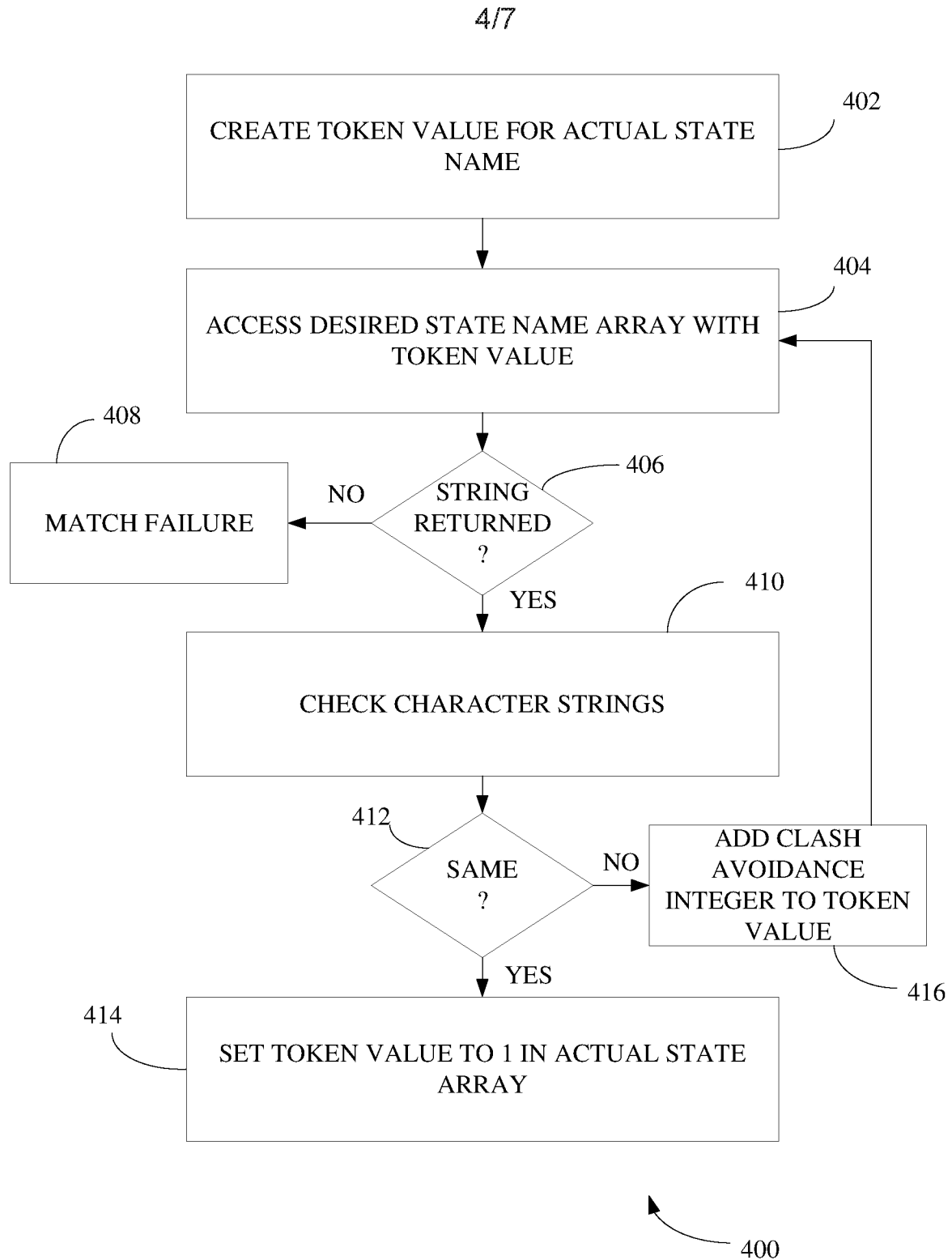


Fig. 4

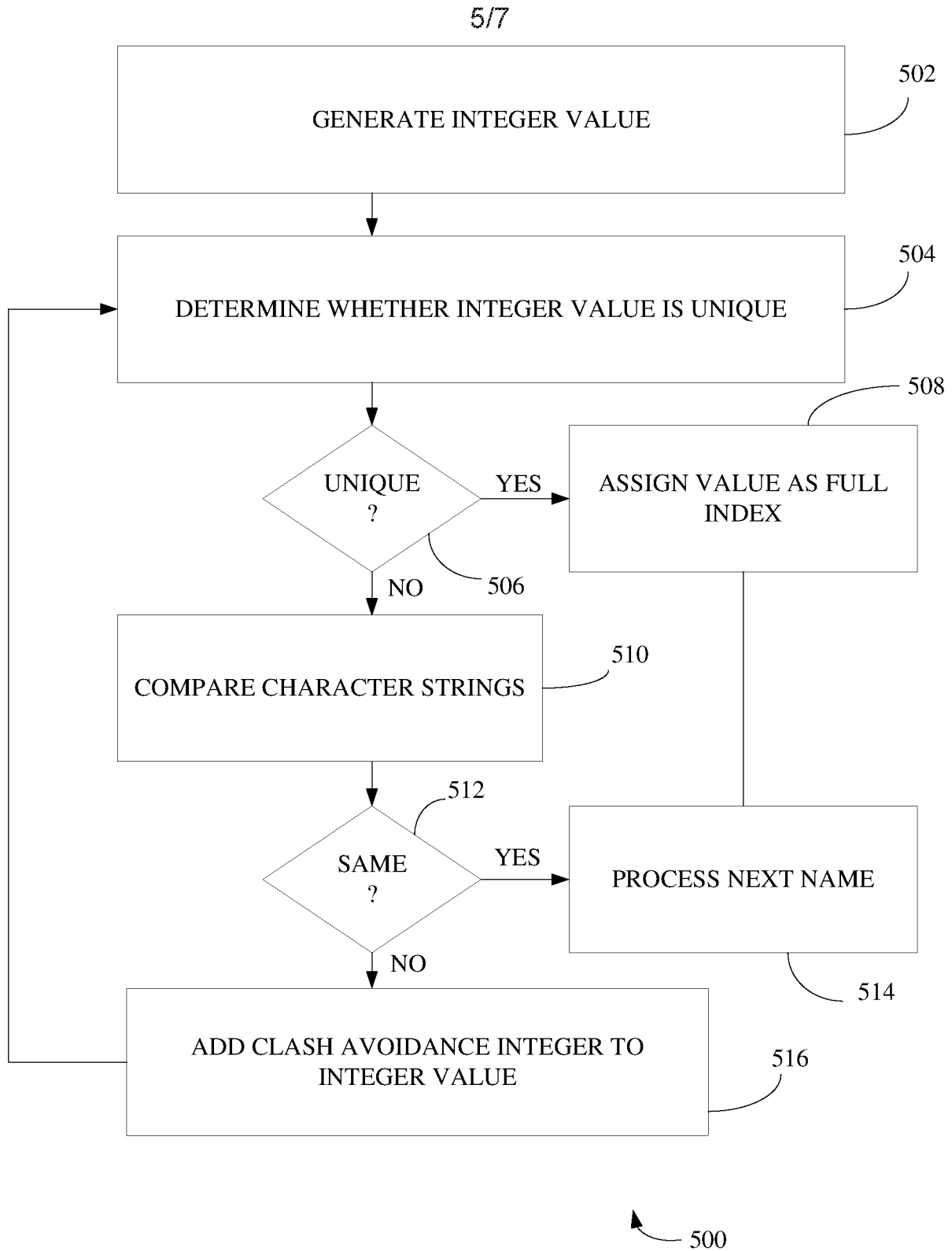


Fig. 5

6/7

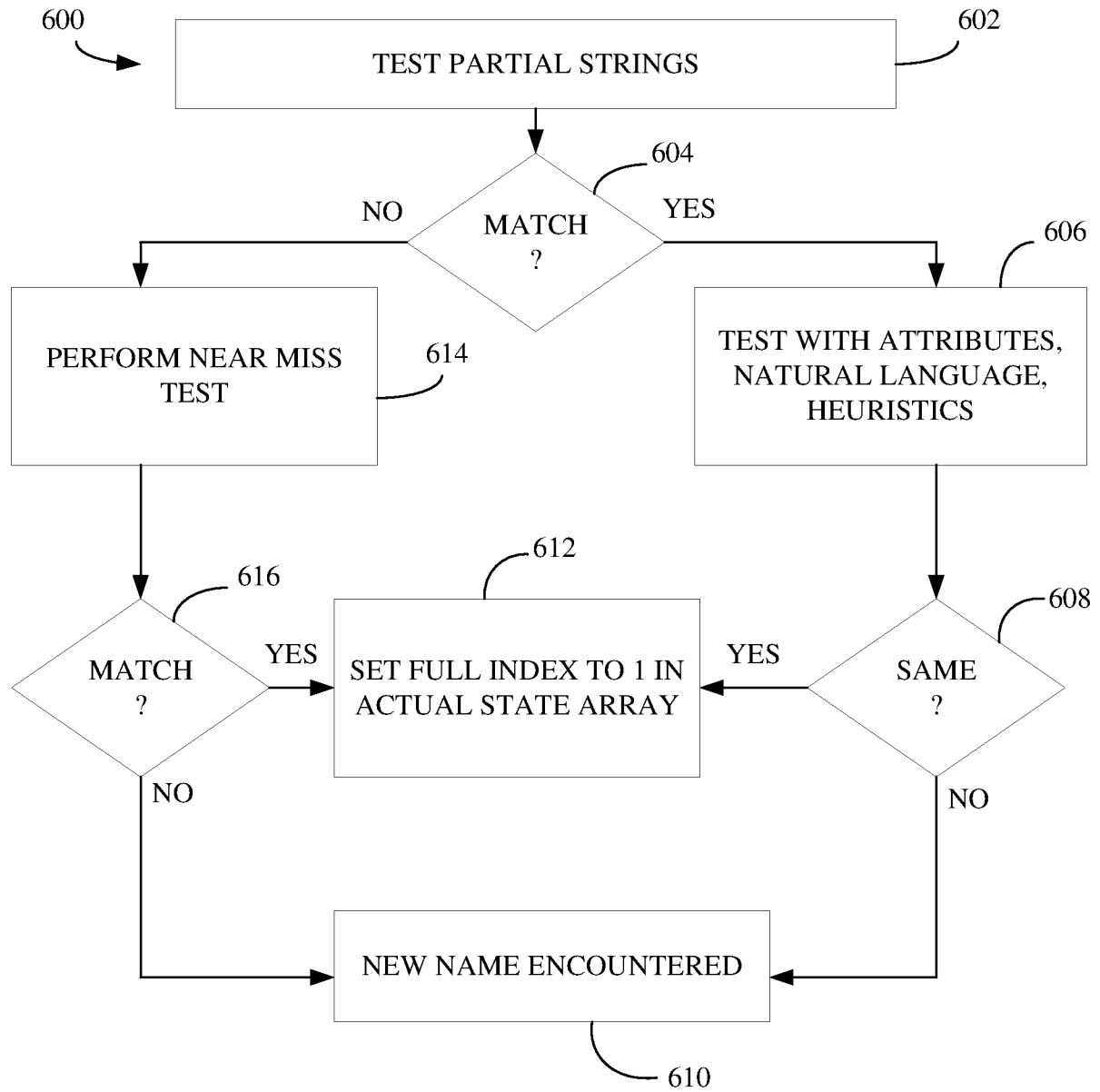


Fig. 6

7/7

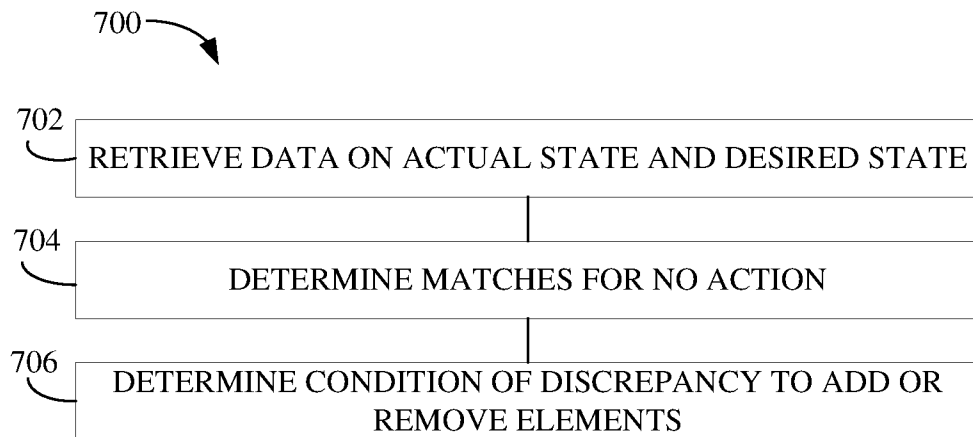


Fig. 7

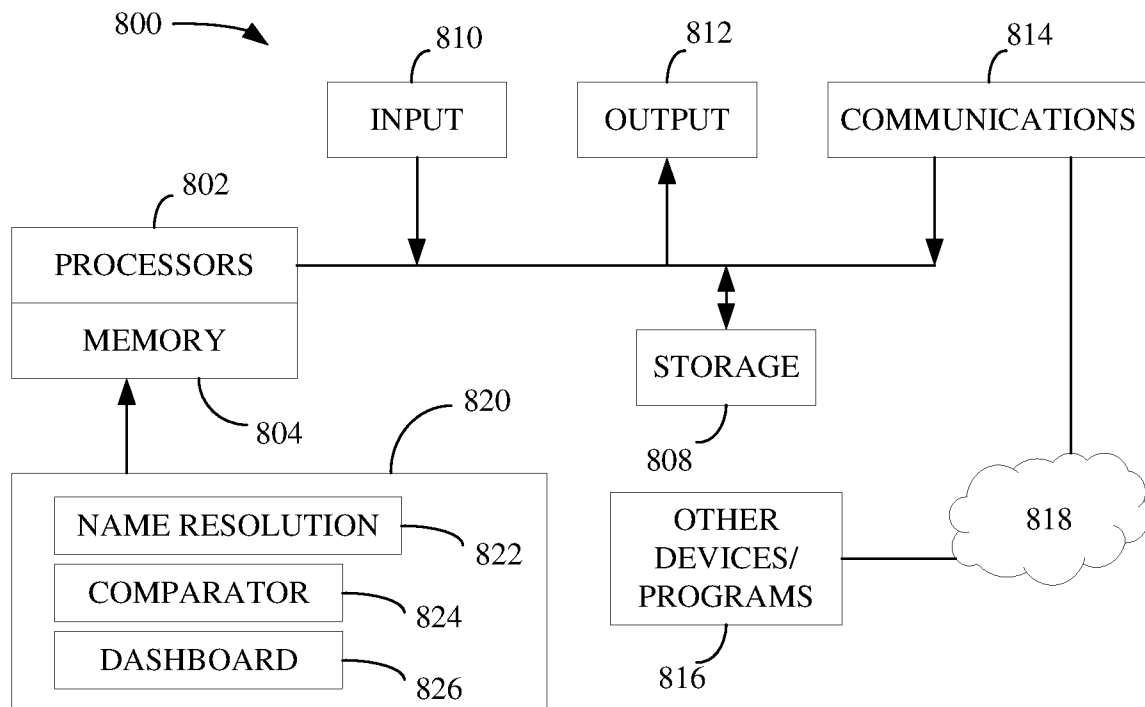


Fig. 8

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/014937**A. CLASSIFICATION OF SUBJECT MATTER****G06Q 10/06(2012.01)i, G06Q 10/10(2012.01)i**

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06Q 10/06; G06F 15/173; G06F 17/00; G06F 9/44; G06F 17/30; G06Q 30/00; G06Q 10/00; G06Q 10/10

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

eKOMPASS(KIPO internal) & Keywords: actual state, desired state, propagate, name, discrepancy, index

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 9009324 B2 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 14 April 2015 See abstract, claims 1-9 and figures 4A-5C, 7A.	1-15
A	US 8095573 B2 (UMESH MADHAV APTE et al.) 10 January 2012 See abstract, claims 1-5 and figures 5-6.	1-15
A	US 7926031 B2 (YASSINE FAIHE et al.) 12 April 2011 See abstract, claim 1 and figure 1.	1-15
A	US 7499865 B2 (VIJAY KUMAR AGGARWAL et al.) 03 March 2009 See abstract, claim 1 and figures 3-6.	1-15
A	US 2012-0221507 A1 (JAMIE LAFLEN et al.) 30 August 2012 See abstract, claim 1 and figures 1-3.	1-15



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

20 October 2016 (20.10.2016)

Date of mailing of the international search report

20 October 2016 (20.10.2016)

Name and mailing address of the ISA/KR

International Application Division

Korean Intellectual Property Office

189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea

Facsimile No. +82-42-481-8578

Authorized officer

KANG, Min Jeong

Telephone No. +82-42-481-8131



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2016/014937

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 9009324 B2	14/04/2015	US 2012-0096163 A1 US 2014-0143211 A1 US 8645543 B2	19/04/2012 22/05/2014 04/02/2014
US 8095573 B2	10/01/2012	US 2009-0019089 A1 US 2012-0117039 A1 US 8671122 B2	15/01/2009 10/05/2012 11/03/2014
US 7926031 B2	12/04/2011	EP 1983451 A1 US 2008-0263084 A1	22/10/2008 23/10/2008
US 7499865 B2	03/03/2009	CN 1831854 A US 2006-0178953 A1 US 2009-0099942 A1 US 2011-0214009 A1 US 8311991 B2 US 8606659 B2	13/09/2006 10/08/2006 16/04/2009 01/09/2011 13/11/2012 10/12/2013
US 2012-0221507 A1	30/08/2012	None	