Specification Fragments with Defined Semantics to Support SW-Evolution

Andreas Bollin
Institute for Informatics-Systems
University of Klagenfurt, Austria
Andreas.Bollin@uni-klu.ac.at

Roland T. Mittermeir
Institute for Informatics-Systems
University of Klagenfurt, Austria
Roland.Mittermeir@uni-klu.ac.at

Abstract

The power of formal specifications is not fully exploited if used only during initial development. However, the linguistic density of specification languages can be seen as an obstacle against reading and easily comprehending a document written by other engineers.

This paper introduces an approach for identifying fragments of specifications with well defined semantic content. Specification chunks and specification slices are introduced as complementary concepts to convey partial, though sufficient understanding to maintainers or other focussed readers.

INTRODUCTION

Formal software specifications are usually recommended as means to produce high quality software. This focus on software production is usually understood to refer to the initial development of software. Hence, following traditional life cycle models, the specification phase, if it takes place at all, is placed very early in the agenda.

This seems unjustified in so far, as specifications could also be used as important driver for test data generation [1]. But they could as well play an vital role during SW-maintenance. The later, however, requires specifications to be kept up to date during the various evolutionary steps taking place already during system development, less to say later, during operations. To keep specifications constantly up to date, requires effort. This effort is justified only, if additional benefits can be reaped from using this high level product in conjunction with code to improve the maintenance process, be it by speeding up SW-comprehension, or be it by assuring higher quality of the maintained product.

As argued in [6] the density of expressing thoughts, which is considered to be a positive attribute during development (for the specification's writer), becomes detrimental for specification comprehension during later phases. This makes managers claim that formal specifications do not scale up and this makes software developers look down on specification languages by referring to them as "write-only"-languages.

To address this criticism, we looked for mechanisms that allow focussed reading and browsing of specifications. The aim of this research is to limit the part of a specification a maintainer needs to understand when trying to resolve some specific question popping up in a change request. The benefits reaped should not only be higher professionalism with respect to high quality specifications, but also better assessment of the effort involved with an incoming change request, as well as higher quality of the maintenance process itself.

To reach this goal, one cannot change the inherent density of specification languages. One has to consider though, that comprehension problems are compounded if problem-inherent complexity is combined with complexity of size [5]. The latter could be reduced, though, if a formal mechanism can be devised which ensures that the peruser of a specification is presented only the portion of the specification relevant to the particular problem at hand.

In code comprehension, slices [11] and chunks [2] have been proposed as alternative mechanisms to identify well defined portions of code. Depending on the problem at hand, a peruser can rest assured that if s/he analyzes the respective portion of code, all that needs to be studied for the problem at hand has been considered. In [6] we focussed on the differences between these concepts on the specification- and on the code level. Here, we elaborate on the issue how such "complete" chunks or even slices of specifications can be defined by presenting the formal basis for these concepts. The definitions are illustrated in the context of the specification language *Z* [8].

To arrive at this end, we first try to disassemble the specification language such that those elements that bear elementary semantics are identified. Then mechanisms have to be identified to relate these semantic units (*primes*). We refer to this mechanism as *augmented specification relationship nets* or *ASRN*s. The *ASRN* is built from a graph containing the textual information contained in a given specifications, the *SRN*. *SRN*s are augmented by arcs relating concepts that are related due to the semantics contained in a specification. As the paper goes along, the concepts discussed are illustrated using the birthday-book example (BB) given in [8].

SPECIFICATION ABSTRACTIONS

The postulate ”*Formal specifications are expressions written in a formal language such that the primitives of the language have a formally defined semantics.*” can be seen as a very general statement about specifications. It suffices as point of departure for dis-aggregating the information contained in specification and for mapping it into explicit structures.

Apparently, each specification language has linguistic elements at lower granules than postulated in the previous sentence. We refer to them as *literals* of the language. Examples of literals are identifiers, linguistic operators, e.t.c.

Minimal linguistic expressions that can be assigned meaning are called *prime objects* or *primes* for short. They are syntactic elements of a specification with formally defined semantics. Primes are constructed from literals and form the basic entities of a specification. Examples are predicates or expressions. But particular specification languages will allow also for semantically richer primes. E.g. in *Z*, a schema or a generic type can be seen as a prime. These higher level primes are defined arrangements of literals and other primes. The important aspect is that prime objects are immutable as far as they constitute fundamental units (states and operations) specifications are built upon and that they constitute a consistent syntactic entity with defined semantics.

To support the comprehension process, independent prime objects, even if they might get complex, are usually insufficient. The peruser of a specification usually needs more than a given prime but less than the full fledged specification to get an answer to the problem at hand. Usually, one needs a set of different primes which are to be found not necessarily in textually contiguous parts of the specification. We refer to such a set, falling short of being the full specification and falling short of any syntactical constraints observed between different primes as *specification fragment*.

With these definitions, specification languages are structured into elements having defined semantics on their own (primes) and elements with obtain their semantics from their arrangement in a broader context. The crucial issue now has to be:

> *Given an element with defined semantics (a prime), what other elements are needed so that the person having a certain maintenance problem obtains enough information about this problem?*

Apparently, the answer to this question depends on the specific problem at hand. In this paper, we formally define specification slices and specification chunks as first order alternatives to answer this question. Further analysis and improvements on the selection mechanism of the prototyping tool developed will allow also for other, possibly more focussed questions to be addressed against a specification.

In our aim to slice specifications, one has to see that the primes making up a specifications are not lined up by control flow. Nevertheless, one has to recognize that parts of a specification are in essence controlled by other parts of it. In languages where pre-conditions are explicitly highlighted, this is evident. In other languages, such as *Z*, one may resort to theorem proving techniques to identify pre-conditions. In order to have an efficient translation mechanism, we refrain from this and use the heuristic that primed variables or variables decorated with "!" are contained in post-conditions, while primes containing only undecorated variables or input variables constitute pre-conditions.

Specifically with respect to specification slicing, this work builds on earlier work. In 1993, Oda and Araki [7] first used static slicing techniques for analyzing specifications. Their work is based on a simple definition of data-dependency. One year later, Chang and Richardson [3] extended this idea by introducing dynamic specification slicing.

The original idea of slicing goes back to the PhD-thesis of Weiser [10]. It is based on static data-flow analysis (flow-graph), allowing to find the slice in linear time as the transitive closure of a dependency graph. Following the definitions of Weiser, a slice S is defined by a program's subset of statements and control predicates, that are dependent on a slicing criterion (the point of interest). The slicing criterion, in general, is a pair consisting of a line-number and a set of variables. The calculated slice must preserve the effect of the original program on these variables at the given line number.

There are several approaches for slicing programs (see e.g. the survey paper of Frank Tip [9]). But program slices are computed by analyzing data and control dependencies in code. As there are no evident control dependencies in specifications and as the slicing criterion cannot be directly transferred from texts containing programming code to texts containing software specifications, none of the approaches applies directly. But following the reasoning behind program slices, specification slices can be defined as follows:

> *A specification slice is a syntactically and semantically correct specification, which is the result of adding those primes to an (initially empty) specification, that are directly or indirectly contributing to the slicing criteria.*

With this preparatory remarks, we are ready to define a mechanism to carve out slices from an existing, large or very large, but formally correct specification. In contrast to the definition in [3], this definition constructs slices in a bottom up manner. We do so in order to assure that each slice derived has well defined semantics. Starting with a prime in the slicing criterion, the slice has this prime's semantics. In aggregating further primes properly, the remaining specification fragment will always have defined semantics. In a process that deletes "the parts not needed" either the word "needed" gets very complex semantics or the semantics of the torsal specification cannot necessarily be given.

IDENTIFICATION OF ABSTRACTIONS

The identification of lower-level patterns in specifications can be reduced to the identification of strongly related, i.e. syntactically or semantically dependent parts in the specification. Higher-level patterns represent units, that describe states and operations of the specification mixed with control and data-flow information. For that reason, control and data dependencies have to be taken into consideration.

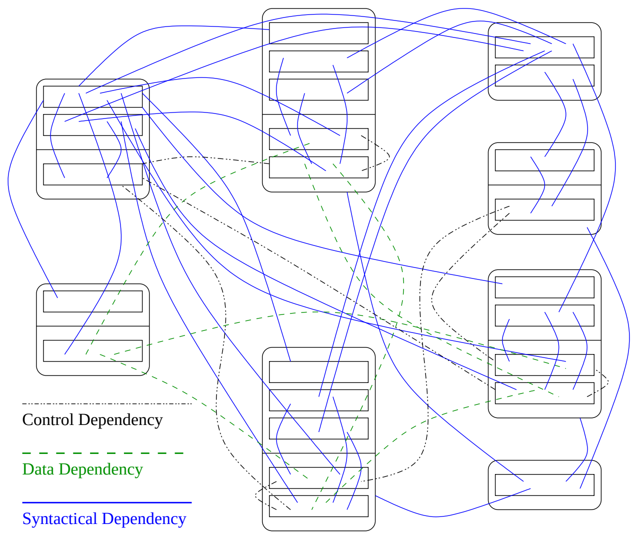

Figure 1. Direct Dependencies in the Z-specification of the birthday book (see also appendix A).

Due to the compact nature of specifications, dependencies are hard to identify. Even if specifications are small, the number of dependencies can be rather large, as Figure 1 shows on the example of the well-known Birthday-book specification [8]. While the BB-specification is simple enough that nobody would bother about it, Figure 1 shows already a lot of intertwined relationships. It is easy to project from there, how tightly interwoven the dependencies between primes are in those situations, where the complexity of the problem calls for a formal specification. Even when only considering structural, control-, and data-flow dependencies, it is hard to isolate "closely related" parts in this specification. Nevertheless, the problem can be solved in the following way:

- Defining suitable dependencies between prime objects in this specification. In this process, like in programming languages, control AND data dependencies are of interest.
- Using the dependencies to define filter functions that can be applied onto the specification.

Based on given dependencies and filter functions, it is possible the generate low-level specification patterns, but also specification chunks and specification slices. We will next present a mechanisms, how chunks and slices, to be understood as predefined patterns, can be isolated.

Programs and Specifications

At first glance the idea of applying decomposition techniques onto specifications seems to be nothing new or challenging. Abstract refinement is state-of-the-art in the world of programming languages and has been applied in debugging, maintenance and testing. Seen carefully though, programs and specifications are different. Mapping decomposition concepts from programming languages to declarative languages is a challenging part. With imperative programs, an explicit flow of control is given. Line-numbers represent an explicit order among statements or other constructs. Hence, the well-known technique of constructing a PDG can be applied. On the other hand, when looking at declarative specification languages like VDM [4] or Z [8], there is no explicit flow of control. Hence, the notion of data-dependency has to be re-interpreted. Thus, when talking about the decomposition of a specification, one has to be careful in putting specifications on a par with imperative programs.

This will be done in the sequel by the definition of a graph, referred to as specification-relationship net, (Definitions 1 to 6). This net has to carry the full information contained in the original specification, both in terms of its direct as well as in terms of its implied semantics. Thus, this properly augmented graph builds the basis out of which semantically meaningful fragments of specifications can be carved. In the context of this paper, we identify specification chunks and slices as prototype of such fragments (Definitions 10 to 12). Definitions 7 to 9 are needed to establish the dependencies that define them. Specific maintenance tasks might warrant different end results as those defined in defs. 11 and 12. The illustrative example should show, how these results can be automatically translated back to the syntactic form software professionals are familiar with. An assessment of this approach is given in [6].

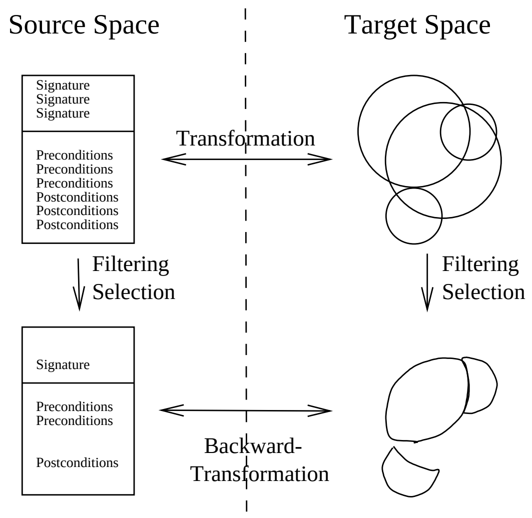

Figure 2. Using transformations and filtering functions in order to support the comprehension process.

Transformation

To identify different dependencies, it is useful to convert a given specification into another type of representation (annotated graphs). The basic idea is that the transformed representation can be analyzed more easily such that only relevant parts can be selected (filtered). The transformation of a specification into another representation, the target space (Fig. 2) is needed to make explicit dependencies that are only implicit in the original specification. Only on this basis, filter and selection functions can automatically construct a somehow minimal context around the prime constituting the slicing-/chunking-criterion. The resulting sub-graph is then transformed back to the source space, leading to chunks or slices.

Definitions

The definition of *control dependency* can be kept rather simple when based on pre- and post-condition analysis. For Z, it might read:

> *A predicate q (the prime object) is control-dependent on a prime p, if p potentially decides whether q is applied or not.*

In the case of Z, the operation schemas are split into pre- and post-conditions. Post-conditions are said to be control-dependent on their pre-conditions. The same idea holds, if elements of the specification are logically combined. If, again in Z, a schema object p is conjuncted to a schema object q, all post-conditions of p are control dependent on the pre-condition of p and q, and all post-conditions of q are control dependent on the pre-condition of q and p.

The definition of *data dependency* is similar:

> *A predicate (a prime) q is data-dependent on a predicate p, if data potentially propagates from p to q through a series of state changes.*

The specification is transformed into a representation called *specification-relationship net* (*SRN*). The *SRN* contains all information that is textually contained in the specification (including information about layout and spacing). We refer to the information contained in the *SRN* also as *syntactical structural dependencies*. The *SRN* is further augmented by *semantical structural information*, yielding an *ASRN*. Such extensions are dependencies between primes due to *type* and *variable declarations*, as well as definition and use *(def/use)-information*. In the sequel, the definitions of the *SRN* and *ASRN* are given.

Definition 1: A *bipartite graph* is an ordered pair (V, A), where V is a finite set of elements called *vertices*, and A is a finite set of elements of the Cartesian product $V \times V$, called *arcs* (A is a binary relation on V, $A \subseteq V \times V$). A *simple bipartite graph* is a bipartite graph (V, A), such that no $(v, v) \in A$ for any $v \in V$. For any arc $(v_1, v_2) \in A$, v_1 is called the *initial vertex* of the arc and v_2 is called *terminal vertex* of the arc.

Definition 1 is the general definition of a bipartite graph. As arcs in an *SRN* can belong to different classes, the definition of an arc-classified bipartite graph is given in definition 2.

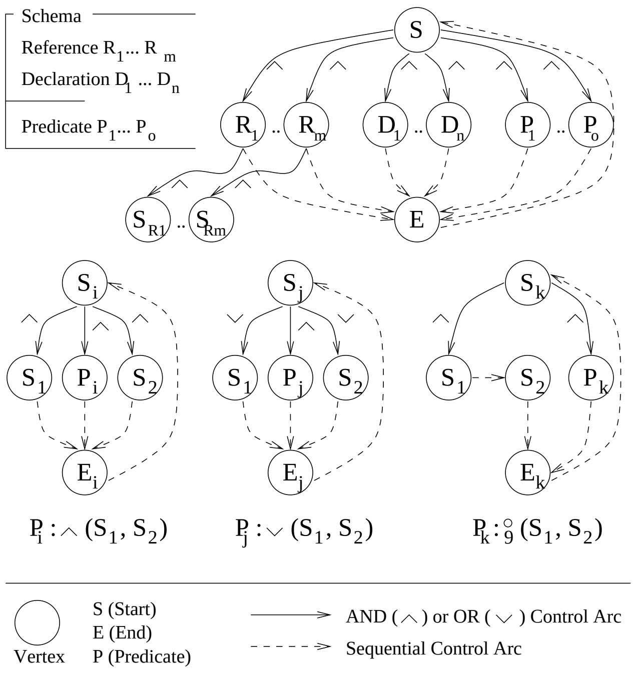

Figure 3. Major transformations for simple Z-specifications into a specification relationship net. Prime objects are enclosed between start and end vertices. Logical combinations are enabled by AND and OR control arcs.

Definition 2: An *arc-classified bipartite graph* is an n-tuple $(V, A_1, A_2, ..., A_{n-1})$ such that every pair (V, A_i) $(i \in \{1..(n-1)\})$ is a bipartite graph and $A_i \cap A_j = \varnothing$ for $i, j \in \{1..(n-1)\}$, and $i \neq j$. A *simple arc-classified bipartite graph* is an arc-classified bipartite graph $(V, A_1, A_2, ..., A_{n-1})$, such that no $(v, v) \in A_i$ $(i \in \{1..(n-1)\})$ for any $v \in V$.

Definition 2 forms the basis, upon which the specification relationship net, a bipartite graph containing arcs and vertices of special (but mutual exclusive) classes, can be defined:

Definition 3: A *specification relationship net (SRN for short)* of a specification is an 8-tuple $(V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ where $(V, A_c, A_{and}, A_{or})$ is a simple arc-classified bipartite graph. $V_{lit} \subset V$ is a set of vertices, called literal vertices such that $V_{lit} = \{v \mid v \textit{ represents a literal of the specification}\}$. $V_{start} \subset V$ where $V_{start} \cap V_{lit} = \varnothing$ is a set of vertices, called start vertices, $V_{end} \subset V$ where $V_{end} \cap V_{lit} = \varnothing$ is a set of vertices, called end vertices, $V_{start} \cap V_{end} = \varnothing$, $A_c \subset V \times V$, $A_{and} \subseteq V \times V$, $A_{or} \subseteq V \times V$. Vertex $t \in V$ is called *totality vertex*. It is a unique vertex, such that the *in-degree*$(t) = 0$. Any arc $(v_1, v_2) \in A_c$ is called a sequential control arc, any arc $(v_1, v_2) \in A_{and}$ is called an AND-control arc, and any arc $(v_1, v_2) \in A_{or}$ is called an OR-control arc.

Fig. 3 shows how some *Z*-constructs are translated to *SRN*-primitives. The general structure is that the elements contained in a prime are embedded by start- and end-nodes. Primes that consist of other primes (such as *Z*-schemata contain vertices that reference those primes. I.e., each prime is represented in the *SRN* by a relatively trivial directed graph. The whole specification is rooted in the totality vertex t.

The *SRN* is defined independent of particular specification languages. Hence, it is general. However, the rule set to translate a specification in a syntax- and semantic-preserving manner to an *SRN* has to be defined in a language dependent manner. As our current work focuses on *Z*, the transformation tool parses *Z*-sources to develop an *SRN* and augment it to an *ASRN*. However, as such a parser could also be written for other formal specification languages, the approach is not inherently dependent on *Z*.

The *SRN* can be augmented by information to be inferred from semantical structural definitions of the respective specification language. The particular nature of such dependencies is again dependent on the particular specification language under consideration. Therefore, we treat this as distinct augmentation step. The interested reader will note, however, that the definitions are coined in such a way that it suffices to resort to compiler technology when building those semantical structural augmentations. Thus, the order of presentation is not to suggest that a multi-phased algorithm is necessary to compute the *ASRN*.

Definition 4: An *augmented specification relationship net (ASRN)* of a specification is a 6-tuple $(N_{SRN}, \Sigma_v, T, C, D, U)$ where N_{SRN} is a specification relationship net $(V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$, Σ_v is a finite set of symbols called variables, and $D : V \rightarrow P(\Sigma_v)$, $U : V \rightarrow P(\Sigma_v)$, $T : V \rightarrow P(\Sigma_v)$ and $C : V \rightarrow P(\Sigma_v)$ are four partial functions from V to the power set of Σ_v. The function $D(v)$ leads to the set of all variables, that have a value assignment (are defined at v), $U(v)$ leads to a set of variables, that are used at vertex v. Thus, D and V serve to compute def-/use relationships. Function $T(v)$ (type def.) leads to the set of all variables at vertex v, that are syntactically declared. The function $C(v)$ (channel def.) leads to a set of values, that are syntactically defined (variable declaration).

An augmented *SRN* is an extension of an *SRN*, where vertices are getting attributes describing the “application” of identifiers belonging to that vertices. This “application” can be the (re)definition of variables, the use of variables, a type definition or a(n) (input/output) channel definition. Once again it has to be noted that the identification of a state change strongly depends on the specification language. (When looking at Z, an assignment to an after-state variable or a channel variable indicates a possible state change. For example, in the last prime object v of the BB-Delete schema – the identifier *birthday* – belongs to the set $U(v)$ and to the set $D(v)$, whereas *name* only belongs to the set $U(v)$ of the respective prime.)

As the syntactical structure of the specification is relevant, special operations, that enable the selection of specific paths in the net, have to be defined:

Definition 5: Let $(V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ be a *SRN* of a Z specification and $(v_1, v_2) \in A_c \cup A_{and} \cup A_{or}$. v_1 is referred to as *AND-antecessor* of v_2 if $(v_1, v_2) \in A_{and}$, as *OR-antecessor* if $(v_1, v_2) \in A_{or}$. Otherwise, v_1 is referred to as *antecessor* of v_2.

Definition 6: Let $(V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$ be a *SRN* of a Z specification and $(v_1, v_2) \in A_c \cup A_{and} \cup A_{or}$ and $(v_1, v_3) \in A_c \cup A_{and} \cup A_{or}$. v_2 is said to be an *AND-neighbour* of a vertex v_3 iff $((v_1, v_2) \in A_{and}) \wedge ((v_1, v_3) \in A_{and}))$. Iff $((v_1, v_2) \in A_{or}) \wedge ((v_1, v_3) \in A_{or}))$, v_2 and v_3 are OR-neighbours.

In this graph, vertices can have antecessors and neighbours. If we are only looking at antecessors that are reachable via AND-control arcs, we are looking at so-called AND-antecessors (likewise for OR-antecessors). If neighbours of the vertex are reachable via AND-control arcs, we refer to them as AND-neighbours (analogously for OR-neighbours).

Based on the definitions of antecessors and neighbours, syntactical dependencies between vertices can be defined. This provides for definitions of control and data-dependencies in *ASRN*s:

Definition 7: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification, $(N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$, and u and v be any two vertices of the net $(u \neq v)$. Vertex (prime) u is said to be *semantically structural dependent* on v, iff there exists at least one variable z such that $z \in U(u)$ and $z \in T(v) \cup C(v)$, and v is an AND-neighbour of u or there exists a path p $(p \in A_{and} \cup A_{or})$ from an AND- or OR-antecessor of u to v such that there exists no other path p_2 with length shorter than the length of p.

One should note that definition 7 does not refer to deep semantics of a specification. From a specifiers perspective, it shows about those dependencies that a good *Z*-checker would identify. From the parsers perspective, however, these dependencies are beyond simple syntax. We tried to accommodate both kind of readers by referring to these dependencies as *semantically structural*. Type-dependencies and channel-definitions (variable declarations) are usually dependent on some structural constraints of the respective specification language.

The application-semantics of a specification, in contrast are expressed by *data dependencies* and *control dependencies*. Their representation in the *ASRN* is defined in definitions 8 and 9:

Definition 8: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification, $N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$, and u and v be any two vertices of the net $(u \neq v)$. Vertex u is said to be *data dependent* on v, iff there exists at least one variable z such that $z \in U(u)$ and $z \in D(v)$ and $z \notin T(v)$, and there exists a path $p \in A_c \cup A_{and} \cup A_{or}$ from v to u without definition of z between u and v, and there is no AND-neighbour u_1 of u with $z \in T(u_1) \cup C(u_1)$.

Definition 9: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification, $N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$, and u and v be any two vertices of the net ($u \neq v$). Vertex u is said to be *control dependent* on v, iff $D(v) = \varnothing$ and $D(u) \neq \varnothing$ and $C(v) = \varnothing$ and $T(v) = \varnothing$ and v is an AND-neighbour of u or there exists a path p ($p \in A_{and} \cup A_{or}$) from an AND-antecessor of u to v.

With an *ASRN* containing all those dependencies that are inherent in the semantics of a textual specification, it is possible to define for a specification chunks and slices. Chunks and slices are defined on a specific point of interest in the specification. This point of interest is called abstraction criterion and is defined as follows:

Definition 10a: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification, $N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$. An *abstraction criterion* for a specification is a triple (l, Θ, Γ), where l is a literal in the specification (represented by vertex $v_l \in V$). Further holds $\Theta \subseteq \Sigma_v$ is a set of variables defined at literal l, and Γ is a set of dependencies, $\Gamma \subseteq \{C, D, S\}$, with C denoting control dependency, D denoting data dependency and S denoting syntactical dependency.

We now first address the definition of chunks. In contrast to slices, chunks are rather an intuitive concept, rooted in sufficiency of the context around a focussed point of interest. This point of interest, however, might be defined in a weaker manner than with slices. Hence, we first weaken definition 10 slightly, giving definition 10b:

Definition 10b: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification, $N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$. A *chunking criterion* for a specification is a triple (PL, Θ, Γ), where PL is a set of primes and literals $P \cup L$ appearing within some uniquely identifiable prime p in the specification (represented by vertex $v_p \in V$) with $V_P \subseteq V$ and $L \subseteq \Sigma_v$. Further holds $\Theta \subseteq \Sigma_v$ is a set of variables defined at literal l, and Γ is a set of dependencies, $\Gamma \subseteq \{C, D, S\}$, with C, D, S denoting control-, data-, or syntactical dependency respectively.

Thus, the chunking criterion according to def. 10b postulates a specification fragment contained within a common context (prime p). According to Burnsteins original idea, we define in definition 11b a *B-chunk* as the immediate context of this specification fragment according to some filter criterion:

Definition 11b: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification S and $N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$. A static *B-chunk* referred to as $BChunk(PL, \Theta, \Gamma)$ of S on chunking criterion (PL, Θ, Γ) (where $v_p \in V$ and $V_P \subseteq V$ and $\Gamma \neq \varnothing$) is a subset of vertices $V_{PL} \subseteq V$ such that for all $v \in V$ holds: vertex $v \in V_{PL}$ iff v is either directly data-dependent with respect to the variables defined in Θ on V_P and $D \in \Gamma$, or directly control-dependent on V_P and $C \in \Gamma$, or syntactical dependent on V_P and $S \in \Gamma$.

This definition provides the immediate context of the chunking criterion. However, it is not safe in the sense that primes outside of this context might still indirectly influence this chunk. Likewise, the chunk might have secondary effects beyond those elements of the specification it affects directly. Hence, we propose definition 11a, the *full specification chunk*. This definition is the transitive closure over the B-chunk, defined in 11b. However, to focus the point of departure, we use now a more compact chunking criterion, i.e. definition 10a for the abstraction criterion.

Definition 11a: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification S and $N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$. A *full static chunk* referred to as $SChunk(l, \Theta, \Gamma)$ of S on abstraction criterion (l, Θ, Γ) is a subset of vertices of $V_l \subseteq V$, such that for all $v \in V$ holds: vertex $v \in V_l$, iff l is either data-dependent with respect to the variables defined in Θ on v and $D \in \Gamma$, or control-dependent on v and $C \in \Gamma$, or semantically structural dependent on v and $S \in \Gamma$.

As this chunk extends over the whole specification, it can be directly used in the definition of slices. If all types of dependencies are included in the abstraction-criterion of the full static chunk, the chunk equals a *static specification slice*:

Definition 12: Let $(N_{SRN}, \Sigma_v, T, C, D, U)$ be an *ASRN* of a Z specification, $N_{SRN} = (V, V_{lit}, V_{start}, V_{end}, A_c, A_{and}, A_{or}, t)$. A *static slice* $SSlice(l, \Theta)$ of a specification on a given abstraction criteria (l, Θ) equals to the full static specification chunk $SChunk(l, \Theta, \{D, C, S\})$.

As can be seen, a specification slice is nothing else than a special case of a specification chunk. It is a full static chunk, containing all types of defined dependencies. As syntactical dependencies are included in the slice, the slice stays syntactically correct and understandable. As the vertices identified in definitions 11(a, b) or 12 are vertices from the *SRN*, the full information about the textual representation of the specification contained in the *SRN* can be used in the backwards-transformation from the *ASRN*-slice/-chunk to the respective specification-slice/-chunk.

Definition 10 enables the abstraction criterion to be any identifier used at a specific vertex in the *ASRN* representation. In fact, the choice of the identifiers and vertices in combination with the underlying specification language strongly influences the resulting specification abstractions. It is possible to slice a specification with respect to different identifiers in the state space and the resulting slices can be analyzed in relation to their disjoint and interleaving specification fragments.

Chunks enable the reader to scope on a rather narrow domain, highlighting specific parts in a specification. In contrast, slices focus on a broader domain, as they always include all types of dependencies. The advantage is, that slices are understandable in the specific domain. However, as with code-slices, specification slices can become rather big. With highly interwoven specification, the slice might range over the full specification. Experiments have shown though [6], that even with relatively small specifications, substantial reduction of size can be witnessed.

Example

To illustrate the approach, we refer to the birthday book in appendix A. If an author wants to understand the definition of the state-space identifier *birthday* at line 12 in the Add Operation schema, the prime starting with *birthday'* = ..., the slicing function $SSlice(v_{12}, \{birthday\})$ should be applied Applying this filter leads to a subset of nodes that can be transformed backward, yielding the Z-specification shown in Appendix B. Fig. 4 shows the *ASRN* of the original specification. Fig. 5 highlights only the nodes partaking in the slice. As these nodes are identified in the original *ASRN*, all link information needed to recreate the Z-representation is still available. Figs. 4 and 5 show also how specification languages hide complexity. Even with this trivial example, the relationships between primes almost don't fit on one page. The resulting fragment contains only the four schemas *BB*, *InitBB*, *Add* and *Delete*. It describes a view on the database, where insert and delete operations are possible. Report and Success operations are eliminated. In this small example, helping the user to focus on the point of interest, the state-variable birthday, is of moderate significance. In specifications with the number of primes being several orders of magnitude larger, this yields substantial savings in specification comprehension though. Depending on the type of fragments to be constructed, reductions in size and complexity between 10 to 75 percent are possible [6].

Imagine, that the specification of the birthday book contains the state-variable "presents", in order to represent a list of presents, a person already has got so far.

$[PRESENT]$

BBP

$$\begin{array}{|l}
known : \mathbb{P}\ NAME \\
birthday : NAME \nrightarrow DATE \\
presents : NAME \times PRESENT \\
\hline
known = \mathrm{dom}\ birthday
\end{array}$$

Furthermore, the operation schema "AddPresent" is added to the specification, a schema, that adds presents to the relation called "presents":

AddPresent

$$\begin{array}{|l}
\Delta BBP \\
name? : NAME \\
pres? : PRESENT \\
\hline
name? \in known \\
presents' = presents \cup (name? \mapsto pres?)
\end{array}$$

$$FunctioningDBP == ((Add \wedge Success) \vee (Delete \wedge Success)) \vee Find \vee AddPresent$$

The generation of all possible slices for all of the identifiers in the state space (in the BirthdayBook example these are $SSlice(InitBB, known)$, $SSlice(Add, known)$, $SSlice(Add, birthday)$, $SSlice(Delete, known)$, $SSlice(Delte, birthday)$ and $SSlice(AddPresent, presents)$) shows that *AddPresent* is the only operation which modifies the state variable *presents*. In contrast, *known* is included in nearly all schema operations, and birthday is modified only in the *Add* and *Delete* schema.

CONCLUSION

Based on the argument that formal specifications are useful not only during initial software development but also during maintenance, concepts helping maintainers to obtain a focussed understanding of specifications have been presented. The concepts have been shown on two prototypical specification fragments, chunks and slices.

The notions presented are formally defined. The tool prototype for specification chunking and specification slicing has been developed for *Z*. The concepts generated can be extended to other formal specification languages though.

REFERENCES

[1] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software Systems*. John Wiley & Sons, Inc., 1995.

[2] I. Burnstein, K. Roberson, F. Saner, A. Mirza, and A. Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In *TAI-97, 9th International Conference on Tools with Artificial Intelligence*. IEEE press, November 1997.

[3] J. Chang and D. J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[4] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 2nd edition, 1990.

[5] R. Mittermeir, A. Bollin, H. Pozewaunig, and D. Rauner-Reithmayer. Goal-driven combination of software comprehension approaches for component based development. In *Proc. Symposium on Software Reusability*, ACM Press, pages 95–102, May 2001.

[6] R. T. Mittermeir and A. Bollin. Demand-driven Specification Partitioning: Concepts to support mastering the complexity of specifications. In *Proceedings 5th Joint Modular Languages Conference*. Springer, to appear, 2003.

[7] T. Oda and K. Araki. Specification slicing in a formal methods software development. In 17th *Annual International Computer Software and Applications Conference*, IEEE Computer Socienty Press, pages 313–319, November 1993.

[8] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2nd edition, 1992.

[9] F. Tip. A Survey of Program Slicing Techniques. Technical report, CWI Netherlands, 1994.

[10] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[11] M. Weiser. Program slicing. In *Proceedings of the* 5^{th} *International Conference on Software Engineering*, pages 439–449. IEEE, 1982.

APPENDIX A - BIRTHDAYBOOK

$$[NAME, DATE]$$
$$REPORT ::= OK \mid NOK$$

$$\begin{array}{|l} \textit{BB} \\ \hline known : \mathbb{P}\, NAME \\ birthday : NAME \nrightarrow DATE \\ \hline known = \operatorname{dom} birthday \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{InitBB} \\ \hline BB \\ \hline known = \varnothing \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{Add} \\ \hline \Delta BB \\ name? : NAME \\ date? : DATE \\ \hline name? \notin known \\ birthday' = birthday \\ \qquad \cup \{name? \mapsto date?\} \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{Delete} \\ \hline \Delta BB \\ name? : NAME \\ \hline name? \in known \\ birthday' = birthday \setminus \\ \quad \{name? \mapsto birthday(name?)\} \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{Success} \\ \hline result! : REPORT \\ \hline result! = OK \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{Find} \\ \hline \Xi BB \\ name? : NAME \\ date! : DATE \\ \hline name? \in known \\ date! = birthday(name?) \\ \hline \end{array}$$

$$FunctioningDB == \\ \quad ((Add \land Success) \lor (Delete \land Success)) \lor Find$$

APPENDIX B - SSLICE(12,BIRTHDAY) AND ITS ASRN REPRESENTATION

$$[NAME, DATE]$$

$$\begin{array}{|l} \textit{BB} \\ \hline known : \mathbb{P}\, NAME \\ birthday : NAME \nrightarrow DATE \\ \hline known = \operatorname{dom} birthday \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{InitBB} \\ \hline BB \\ \hline known = \varnothing \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{Add} \\ \hline \Delta BB \\ name? : NAME \\ date? : DATE \\ \hline name? \notin known \\ birthday' = birthday \\ \qquad \cup \{name? \mapsto date?\} \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{Delete} \\ \hline \Delta BB \\ name? : NAME \\ \hline name? \in known \\ birthday' = birthday \setminus \\ \quad \{name? \mapsto birthday(name?)\} \\ \hline \end{array}$$

$$FunctioningDB == \\ \quad ((Add) \lor (Delete))$$

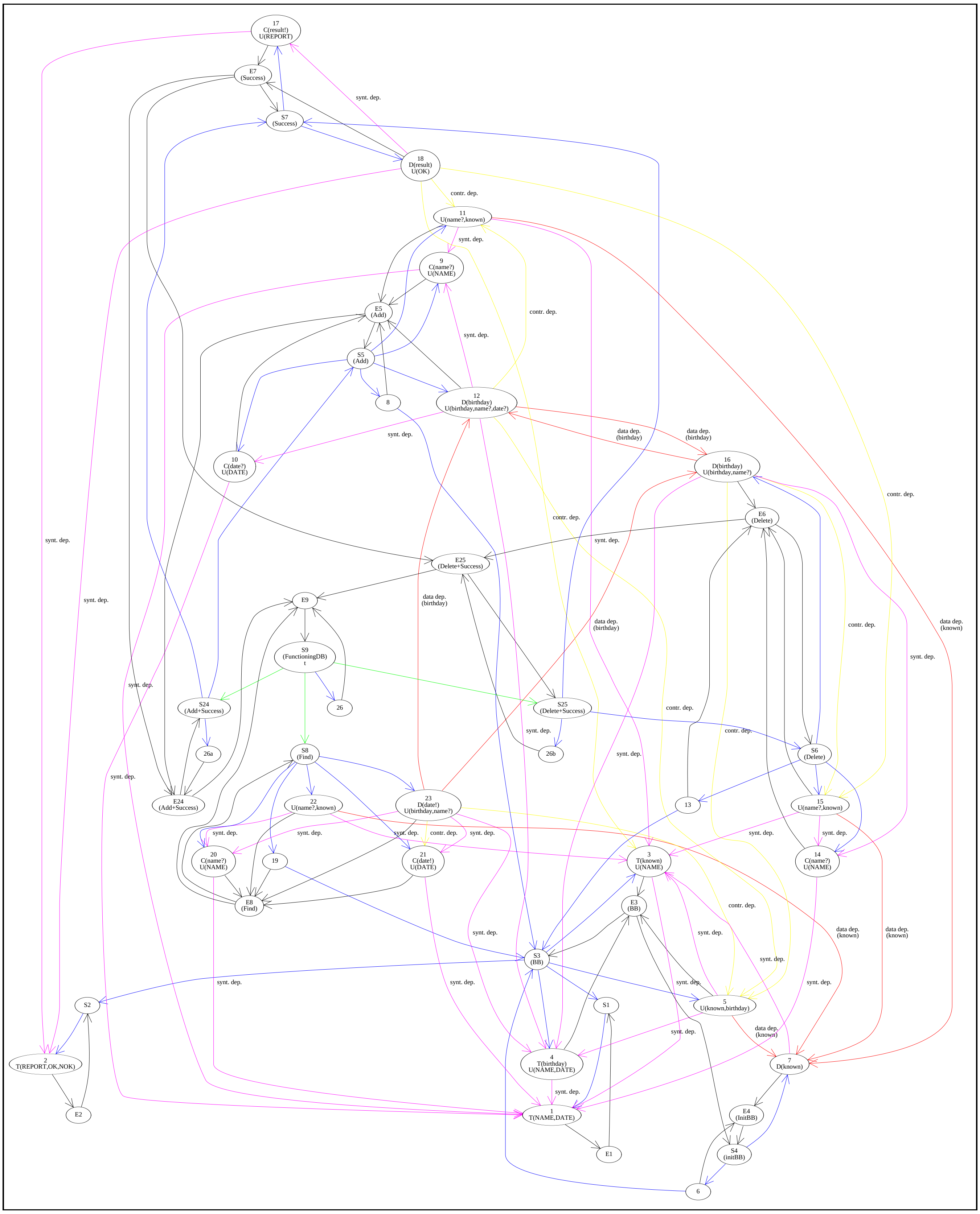

Figure 4. ASRN graph representation of the BB specification *(Control-, data- and syntactical dependencies are included, syntactical comment nodes are omitted. Node v_{12} (middle, slightly above center) represents line 12 of the BB specification, containing the definition of the literal “birthday” and the use of the literals “birthday”, “name?” and “date?”.)*

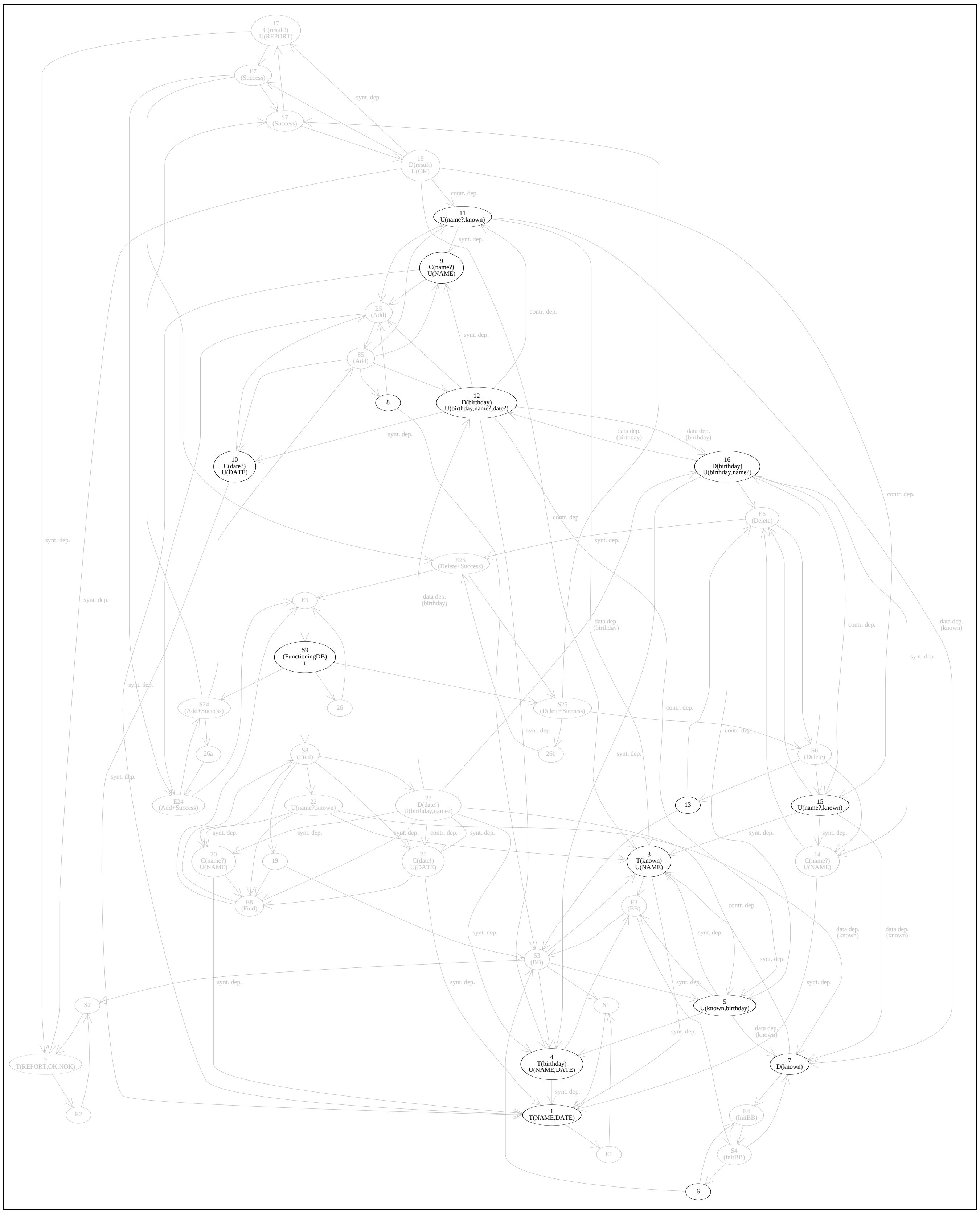

Figure 5. ASRN graph representation of the BB specification after the application of SSlice(v_{12},$\{$birthday$\}$). *(Syntactical comment nodes are omitted. Control-, data- and syntactical dependencies are included but greyed-out in order to stress the reduction of the complexity of the resulting graph. After the application of the slice, several nodes (and related dependencies) are vanishing.)*